```
;;;; THE DECLARATIVE COMPILER DRIVER ;;;;;
;;                                      ;;
;;  An Exercise in Function Composition ;;
;;                                      ;;
;;   Norman Ramsey, Tufts University    ;;


; class: "Simple Virtual Machines & Language Translation"

; Subtext: next step in functional programming

; *Implementations* of `lambda`, `case`

; No register allocation or instruction selection (target a VM)


r1 := "Hello world"   ;; "virtual assembly language"
println r1

; uft vs-vo 00.vs | svm
; uft vs-vo 00.vs
```

```
;; K-normal form

(let ([$r0 'Hello-world])
  (println $r0))

;; uft kn-vo 01.scm | svm
;; vscheme 01.scm

;; echo "(println 'Hello-world)" | uft fo-kn

;; echo "(println (+ 1 (+ 1 (+ 2 (+ 3 5)))))" | uft fo-kn
```

```
$ uft
Usage:
  uft <from>-<to> [file]
where <from> and <to> are one of these languages:
  es   eScheme (ho vScheme plus Erlang-style pattern matching)
  hox  Higher-order vScheme with mutable variables in closures
  ho   Higher-order vScheme
  fo   First-order vScheme
  cl   First-order vScheme with closure and capture forms
  kn   K-Normal form
  vs   VM assembly language
  vo   VM object code


;;        es    fo
;;          \     \
;;           \     \
;;      ho -- cl -- kn -- vs -- vo
;;      /
;;     /
;;   hox
```

```
; Differential debugging
; (here, of closure conversion)

(define >>> (f g)  ;; composition, Elm style
  (lambda (x)
    (g (f x))))

(define double (n)
  (+ n n))

(check-expect ((>>> double double) 1) 4)

; uft ho-cl 03.scm
; uft ho-vo 03.scm

; uft ho-cl 03.scm | vscheme  # run with interpreter
; uft ho-vo 03.scm | svm      # run with VM
```

```
signature UFT = sig
  datatype language = ES | HOX | HO | FO | CL | KN | VS | VO
  val translate : language * language
                -> TextIO.instream * TextIO.outstream
                -> unit Error.error
end

(*
  Input language 𝒥 READER:

          val 𝒥_of_file : instream -> 𝒥 error

  Output language 𝒦 WRITER:

          val emit_𝒦 : outstream -> 𝒦 -> unit

  Secret sauce: output language ℒ MATERIALIZER,
                takes NAME of 𝒥 as parameter:

          val ℒ_of : language -> instream -> ℒ error
*)
```

```
fun f >>> g = fn x => g (f x)
    (* again, function composition, Elm style *)

val ! = Error.fmap  (* abbreviate *)

fun translate (inLang, outLang) (infile, outfile) =
  (case outLang
     of VO => VO_of      inLang >>> ! (emitVO outfile)
      | VS => VS_of      inLang >>> ! (emitVS outfile)
      | KN => KN_text_of inLang >>> ! (emitKN (embedding inLang) outfile)
      | FO => FO_of      inLang >>> ! (emitFO outfile)
      | CL => CL_of      inLang >>> ! (emitCL outfile)
      | HO => HO_of      inLang >>> ! (emitHO outfile)
      | HOX => HOX_of    inLang >>> ! (emitHO outfile)
      | ES => ES_of      inLang >>> ! (emitHO outfile)
  ) infile
  handle Backward => raise NotForward (inLang, outLang)
       | NoTranslationTo outLang => raise NotForward (inLang, outLang)
```

```
  fun VO_of VO     = VO_of_file
    | VO_of inLang = VS_of inLang >=> Assembler.translate

  fun VS_of VS     = VS_of_file
    | VS_of inLang = KN_reg_of inLang >=> VS_of_KN

  val VS_of_KN : ObjectCode.reg KNormalForm.exp list
              -> AssemblyCode.instr list error
   = map Codegen.forEffect >>> List.concat >>> Error.OK

(*  ... and so on up the chain ... *)
```

Conclusion

Happiness

- **Stop anywhere without `-c`, `-E`, `-S` nonsense**
- **Start anywhere**
- **Driver module is all function composition**

Complications

- **Multiple pathways**
- **Conditional/options on individual passes**

```
val schemeOfFile : instream -> VScheme.def list error =
  lines                                  (* line list *)
  >>>  SxParse.parse                     (* sx list error *)
  >==>  Error.mapList VSchemeParsers.defs  (* def list list error *)
  >>>  Error.map List.concat             (* def list error *)
  >>>  Error.map VSchemeTests.delay

val schemexOfFile : instream -> UnambiguousVScheme.def list error =
  schemeOfFile >>>
  Error.map (map Disambiguate.disambiguate)



val FO_of_file : instream -> FirstOrderScheme.def list error =
  schemexOfFile >==> Error.mapList FOUtil.project

val KN_of_file  : instream -> string KNormalForm.exp list error =
  schemexOfFile  (* def list *)
  >==>
  Error.mapList KNProject.def


fun HO_of HOX  = schemexOfFile >>> ! (map Mutability.moveToHeap)
  | HO_of HO   = schemexOfFile >==> Error.mapList Mutability.detect
  | HO_of _    = raise Backward



fun KN_reg_of KN     =
      KN_of_file >==> Error.mapList (KNRename.mapx KNRename.regOfName)
  | KN_reg_of inLang =
      CL_of inLang >>> ! (map KNormalize.def)

fun KN_text_of KN     =
```

```
      KN_of_file
  | KN_text_of inLang =
      KN_reg_of inLang >==>
      Error.mapList (KNRename.mapx (Error.OK o KNormalize.regname))



fun CL_of CL     = CL_of FO   (* really *)
  | CL_of HO     = HO_of HO     >>> ! (map ClosureConvert.close)
  | CL_of HOX    = HO_of HOX    >>> ! (map ClosureConvert.close)
  | CL_of ES     = ES_of ES     >>> ! (map ClosureConvert.close)
  | CL_of inLang = FO_of inLang >>> ! (map FOCLUtil.embed)
```