Type highlighting

December 9, 2023

Abstract

A brief case study of taking a second look at an object type highlighting strategy, now that Codswallop RPL has improved enough to accomodate more creative methods.

Overview

One of the early goals in Cods was a quality interactive experience, and to that end it was important for the user to be presented with RPL objects in a friendly, readable way. Aside from formatting to fit the screen, code is much more readable with indentation, and with colors to differentiate object types. Separately, code was drawn up to show the contents of the named store in a compact way: a list of names, where each name is colored to match the type of object stored there. The entirety of the highlighting and directory listing code was written when Cods was much younger, and as a result it doesn't take advantage of several new features of the language. It was also written quickly, and with no regard for composite types which came later, such as subdirectories, quoted objects, and tags – these don't break the old highlighting routines, but have very limited support.

Because the original type highlighting scheme was a hurried first crack at the problem, and because the language was not quite to where it is today, it was not worth saving the bulk of that effort. As a result there may be some value in looking at the two starkly different approaches.

The first try

Making and selecting colors

Standard ANSI terminal codes are used to change text color. A function called *XTFORE* selects a foreground color, accepting a color number and turning it into an ANSI color select string. That is Escape+"[38;5;*n*m" where *n* is the color number:

```
:: >STR #1 #4 SUBS "m" + #27 >ASC "[38;5;" + SWAP + ;
```

This is a bit more complex than it needs to be; *>STR* returns "#*n*" and the subsequent *SUBS* strips the leading pound sign. It's possible when this code was written, adding a string to an integer would not have yielded the correct result, but it does now. The Control Sequence Introducer, Escape-[, is also stored as *COLOR.CSI*, so a more compact and legible *XTFORE* could be:

```
:: "m" + COLOR.CSI "38;5;" + SWAP + ;
```

The original highlighter does use a sensible means of storing the type colors. On boot, a list is made of empty strings, one for each registered object type. Then, each object type that has a color assigned uses *XTFORE* to make the required ANSI string and then *PUT* it into the list by name, before the whole list is stored. Then the highlighter can pull color strings from that list according to object type with a straightforward *TYPE* and *GET*.

```
{ "" } Types.n LEN *
#6 COLOR.XTFORE Types.Integer PUT
#2 COLOR.XTFORE Types.Comment PUT
...
'COLOR.TYPES STO
```

This is a straightforward scheme which is carried over wholesale to the new highlighter. As a professional courtesy, *>TYPE* adds an empty string to *COLOR.TYPES* for each new user type, to keep the highlighter from falling on its face.

The grand unified COLOROBJ

In the original scheme, *COLOROBJ* is one unified program which contains several named functions. It isn't very large, but it is tightly wound. In outline, it looks like:

```
:: (Syntax highlighting routine.)
  ':: (newline function) ;
  ':: (mkone function) ;
  ':: (mklist function) ;
  #15 (Starting background color)
  COLOR.margin (Screen width)
  #0 (Starting indent)
  #0 (Starting line position)
  ':: "" SWAP mkone + COLOR.ANSINORMAL SWAP + COLOR.ANSINORMAL + ;
  { linepos indent margin bgcolor mklist mkone newline } LOCAL ;
```

To actually highlight a single object, the code here calls *mkone* with a blank string and the object to be highlighted. *mkone* returns a colorized string, and *COLOROBJ* then bookends it with *COLOR.ANSINORMAL*, to ensure no colors escape it.

The newline function

When the line needs to wrap or some suitable space is needed for legibility purposes, *newline* is the function. Because *COLOROBJ* only generates a single output string, and doesn't print anything, *newline* appends line breaks within that string. Its one trick is that if the current cursor position is equal to the current indent, it suppresses a newline. In either case, it handles the background color; if a new line is actually generated, it also inserts as many spaces as needed for the current indent, and resets the cursor position:

```
:: (newline: Create a string representing a new indented line.)
    indent linepos ==
    ':: bgcolor COLOR.XTBACK ;
    ':: #0 COLOR.XTBACK COLOR.ANSIENTER + " " indent * + bgcolor COLOR.XTBACK + ;
    IFTE
    indent 'linepos STO ;
```

The mkone function

As you've deduced from above, *mkone* accepts a string in progress and an object to highlight. It has to do several things including looking up the per-type color, and sensibly wrapping lines according to the printed length of the object. First it gathers the type, saves it, and determines if it's a list or code object, which must be dealt with separately:

```
:: (mkone: Color one object.)
  DUP TYPE
  DUP DUP `Types.List == SWAP `Types.Code == OR
```

In the case of lists and code, it first discards the type number, then forces a newline, increments *indent* by two characters and the background color by one color number, and calls *mklist*. Then it decrements *indent* and forces another new line:

```
':: DROP SWAP newline + SWAP indent #2 + 'indent STO bgcolor #1 + mklist +
    indent #2 - 'indent STO newline + ;
```

Whereas for all other types, it does a bit more work. It first turns the object into a string, then catches its length to see if the line needs to wrap, but it won't force a new line if it's already at the beginning of one. The resulting *IFT* condition ((cursor+length)>margin AND cursor != indent) was very hard to write as the comments show, and it is now also hard to read as well:

```
  ':: COLOR.TYPES SWAP GET SWAP >STR
     DUP LEN DUP linepos + margin > (Have we ventured past our margin?)
     linepos indent == NOT AND (Aaaaaaa)
```

In the case of a line wrap, the new cursor position is the string's length, plus one for a separator space, plus the current indent. Otherwise, the cursor position is incremented and the string appended without a newline, and normal text color is restored at the end:

```
'::  indent + newline SWAP #1 + 'linepos STO SWAP + + " " + ;
        ':: linepos #1 + + 'linepos STO + " " + ;
        IFTE ;
  IFTE
  COLOR.TYPES #0 GET + ;
```

The mklist function

If mkone encounters a list or a code object, it's handed off here. It opens with some jiujitsu to insert into local variables: an index, the object type, its length, the object itself, and the background color. While *indent* is incremented and decremented manually by *mkone*, *bgcolor* is incremented in *mkone* and stored as a local variable here. This covers the original *bgcolor* value here and for any further recursion.

```
:: (mklist: Color a composite object.)
  SWAP DUP DUP LEN SWAP TYPE #0
```

Then we get to the local variable context. First up is to choose and color a delimiter based on the object type. This could have been done with *IFTE*, but my guess is that *KCASE* was a new and fancy toy at the time, and felt like it would be a good way to add support for other delimiters later:

```
  ':: lstype
    { { :: `Types.List == ;
        :: COLOR.TYPES `Types.List GET "{ " linepos #2 + 'linepos STO ; }
      { :: `Types.Code == ;
        :: COLOR.TYPES `Types.Code GET ":: " linepos #3 + 'linepos STO ; } }
    KCASE + bgcolor COLOR.XTBACK SWAP +
```

Then, if the list actually has a length, this very simple loop rummages through it and calls *mkone* on every object in turn. If that object is another list, *mkone* will recurse with another call to *mklist* as needed:

```
    lstlen
    '::
      '::
        'lst RCL index GET mkone +
        index #1 + DUP 'index STO
        lstlen < ;
      REP ;
    IFT
```

Once the list has been thoroughly rummaged through, one more *KCASE* is used to select the closing delimiter, and it returns with just the string it was given, appended with whatever work was done.

```
lstype
{ { :: ‘Types.List == ;
    :: COLOR.TYPES ‘Types.List GET "}" ; }
  { :: ‘Types.Code == ;
    :: COLOR.TYPES ‘Types.Code GET ";" ; } }
KCASE + ;
{ index lstype lstlen lst bgcolor } LOCAL ;
```

Directory colorization

Colorizing the named store was needlessly complicated, implemented in two nearly identical functions, names and dirnames, which colorized the root directory and a subdirectory respectively. In contrast to *COLOROBJ*, these two functions print one line at a time as they're generated. *COLOROBJ* itself has no support for directories, so if it encounters one, it will return a pretty blue string declaring "[directory]".

By the time this was last touched, *LOCAL* accepted both objects off the stack and also tagged objects within the local variable list. This makes things a bit easier to read, but because it was still written in a hurry, it's uncommented. It also doesn't use a tag for the empty list *dirs*, putting it on the stack first:

```
:: (Generic names highlight routine.)
  NAMES DUP LEN {}
```

And here we're in the local variable context. *dirs* is a list to which subdirectories are added as encountered. *length* is the number of names in the list; *ournames* is the list; *index* is our place within the list; and *linepos* and *itemlen* are the cursor and name length respectively. The loop requires a string, so it gets a little header here to start banging on:

```
':: "Current names:" COLOR.ANSIENTER +
```

This loop rummages through the list of names, first fetching a name, incrementing the cursor, and checking to see if that exceeded the margin. If it did, the current string is printed, the cursor reset, and a new string started:

```
'::
  ournames index GET DUP
  >STR LEN #2 + DUP 'itemlen STO linepos + DUP 'linepos STO
  COLOR.margin >
  ':: SWAP DISP "" SWAP itemlen 'linepos STO ; IFT
```

Then, the name is recalled, and a color picked according to the type of its contents. If the type is a directory, it's added to the *dirs* list for later.

```
          DUP RCL TYPE
          DUP Types.Directory ==
          ':: dirs ournames index GET + 'dirs STO ; IFT
          COLOR.TYPES SWAP GET SWAP >STR + + COLOR.TYPES #0 GET +
```

Before repeating, a comma and space is added. This is suppressed at the end of the list:

```
          index #1 + DUP 'index STO length <
          ':: ", " + #1 ; #0 IFTE ;
```

Then a trailing period is added and it's done with all the names in the main store.

```
        REP "." + DISP
```

This version doesn't recurse, but it will drop down one level after printing all the names up above, and display the contents of all the directories it found by calling *dirnames* against each one:

```
        #0 'index STO
        ':: index dirs LEN < DUP
          ':: "" DISP dirs index GET COLOR.dirnames
            index #1 + 'index STO ;
          IFT ;
        REP ;
      { dirs length ournames :index:#0 :linepos:#0 :itemlen:#0 } LOCAL ;
```

The only major difference between *names* and *dirnames* is that *dirnames* accepts the name of a directory, uses *DIR* instead of *NAMES* to get a list of its contents, and does some string manipulation to make a dotted name to recall each object:

```
   >STR 'ourdir RCL >STR "." + SWAP + >SYM RCL
```

When this was written, the interpreter actually stored dotted symbols as one string, so this adventurous code snippet was something close to how it would have been done even on the Python side. Since then, the dot notation is just for parsing purposes, and the interpreter internally stores a symbol as a list of strings. Further, + can be used to join two symbols, and directories can be stored as local variables.

Stack colorization

The particulars of the stack colorization routine don't bear repeating here, but it's relatively simple: *COLOROBJ* is called against each object on the stack, up to some limit, and it's printed with the line number attached. Since *COLOROBJ* doesn't know the cursor has moved, that first line can wrap improperly, and when colorizing large objects, there's a noticable delay because nothing is printed until the entire object is colorized (which may be many screens of text.)

The second try

While the first version served well for a long time, it was limited by its hard coding for composite object types, and its general slapdash nature. By the time quote objects were a thing, the original code was barely usable. This led to a rethink of what a modular, expandable highlighter might look like. For example:

- The old way had a generic highlighter calling out to a list highlighter. To really be expandable, a better solution would be to have a list of "ANSI-izers" for each object type, in the same way a list of color strings is kept for each type's color. Then, one "izer" call would quickly select from a list of functions and evaluate the correct one.
- Instead of having a mix of hard-coded and adjustable values for things like indentation and margins, a uniform set of preferences should be available in an obvious place.
- Where previously a highlighter would manage line wrapping and the overall string by itself, a better solution is to store the already-generated text and the current string in process separately. This lets multiple functions work on the same word for objects requiring non-breaking spaces, and a common write call can handle line wrapping.
- While a newline routine would be very similar in either case, it would make sense to be able to replace it: for example, such a routine could write out each printable line as it was finished, or append it to a string to emulate the way *COLOROBJ* works, or redirect its output to a file.
- In general, supplying a common local variable context for any highlighting application is a fine plan, one which would supply reasonable defaults but one which could be overridden as needed, a sort of ad-hoc subclassing for different needs.

To that end, the second try, "New Colors", is a much more creditable work, and one which will last a lot longer than the previous effort.

Making and selecting "ANSI-izers"

Just like the original made a list of colors per object type – a feature retained by New Colors – there is now a list of highlighter functions per object type. Creating it is nearly identical:

```
(And store default colorize routine into our per-type generator list.)
{ ANSI.ize.default } Types.n LEN *
'ANSI.ize.dir   Types.Directory PUT
'ANSI.ize.quote Types.Quote     PUT
'ANSI.ize.list  Types.List      PUT
```

```
'ANSI.ize.code  Types.Code      PUT
'ANSI.ize.tag   Types.Tag       PUT
'ANSI.default.izers STO
```

This list of functions goes into the *ANSI.default* directory, as do several other things. Because the parser can now accept a directory object, the entire tree is created just before the izers list is placed into it. It also contains some useful defaults right up top, such as the margin, recursion depth, tab stop, and the number of stack lines to show when displaying it:

```
(Our base directory tree.)
[dir:
  :codes:   [dir:]
  :ize:     [dir:]
  :default: [dir: :margin: #70
                  :depth:  #20
                  :tab:     #2
                  :stack:   #8 ]]
'ANSI STO
```

The grander, less unified *tocolor*

To print a colorized object, the object is placed on the stack and one call to *ANSI.tocolor* will display it. In its entirety:

```
::
  ':: izer newline ;
  {}
  ANSI.default.environment ;
```

The environment

One of the secret sauces in the new highlighter is the local variable environment in which any application can be run. *ANSI.default.environment* is a short function that works exactly like *LOCAL*, but also prepares all the variables used by the various ANSI-izers and the internal functions. More importantly, any of the defaults can be overridden by including them in the variable list. Anything supplied will cover the defaults, but if nothing is supplied, nothing will be missed, either:

```
::
  { :margin: ANSI.default.margin
    :depth: ANSI.default.depth
    :tab: ANSI.default.tab
    :stacklimit: ANSI.default.stack
    :izers: ANSI.default.izers
    :colors: ANSI.default.colors
    :word: ""
    :length: #0
    :text: ""
    :cursor: #0
    :indent: #0
    :textready: #0
    :newline: ANSI.ize.newline
    :write: ANSI.ize.write
    :izer: ANSI.ize.izer }
  SWAP + LOCAL ;
```

Because local variables are assigned not just for default values, but for common functions and even the whole lists of ANSI-izers and colors, a tremendous amount of flexibility is on offer for special cases without affecting the default setup or requiring much additional code.

The default *izer*

The closest equivalent to the old *mkone* is *ANSI.ize.izer*, which is nearly as simple as the *tocolor* which calls it. It quotes the object to be highlighted (so it can be recalled without evaluating it), stores it in a local variable *obj*, then selects and evaluates a type-specific izer from the *izers* list. It uses the new builtin *GETE*, get-evaluate, which fetches and evaluates an object from a list in one operation.

```
:: QUOTE
  '::
    izers obj TYPE GETE ;
  { obj } LOCAL ;
```

The default *write*

Whenever *word* – whether it contains a complete object, or some part of an object, or several objects – is ready to be written, *write* is called. It handles line wrapping as needed, and recording to the current line of text. Every time it's called, it also clears the current *word* and *length*, and sets the *textready* flag. *textready* is used for suppression of consecutive newlines, instead of guessing based upon cursor position.

```
::
  (See if this object exceeds our margin; if it does, force a new line.)
  length cursor + margin > 'newline IFT
  (Then append the text we received, and update our cursor, clear word,
   and set the text ready flag.)
  text word + 'text STO
  cursor length + 'cursor STO
  "" 'word STO
```

```
    #0 'length STO
    #1 'textready STO ;
```

Because *write*, *word*, and *length* are available anywhere, multiple independent izers can cooperate on composite objects. For example, a quote object izer might add a colorized apostrophe to *word* before evaluating *izer* against the object it's quoting, thus ensuring there's no line break between the quote mark and the object it's attached to.

The default *newline*

Like *write*, *newline* can be called from anywhere to request a line break; also, like its forebear, it will suppress repeated newline calls. Unlike the one in *COLOROBJ*, it doesn't support background color, but it does display and then erase the current line of text, so large objects are shown progressively.

```
::
  (Suppress newlines if no text has been written; otherwise display the
   current text and clear both it and the cursor.  Indent as needed.)
  textready
  ':: text DISP
    " " indent * 'text STO
    indent 'cursor STO
    #0 'textready STO ;
  IFT ;
```

Type-specific ANSI-izers

Having separated out the boring work of picking what to call based on type, and having callbacks for actually recording the generated text and handling line breaks and so forth, the actual type-specific izers are handed a rich working environment, including the object to be highlighted itself, neatly quoted and stored as *obj*. They can recurse as needed, with everything using the same *write* and *newline*, sharing the same *word* and *length*, and they can modify the environment as needed too.

ize.default

Most objects are simple atoms which don't need special treatment, just a color splashed upon their string representation. *default* handles all these cases in a way very similar to *mkone*. One important difference is that when retrieving from the list of colors, *GETE* is now used; the object stored in the *colors* list no longer has to be a string, but can be anything which evaluates to a string.

```
::
  (Fetch our object, turn it into a string.)
  word obj DUP TYPE SWAP >STR
```

```
        (Add its length, and a space, to our word length.)
        DUP LEN length + #1 + 'length STO
        (Get a color for it according to type.)
        SWAP colors SWAP GETE SWAP +
        (And shut off the color, insert trailing space, and add our word.)
        ANSI.codes.nofore + " " + + 'word STO
        write ;
```

ize.quote

One of the motivating factors behind the new scheme was the inability of *COLOROBJ* to usefully handle the new quote objects. This demonstrates how trivial it was to add support for it in the new way:

```
    ::
      (First, add a color quote symbol to our word, and increment length.)
      word colors Types.Quote GETE + "'" + 'word STO
      length #1 + 'length STO
      (Then recall the contents of our object and hand that off.
       It will take care of writing back the completed string.)
      obj EVAL izer ;
```

Because this function doesn't call *write* at all, the leading apostrophe, with its color, prepends whatever its contents writes without a space between or a line break. Deeply nested quotes will naturally recurse, with *izer* calling *ize.quote* calling *izer* and so forth, as many times as needed, and because both functions make tail calls, there's no recursion limit per se (though the named store will get very deep with *obj*s.)

Tags are handled in a similar fashion to quotes, with the tag's name joined with nonbreaking space to the first word of the object it contains. The only difference with tags is they respect the recursion depth limit.

ize.code and ize.list

The rules for drawing code and lists are slightly different, as are the delimiters, but the internals are broadly similar enough that they use the same secondary function for their inner loop. Covering just code here, another trick is evident. First, though, the opening delimiter is written:

```
    ::
      (Force a newline and write our opener in list color.)
      newline
      word colors Types.Code GETE + ":: " + 'word STO
      length #3 + 'length STO
      write
```

Then, instead of decrementing the recursion depth and incrementing the indent on one side, and reversing it on the other, the math is done once and the rest of the routine is evaluated with the new numbers as local variables. This will cover the old *indent* and *depth* both in the routine and also in any routines it calls:

```
(Now add a tab and subtract from our recursion depth for the rest of this.)
  depth #1 - indent tab +
```

A quick check to make sure the code isn't a null list, and if it isn't, a newline is issued (which will respect the new indent level) and another function, *listinnards*, is called with the list length as a convenient argument:

```
  '::
    (Then, if there is anything in it, add a newline and run listinnards.)
    obj LEN DUP
    ':: newline ANSI.ize.listinnards ; 'DROP IFTE
```

Either way, once we're done with whatever we did, the closing semicolon is colored, added, and written:

```
(Finally, write out our closing bracket, again in list color, and a
     newline to keep things tidy.)
    word colors Types.Code GETE + ";" +
    ANSI.codes.nofore + 'word STO
    length #1 + 'length STO
    write ;
  { indent depth } LOCAL
```

Finally the small local context ends, reverting *indent* and *depth* to their previous values, and another newline call brings the cursor back to the non-indented next line before *ize.code* returns.

```
  newline ;
```

ize.listinnards

Since there's significant commonality between lists and code, they both call *listinnards* for the task of highlighting their contents. It's here that recursion depth is checked, and any recursion past the prescribed limit will just show an ellipsis instead of the list's contents, preventing further recursion. The loop itself is very simple, fetching each list object in turn and evaluating *izer* against it:

```
::
  '::
    depth
    (If we still have depth to go, process the list.)
  '::
    '::
       obj idx GET izer
       idx #1 + DUP 'idx STO
       max < ;
    REP ;
  (Otherwise, politely refuse.)
  ':: word "... " + 'word STO
    length #2 + 'length STO
    write ;
  IFTE ;
{ max :idx: #0 } LOCAL ;
```

ize.dir

The real rabbit out of the hat is the new scheme's ability to represent directories, which are shown in a fashion similar to the COLOR.NAMES routine. The interpreter doesn't have a useful string representation, so all the work is done here.

```
::
  word colors Types.Directory GETE + "[dir: " + 'word STO
  length #6 + 'length STO write
  indent tab +
  ':: ANSI.ize.dirinnards ;
  { indent } LOCAL
  word colors Types.Directory GETE + "]" +
  ANSI.codes.nofore + " " + 'word STO write ;
```

You'll notice that, other than delimiters and temporarily incrementing the indentation, this function does very little work. This is because directory highlighting can recurse, and while a directory object itself is always anonymous, any subdirectory encountered here will have a name.

ize.dirinnards

When it comes to processing the inside of a directory, there's considerably more going on. First, some setup: our quoted directory object is recalled, then stored back to *obj* without a quote. Its list of names is generated, and the length of that list found:

```
::
  obj DUP 'obj STO DIR DUP LEN
```

Those are stored as *names* and *max* respectively, in local variables, along with *idx* and *wasdir*. Before anything is actually done, though, a check is made against the recursion depth, and most of the routine is skipped if *depth* has reached 0. A check is also made against *max*: if there are no entries in the directory, there's nothing to iterate through.

```
    '::
      depth
      ':: max
        '::
```

If we do have stuff to do, though, this code is repeated. The symbol obj is combined with each directory name in turn, then recalled to find its type. Normally, the name will be highlighted in the color associated with the object's type. But if it's a directory, other things happen.

```
          '::
            (Pull the actual object in question out of the dir to get its type.)
            'obj names idx GET + RCL TYPE DUP colors SWAP GETE SWAP
            (Check to see if we ended up with a directory.)
            Types.Directory ==
            (If it is, take a quick detour to explore it.)
```

For a directory, a newline is requested, and then instead of just writing the name, a word is formulated as "[name: " and written:

```
            '::
              newline word SWAP +
              "[" + names idx GET >STR DUP LEN #3 + length + 'length STO
              + ": " + 'word STO write
```

Then a new local environment is formulated to recurse straight back into *dirinnards*, with the recursion depth decremented, the indent incremented, and *obj* bearing the subdirectory we found:

```
              (Get our subdirectory again.)
              'obj names idx GET + EVAL
              (Bump our indent and reduce our recursion depth.)
              indent tab + depth #1 -
              ':: ANSI.ize.dirinnards ; { depth indent obj } LOCAL
```

After *dirinnards* is run, *indent*, *depth*, and *obj* revert to what they were, a closing bracket is added to the current word, and *wasdir* is set to true to improve formatting later on:

```
              word colors Types.Directory GETE + "]"
              length #1 + 'length STO
              (Once we come back from all that, flag it.)
              #1 'wasdir STO ;
```

If a non-directory object needs to be colorized, that's much simpler: the name is added to the word in progress, and that's it.

```
                (Colorize the entry name with that color.)
                ':: word SWAP + names idx GET >STR DUP LEN #2 + length + 'length STO ;
                IFTE
```

No matter what it was, color is removed here, and the index incremented. There are then two different behaviors depending on whether the end of the list has been reached.

```
                + ANSI.codes.nofore + 'word STO
                (And do the loop stuff.)
                idx #1 + DUP 'idx STO
                max < DUP
```

If the list isn't over, a comma and space are added to the current word and written, and if a subdirectory was just processed, a newline is issued for readability's sake (and the wasdir flag reset). When the list is over, a period is added instead, and the word is not written, so that the final name, period, and closing delimiter are nonbroken.

```
                ':: word ", " + 'word STO write
                  (If there's more things to list, and we just did a directory,
                   also force a newline for readability.)
                  wasdir
                  ':: newline #0 'wasdir STO ; IFT ;
                ':: word "." + 'word STO ;
                IFTE ;
              REP ;
            IFT ;
```

If the recursion depth was exceeded, just like with a list, an ellipsis is written out and the function returns, preventing further recursion.

```
          ':: word "... " + 'word STO
             length #3 + 'length STO
             write ;
          IFTE ;
        { max names :idx: #0 :wasdir: #0 } LOCAL ;
```

Directory colorization redux

Now that there's a standard ANSI-izer for directory objects, a fully recursive replacement for *COLOR.NAMES* is trivial. It did need one small hand up from the Python side, a new internal called *firstobj* that returns the entire named store as a directory. This isn't exposed as a builtin because one can get into trouble with access to it, but for purposes of displaying the directory listing it's quite safe:

```
     :: "Current names:" DISP `I*.firstobj ANSI.tocolor ;
```

New tricks: adjusting defaults

Any of the features of the new highlighter can be adjusted by writing new defaults. Setting a wider margin could be as simple as storing a new value:

```
#120 'ANSI.default.margin STO
```

Or it could be anything which evaluates to an expected type:

```
':: margin #40 / ; 'ANSI.default.tab STO
```

New color strings can be stored at will; ANSI-izers can be replaced and new ones added; *newline* can be replaced with any scheme one sees fit. It's a framework which can accomodate a great deal of customization without needing much if any work on the existing code.

New tricks: subclassing

Subclassing may be a bit of poetic license, but because the default environment's own variables are covered by any that are supplied from outside, significant temporary changes can be made. For example, to display an object without any colors at all, while retaining the formatting and modifying it, one could override several variables in different ways. Here a completely blank colors list is supplied on the stack, with a starting indent, wider tabs, and narrower margins:

```
::
  { "" } Types.n LEN *
  ':: newline izer newline ;
  { colors
    :margin: #50
    :indent: #5
    :tab: #4 }
  ANSI.default.environment ;
```

This permits a great deal of code reuse without affecting the normal working environment seen by the user.

ERRTRACE

The error traceback routine included in colorruntime.rpl uses its own loop to generate output text, where the instruction pointer for each level of the call stack is used to highlight the offending object in red. It doesn't obey any of the formatting rules of *COLOROBJ*, and does not recurse into lists or code. This works, but is a wholly separate program. Even though for a traceback the color of each object is dependent upon its position rather than its type, the new scheme is flexible enough to support it. For this application, the default environment is modified on the fly. Going into *ERRTRACE*, a few things are provided by the interepreter:

- A string representing the plaintiff's name
- A string containing the error message
- A list containing a traceback of all the calls and the interpreter's instruction pointer for each, in the form:

```
{ { :: (older call) ; IP }
  ...
  { :: (most recent call) ; IP } }
```

To begin, *ERRTRACE* prints its opening note of displeasure to ensure that even if nothing else works, it's clear the code has been hit. It also grabs the size of the call list:

```
::
  (First, let the user know their sins have caught up with them.)
  "" DISP "You have died of dysentery." DISP "" DISP
  (Hang onto the length of our call stack as 'size'.)
  DUP LEN
```

While all of the ANSI-izers stay the same, instead of having different colors for each object type, instead a small piece of code is duplicated across all object types in the replacement *colors* list. This code will be evaluated against each object, and will select either a red string or a null string dependent upon whether the list index equals the instruction pointer:

```
{ :: idx ip ==
    ANSI.codes.red
    ""
    IFTE ; }
Types.n LEN *
```

Then, a default environment is created with some extras. The details of the traceback are stored as *core*, *size*, *reason*, and *complainant*. Relevant to the highlighting code is a *depth* of 2, to prevent overwhelming the user with screensful of code, and replacing the default *colors* with the list we just generated above. Then, using the same limit used by the stack colorization routine, calls in excess of the default number are skipped:

```
'::
  (If there are too many calls, skip the oldest ones and find our
   starting index.  Otherwise our start index is 0.)
  size stacklimit >
  ':: size stacklimit - DUP " ( +" SWAP + " lines )" + DISP ;
  #0 IFTE
```

It's possible to have an error thrown when no code is in the call stack, and that possibility is dealt with further down.

```
          (Show our calls, if any.)
          core LEN
          '::
            (Now, for each call we're going to show, make an abbreviated
             printable version of each code block.)
```

Meanwhile, for each line of the call stack which is actually going to print, a header is first made showing which line it is:

```
            '::
              ':: (Make a heading.)
                newline
                "In call " size idx - + ": " +
                DUP LEN 'length STO
                ANSI.codes.white SWAP + ANSI.codes.nofore + 'word STO
```

Then, the actual line is fetched, with the code object and instruction pointer split out of the list. The code stays on the stack to be ANSI-ized, and the instruction pointer is stored as a local variable. Because our color selection is looking for an *idx* value from *ize.code*, we cheekily set it to -1 here so it won't trigger until the code is actually being processed by *listinnards.* From here, displaying the object is as straightforward as ever:

```
                (Then grab our line and display it.)
                core idx GET OBJ> DROP
                ':: izer newline ;
                { ip :idx: #-1 } LOCAL
```

And a simple loop cycles through all the calls which need to be displayed.

```
     (Increment our index and see if we're at the end.)
                idx #1 + DUP 'idx STO size < ;
              REP ;
            { idx } LOCAL ;
```

If the user somehow managed to cause an error while no code was being executed, that's covered here too. This can happen if the interpreter's traceback limit is set too high.

```
     ':: DROP "You were not doing anything particular at the time." DISP ;
         IFTE
```

And finally the actual error message is printed.

```
     (And now that we've printed as much traceback as we're going to do, it's
     time to say what actually went wrong.)
    "" DISP
    "The complaint leveled against you by " complainant + " is as follows:" +
    DISP reason DISP ;
  { colors size core reason complainant
    :depth: #2 }
  ANSI.default.environment ;
```

While this application is somewhat toward the edge of what's straightforward to do with the new framework, it works fine, isn't extraordinarily complex, and brings all the nice formatting to an error report where it can be very useful.

STEP

Another useful application for the new scheme is STEP, which handles the display for single stepping through code. While it hasn't been modified yet to work with the environment, it will be straightforward to do so, and it'll benefit from being able to modify the recursion depth and stack limit among other things.