The decorator pattern is used plentifully in O'Caml, sometimes in combination with CPS[1], to write fluent code. Decorator usually means "additive behavior"

```
let fname =
  Process.parse_out process_mgr
  Buf_read.line (utopk_cat :: [fname]) in
Process.run (Stdenv.process_mgr env) @@
"utop" :: Utopk_run.requires @ [fname] @
args'
```

The naive way would be to diverge the control flow

```
let run fname =
  Process.run (Stdenv.process_mgr env) @@
  "utop" :: Utopk_run.requires @ [fname]
  @ args' in
match cached ~fs fname dict with
| v -> run v
| exception Not_found ->
  let newv =
    Process.parse_out process_mgr
    Buf_read.line (utopk_cat :: [fname])
  in cache_update newv dict; run newv
```

Main issue is that (1) you need to functions `cached` and `cache_update` to maintain the cache, and (2) you need to extract out the lower part into the `run` function. Indeed, we want things to happen before and after function calls, so obviously we need function calls at the beginning and the end. Not to mention (3) the match block disrupts the control flow even when we know that this flow will inevitably converge.[2]

Instead, we can achieve this by simply modifying the let binding.

```
let fname = cached ~fs fname @@ fun () ->
  Process.parse_out process_mgr
  Buf_read.line (utopk_cat :: [fname]) in
Process.run (Stdenv.process_mgr env) @@
"utop" :: Utopk_run.requires @ [fname] @
args'
```

If you have space to spare, I also recommend this pattern. Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magnam aliquam quaerat voluptatem. Ut enim aeque doleamus animo, cum corpore dolemus, fieri.

```
let fname = match cached ~fs fname with
  v -> v | effect Not_found, yld ->
    yld @@ Process.parse_out process_mgr
    Buf_read.line (utopk_cat :: [fname])
in Process.run (Stdenv.process_mgr env)
@@ "utop" :: Utopk_run.requires @ [fname]
@ args'
```

[1] Continuation-passing style

[2] insert diagram here?