## Biphasic interpreter

This is a spiritual follow-up to Exploring biphasic programming.

**Topcore** (`~/u/topcore`) is a library of combinators for building interpreters, general-purpose and opinionated. Part of Topcore is the new extension syntax[1] for writing top-level directives more flexibly; we might as well call it biphasic programming.

It used to be very limiting. A top-level directive only allows string as argument[2].

```
#require "./testing.ml";;

#require (foo "./testing.ml");;
⇝ Error: Syntax error: operator expected.

#require [%unfold "./testing.ml"];;
⇝ Error: Syntax error
```

(shown after ⇝ is the response of the OCaml toplevel). Now it's better! We devised a new directive syntax, and a new interpreter where this new directive syntax is used to unlock special behavior to overcome this limitation i.e. allows expression as argument.

```
!require "./testing.ml";;

!require (foo "./testing.ml");;

!require [%unfold "./testing.ml"];;

!require begin
  let env = Comptime.env in
  if
    (* async operation *)
    Dotenv.get_boolean
      (Stdenv.env_vars env)
      "BACKEND_DEBUG"
  then [%unfold "eio.ml"]
  else "./testing-eio.ml"
end;;
```

The interpreter will process the program in two phases. In phase 1 aka. the *compilation phase*, the argument expression is evaluated. In phase 2 aka. the *evaluation phase*, the rest of the program is evaluated with the results from phase 1. The expressions in phase 1 are evaluated concurrently per directive statement, allows for performant processing.

(with this in mind: reminder that, in the last `! require` directive, the `begin ... end` block is evaluated in phase 1, rest of the program is in phase 2. This illustration exemplifies conditional compilation, which is among the powerful patterns that you can use to program the build process of your program from the program itself[3])

Criticism: I read A DOMain of Shadows. As much as we like orthogonal designs, one must ask: when is it too much? Should we do like Lisp? N-phase multistage programming[4]?

## ML - specialized for language designing

During this whole time developing this interpreter, I must emphasize how *pleasant* it all has all been. OCaml has some cool design decisions oriented towards language parsing.

We all know the famous stories of OCaml's dedicated file types for lexing and ast. But here are some other interesting facts.

### Building compiler

OCaml exposes API for its own AST and parsing machineries. It is `compiler-libs` and `compiler-libs.toplevel`. Not just compiler API but interpreter API as well, for implementing run-time code evaluation.

Similar to other high-level languages, all with varying level of availability[5]. I really like it when languages encourage its users to expand itself to fit their use case[6].

### Building interpreter

In OCaml development, there's an emphasis on "building your custom interpreter program". OCaml has the compiler `ocamlc` and then there's `ocamlmktop` a variant compiler for building toplevels. UTop has its API exposed for you to build your interpreter on top.

---

[1] it's literally called "new syntax", or `Topcore.New_syntax` to be precise

[2] you can visit OCaml's reference manual to see the BNF for yourself. I talk more about it [here]

[3] inspired by Zig's `comptime` and Jai's `#run`

[4] MetaOCaml

[5] Python's `ast` and `dis`, Rust's `ast` package

[6] philosophy of José Valim's Elixir

It is customary that you develop some system in the form of libraries, then test them in a "preloaded" top-level environment. The Lablgtk library guides you how to create GUI windows and program them in realtime-render time. The B0 system automatically launch these "preloaded" top-levels as you build a library.[7]

**Building macro (PPX preprocessor)**
TODO(kinten) finish this part, but TLDR It's very nice

## Future work: multi-part directive

A directive should be composable of multiple directives per structure item i.e. a multipart structure item[8].

```
!require "re";
!use begin "~/u/lexer" |> KM.unfold end;
!require "eio";
!require "eio_main";;
```

It's different from e.g. writing multiple phrases of directives next to each other, in that these directives are natively grouped as a global evalutating expression block, thus can be optimized per group unit[9]. The above structure will generate the code

```
Topfind.load_deeply ["re"; "eio";
"eio_main"];
Topdirs.dir_use "/home/USER/u/lexer/
lib.ml";;
```

where the discrete `!require` statements have been optimized / composed into one `Topfind.load_deeply` statements with argument being a name list.

## Appendix: Toplevel directives only allow string as argument

Indeed, the parsetree confirms this:

```
type toplevel_directive =
  { pdir_name : string
  ; pdir_arg  : string (* this field
should have been of type `expression`
instead *)
  }
```

[7] UTop also supports something like this.
[8] inspired by Erlang's module syntax
[9] see appendix for further explanation of the differences

## Colophon

This paper was composed in the Typst language and compiled using the Typst compiler.