C (source: `xfaces.c` in Emacs)

```c
#if defined HAVE_X_WINDOWS && defined USE_X_TOOLKIT

/* Make menus on frame F appear as specified by the `menu' face.  */
static void
x_update_menu_appearance (struct frame *f)
{
  struct x_display_info *dpyinfo = FRAME_DISPLAY_INFO (f);
  XrmDatabase rdb;

  if (dpyinfo && (rdb = XrmGetDatabase (FRAME_X_DISPLAY (f)), rdb != NULL))
    {
      char line[512];
      char *buf = line;
      ptrdiff_t bufsize = sizeof line;
      Lisp_Object lface = lface_from_face_name (f, Qmenu, true);
      struct face *face = FACE_FROM_ID (f, MENU_FACE_ID);
```

Emacs lisp (source: `esh.el` in this library)

```elisp
(require 'seq)
(require 'color)
(require 'subr-x)

;;; Misc utils

(defun esh--normalize-color (color)
  "Return COLOR as a hex string."
  (upcase (if (= (aref color 0) ?#) color
            (apply #'color-rgb-to-hex (color-name-to-rgb color)))))

(defun esh--filter-cdr (val alist)
  "Remove conses in ALIST whose `cdr' is VAL."
  (seq-filter (lambda (pair) (not (eq (cdr pair) val))) alist))
```

Python (source: `monospacifier.py`)

```python
class AllowWideCharsGlyphScaler(GlyphScaler):
    def __init__(self, cell_width, avg_width):
        """Construct instance based on target CELL_WIDTH and source AVG_WIDTH."""
        GlyphScaler.__init__(self, cell_width)
        self.avg_width = avg_width

    def scale(self, glyph):
        if glyph.width > 0:
            new_width_in_cells = int(math.ceil(0.75 * glyph.width / self.avg_width))
            # if new_width_in_cells > 1:
            #     print("{} is {} cells wide ({} -> {})".format(...))
            GlyphScaler.set_width(glyph, new_width_in_cells * self.cell_width)
```

Perl (source: YAGOpt)

```perl
#&getopt("f:bar") ||
#die &usage("script","f:bar","oo","[files ...]");
sub getopt {
    local($_,$flag,$opt,$f,$r,@temp) = @_;
    @temp = split(/(.):/);
    while ($#temp >= $[) {
        $flag .= shift(@temp);
        $opt .= shift(@temp);
    }
    while ($_ = $ARGV[0], /^-(.)(.*)/ && shift(@ARGV)) {
        ($f,$r) = ($1,$2);
        last if $f eq '-';
        if (index($flag,$f) >= $[) {
            eval "\$opt_$f++;";
            $r =~ /^(.)(.*)/,redo if $r ne '';
```

Ruby (source: `parser.rb` in Ruby's standard library)

```ruby
class NotWellFormedError < Error
  attr_reader :line, :element

  # Create a new NotWellFormedError for an error at +line+ in +element+.
  def initialize(line=nil, element=nil)
    message = "This is not well formed XML"
    if element or line
      message << "\nerror occurred"
      message << " in #{element}" if element
    end
    message << "\n#{yield}" if block_given?
    super(message)
```

Misc

Inline snippets and inline blocks

ESH works inline as well:

- Here's some C and some Python code: ⟨`int main() { return 0; }`⟩, ⟨`def method(self, x): yield x`⟩
- Some Elisp with prettification: ⟨`(λ (x y) (∨ (≤ x y) (≈ (⊕ x y) 0)))`⟩
  without prettification: ⟨`(lambda (x y) (or (<= x y) (approx= (/+/ x y) 0)))`⟩
- And finally a few inline blocks: `def main():\n    return 0` `def main():\n    return 0` `def main():\n    return 0`

Line breaking

ESH allows line breaks to happen within inline code snippets (here is an example: ⟨`private static volatile int counter = 0`⟩), but not in code blocks:

```elisp
(defun esh--normalize-color (color) (upcase (if (= (aref color 0) ?#) color (apply #'color-rgb-to-hex (color-nar
```

Highlighting with non-core Emacs packages

The following examples all depend on externally developped packages, and thus require that you run `cask install` to install these dependencies (Cask is the Emacs Lisp equivalent of Python's virtualenvs).

Haskell (source: `Monoid.hs` in Haskell's standard library)

```haskell
-- | The dual of a 'Monoid', obtained by swapping the arguments of 'mappend'.
newtype Dual a = Dual { getDual :: a }
        deriving (Eq, Ord, Read, Show, Bounded, Generic, Generic1)

instance Monoid a ⇒ Monoid (Dual a) where
        mempty = Dual mempty
        Dual x `mappend` Dual y = Dual (y `mappend` x)

-- | The monoid of endomorphisms under composition.
newtype Endo a = Endo { appEndo :: a → a }
               deriving (Generic)

instance Monoid (Endo a) where
        mempty = Endo id
        Endo f `mappend` Endo g = Endo (f ∘ g)
```

Racket (source: `misc.rkt` in Racket's standard library)

```racket
(define-syntax define-syntax-rule
  (λ (stx)
    (let-values ([(err) (λ (what . xs) (apply raise-syntax-error
                                              'define-syntax-rule what stx xs))])
      (syntax-case stx ()
        [(dr (name . pattern) template)
         (identifier? #'name)
         (syntax/loc stx
           (define-syntax name
             (λ (user-stx)
               (syntax-case** dr #t user-stx () free-identifier=? #f
                 [(_ . pattern) (syntax-protect (syntax/loc user-stx template))]
                 [_ (pattern-failure user-stx 'pattern)])))))])))
```

OCaml (source: `genlex.ml` in OCaml's standard library)

```ocaml
(** The lexer **)
let make_lexer keywords =
  let kwd_table = Hashtbl.create 17 in
  List.iter (λ s → Hashtbl.add kwd_table s (Kwd s)) keywords;
  let ident_or_keyword id =
    try Hashtbl.find kwd_table id with
      Not_found → Ident id
  and keyword_or_error c =
    let s = String.make 1 c in
    try Hashtbl.find kwd_table s with
      Not_found → raise (Stream.Error ("Illegal character " ^ s))
```

Dafny (source: `DutchFlag.dfy` in Dafny's repo)

```
method DutchFlag(a: array<Color>)
  requires a ≠ null modifies a
  ensures ∀ i,j · 0 ≤ i < j < a.Length ⟹ Ordered(a[i], a[j])
  ensures multiset(a[..]) == old(multiset(a[..]))
{
  var r, w, b := 0, 0, a.Length;
  while w ≠ b
    invariant 0 ≤ r ≤ w ≤ b ≤ a.Length;
    invariant ∀ i · 0 ≤ i < r ⟹ a[i] == Red
    invariant multiset(a[..]) == old(multiset(a[..]))
  {   match a[w]
        case Red ⇒
          a[r], a[w] := a[w], a[r];
          r, w := r + 1, w + 1;
```

F* (source: `Handshake.fst` in miTLS)

```
val processServerFinished: KeySchedule.ks → HandshakeLog.log → (hs_msg × bytes) → ST (result bytes)
  (requires (λ h → T))
  (ensures (λ h₀ i h₁ → T))

let processServerFinished ks log (m, l) =
   match m with
   | Finished (f) →
     let svd = KeySchedule.ks_client_12_server_finished ks in
     if (equalBytes svd f.fin_vd) then
        let _ = log @@ (Finished (f)) in
    Correct svd
     else Error (AD_decode_error, "Finished MAC did not verify")
   | _ → Error (AD_decode_error, "Unexpected state")
```

Coq (source: `ExtendedLemmas.v` in Fiat; requires a local Proof General setup)

```
Lemma ProgOk_Chomp_lemma :
  ∀ `{FacadeWrapper (Value av) A} (ev: Env av) (key: StringMap.key)
    (prog: Stmt) (tail₁ tail₂: A → Telescope av) ex (v: A),
    key ∉ ex →
    ({{ tail₁ v }} prog {{ tail₂ v }} ∪ {{ [key ▷ wrap v] :: ex }} // ev ↔
     {{ [[`key ↦ v as vv]]::tail₁ vv }} prog {{ [[`key ↦ v as vv]]::tail₂ vv }} ∪ {{ ex }} // ev).
Proof.
  repeat match goal with
         | _ ⇒ tauto
         | _ ⇒ progress (intros ‖ split)
         | [ H: ?a ∧ ?b ⊢ _ ] ⇒ destruct H
         | [ H: ?a ≲ Cons _ _ _ ∪ _ ⊢ _ ] ⇒ learn (Cons_PushExt _ _ _ _ _ H)
         | [ H: ProgOk ?fmap _ _ ?t₁ ?t₂, H': _ ≲ ?t₁ ∪ ?fmap ⊢ _ ] ⇒ destruct (H _ H'); no_dup
         | [ H: RunsTo _ _ ?from ?to, H': ∀ st, RunsTo _ _ ?from st → _ ⊢ _ ] ⇒ specialize (H' _ H)
         | [ H: _ ≲ _ ∪ [_ ▷ _] :: _ ⊢ _ ] ⇒ apply Cons_PopExt in H
         end.
Qed.
```