1. Hay una condición de carrera en su verificación de existencia del archivo de entrada. El archivo puede ser eliminado entre el momento en que verifican si el archivo existe con `System.Directory.doesFileExist`, y el momento en que intentan abrirlo con `System.IO.openFile`.

2. La solución a este problema es simplemente no hacer la verificación, y dejar que la llamada para abrir el archivo reporte el error si éste no existiera. En particular, `System.IO.openFile` lanza una excepción de `IO` que puede ser atrapada si les parece inadecuado el comportamiento por defecto, que es mostrar un mensaje de error y abortar la ejecución — que es exactamente lo que terminan haciendo.

3. Para casos así, es preferible dejar que la excepción haga lo suyo. Por lo tanto, es razonable simplificar `main` así:

```
main = do
    (filename : _) <- SE.getArgs
    openF filename
```

4. que es casi lo mismo, salvo por el mensaje de error (que es un poco más claro con el patrón), que

```
main = do
    args <- SE.getArgs
    openF (head filename)
```

5. que es equivalente, por definición de `.` (el operador de composición de funciones), a

```
main = do
    args <- SE.getArgs
    (openF . head) filename
```

6. que es sintácticamente equivalente, por definición de la notación `do`, a

```
main = openF . head =<< SE.getArgs
```

7. y cinco palabras suele ser mejor que cinco líneas.

---

1. Noten que

```
openF filename = do
    handle   <- IO.openFile filename ReadMode
    processFile handle
```

2. es sintácticamente equivalente, por la regla de operadores infijos, a

```
openF filename = do
    handle <- filename `IO.openFile` ReadMode
    processFile handle
```

3. que es sintácticamente equivalente, por la regla de secciones de operadores, a

```
openF filename = do
    handle <- (`IO.openFile` ReadMode) filename
    processFile handle
```

4. que es sintácticamente equivalente, por la definición de la notación `do`, a

```
openF filename = processFile =<< (`IO.openFile` ReadMode) filename
```

5. que es equivalente, por la definición de `<=<` (uno de los operadores de composición de Kleisli), a

```
openF filename = (processFile <=< (`IO.openFile` ReadMode)) filename
```

6. que es equivalente, por eta-reducción, a

```
openF = processFile <=< (`IO.openFile` ReadMode)
```

7. y cuatro palabras es mejor que dos líneas.

8. Noten que

```
printTokens []    = putStrLn ""
printTokens (a:b) = do
    putStrLn $ show a
    printTokens b
```

9. es equivalente, por la definición de `print`, a

```
printTokens []    = putStrLn ""
printTokens (a:b) = do
    print a
    printTokens b
```

10. Además, el `putStrLn ""` final podría simplemente eliminarse, ya que no es requerido que se imprima una línea en blanco adicional — el enunciado de la primera etapa, en la §3.9, especifica únicamente que se imprima una línea por cada lexema o cada error lexicográfico. Simplificando, queda

```
printTokens []    = return ()
printTokens (a:b) = do
    print a
    printTokens b
```

11. ya que `return ()` es un cómputo sin efectos y del mismo tipo que `putStrLn ""`. Eso es equivalente, por la definición de `mapM_`, a simplemente

```
printTokens = mapM_ print
```

12. y dos palabras ciertamente es mejor que cuatro líneas con recursión explícita.

13. Noten que

```
processFile handle = do
    line  <- hGetContents handle
    let (a,b) = getTokens line
    putStrLn $ "Error Lexicos encontrados: \n" ++ show a
    putStrLn $ "Tokens encontrados:"
    printTokens b
```

14. puede simplificarse ligeramente a

```
processFile handle = do
    (a, b) ← getTokens =<< hGetContents handle
    putStrLn $ "Error Lexicos encontrados: \n" ++ show a
    putStrLn $ "Tokens encontrados:"
    printTokens b
```

15. y, de paso, el nombre `line` es poco claro, ya que lo que se lee representa la entrada completa, no una sola línea. Por otra parte, ya que `printTokens` es tan sencillo, vale la pena hacer *inlining* de su definición:

```
processFile handle = do
    (a, b) ← getTokens =<< hGetContents handle
    putStrLn $ "Error Lexicos encontrados: \n" ++ show a
    putStrLn $ "Tokens encontrados:"
    mapM_ print b
```

16. y además las impresiones de texto son innecesarias ya que el enunciado no las requiere, así que puede simplificarse a

```
processFile handle = do
    (a, b) ← getTokens =<< hGetContents handle
    print a
    mapM_ print b
```

17. que es equivalente, por la definición de `Control.Monad.***`, a

```
processFile handle = do
    (a, b) ← (print *** mapM_ print) . getTokens =<< hGetContents handle
    a
    b
```

18. que es equivalente, por la definición de la notación `do`, a

```
processFile handle = do
    (a, b) ← (print *** mapM_ print) . getTokens =<< hGetContents handle
    a >> b
```

19. que es sintácticamente equivalente, por la regla de operadores infijos, a

```
processFile handle = do
    (a, b) ← (print *** mapM_ print) . getTokens =<< hGetContents handle
    (>>) a b
```

20. que es equivalente, por la definición de `uncurry`, a

```
processFile handle = do
    (a, b) ← (print *** mapM_ print) . getTokens =<< hGetContents handle
    (uncurry (>>)) (a, b)
```

21. que es sintácticamente equivalente, por la definición de la notación `do`, a

```
processFile handle = uncurry (>>) . (print *** mapM_ print) . getTokens =<< hGetContents handle
```

22. que es equivalente, por la definición de `<=<`, a

```
processFile handle = (uncurry (>>) . (print *** mapM_ print) . getTokens <=< hGetContents) handle
```

23. que es equivalente, por eta-reducción, a

```
processFile = uncurry (>>) . (print *** mapM_ print) . getTokens <=< hGetContents
```

24. que no es demasiado claro, pero puede formatearse como

```
processFile
  = uncurry (>>)
  . (print *** mapM_ print)
  . getTokens
  <=< hGetContents
```

25. y ahora casi todas las líneas tienen apenas uno o dos símbolos, y no hay nombres como `line` que puedan sugerir un significado que no corresponde con su verdadero uso. Este estilo puede resultarles poco legible, pero tiene ventajas.

26. Noten que el único uso de `processFile` ocurría en `openF`:

    “`{.haskell} openF = processFile <=< (IO.openFile‘ ReadMode)`

27. processFile = uncurry (>>) . (print *** mapM_ print) . getTokens <=< hGetContents “‘

28. Sustituyendo (porque las cosas iguales pueden sustituirse unas por otras, claro), se obtiene

```
openF
  = (
    uncurry (>>)
  . (print *** mapM_ print)
  . getTokens
  <=< hGetContents
  )
  <=< (`IO.openFile` ReadMode)
```

29. que es equivalente, por la ley de asociatividad de `<=<` que cumple todo *monad*, a

```
openF
  = uncurry (>>)
  . (print *** mapM_ print)
  . getTokens
  <=< (
    hGetContents
    <=< (`IO.openFile` ReadMode)
  )
```

30. que es equivalente, por la definición de `<=<`, a

```
openF
  = uncurry (>>)
  . (print *** mapM_ print)
  . getTokens
  <=< (\ filename →
    (`IO.openFile` ReadMode) filename
    >>= hGetContents
  )
```

31. que es sintácticamente equivalente, por la definición de la notación `do`, a

```
openF
  = uncurry (>>)
  . (print *** mapM_ print)
  . getTokens
  <=< (\ filename → do
    handle ← (`IO.openFile` ReadMode) filename
    hGetContents handle
  )
```

32. que es sintácticamente equivalente, por la regla de secciones de operadores, a

```
openF
  = uncurry (>>)
  . (print *** mapM_ print)
  . getTokens
  <=< (\ filename → do
    handle ← filename `IO.openFile` ReadMode
    hGetContents handle
  )
```

33. que es sintácticamente equivalente, por la regla de operadores infijos, a

```
openF
  = uncurry (>>)
  . (print *** mapM_ print)
  . getTokens
  <=< (\ filename → do
    handle ← IO.openFile filename ReadMode
    hGetContents handle
  )
```

34. que es equivalente, por la definición de `System.IO.readFile`, a

```
openF
  = uncurry (>>)
  . (print *** mapM_ print)
  . getTokens
  <=< readFile
```

35. y ahora `openF`, `processFile` y `printTokens` se reducen a tres líneas muy cortas. Además,

```
main = openF . head =<< SE.getArgs
```

36. así que, sustituyendo por igualdad, se obtiene

```
main
  = (
      uncurry (>>)
    . (print *** mapM_ print)
    . getTokens
    <=< readFile
  ) . head =<< SE.getArgs
```

37. que es equivalente, por la definición de `<=<`, a

```
main
  = (\ filename →
      readFile filename >>=
      uncurry (>>) . (print *** mapM_ print) . getTokens
  ) . head =<< SE.getArgs
```

38. que es equivalente, por la definición de `=<<`, a

```
main
  = (\ filename →
      uncurry (>>) . (print *** mapM_ print) . getTokens
      =<< readFile filename
  ) . head =<< SE.getArgs
```

39. que es sintácticamente equivalente, por la regla de operadores infijos, a

```
main
  = (\ filename →
      (=<<)
        (uncurry (>>) . (print *** mapM_ print) . getTokens)
        (readFile filename)
  ) . head =<< SE.getArgs
```

40. que es equivalente, por la definición de composición de funciones, a

```
main
  = (\ filename →
      ( (=<<) (uncurry (>>) . (print *** mapM_ print) . getTokens)
      . readFile
      ) filename
  ) . head =<< SE.getArgs
```

41. que es equivalente, por eta-reducción, a

```
main
  = ( (=<<) (uncurry (>>) . (print *** mapM_ print) . getTokens)
    . readFile
    )
  . head =<< SE.getArgs
```

42. que es equivalente, por la ley de asociatividad de `.`, a

```
main
  = ( (=<<) (uncurry (>>) . (print *** mapM_ print) . getTokens)
    . (readFile . head)
    )
  =<< SE.getArgs
```

43. que es equivalente, por eta-expansión, a

```
main
  = (\ x →
      ( (=<<) (uncurry (>>) . (print *** mapM_ print) . getTokens)
      . (readFile . head)
      )
      x
    )
  =<< SE.getArgs
```

44. que es equivalente, por definición de `.`, a

```
main
  = (\ x →
      (=<<)
        (uncurry (>>) . (print *** mapM_ print) . getTokens)
        ((readFile . head) x)
    )
  =<< SE.getArgs
```

45. que es sintácticamente equivalente, por la regla de operadores infijos, a

```
main
  = (\ x →
      (uncurry (>>) . (print *** mapM_ print) . getTokens)
      =<< (readFile . head) x
    )
  =<< SE.getArgs
```

46. que es equivalente, por la regla de asociatividad de `=<<` (la tercera ley de los *monads*), a

```
main
  = (uncurry (>>) . (print *** mapM_ print) . getTokens)
  =<< readFile . head
  =<< SE.getArgs
```

47. y así, las 24 líneas de código del archivo original se reducen a cuatro, donde una sola es compleja.

48. La complejidad de la línea larga se debe principalmente a la forma en que diseñaron la interfaz de `getTokens`. Hubiera sido bastante más sencillo el programa principal —incluso trivial— si hubieran hecho que `getTokens` se encargara de imprimir por sí solo cada lexema o error apenas lo encontraran.

49. En este punto, luego de aplicar todas esas simplificaciones, es razonable volver a una notación más imperativa:

```
main = do
  (errors, tokens) ← getTokens =<< readFile . head =<< SE.getArgs
  print errors
  mapM_ print tokens
```

50. Incluso pueden recuperar el comportamiento original al no obtener el argumento de línea de comando:

```
main = do
  (filename : _) ← SE.getArgs
  (errors, tokens) ← getTokens =<< readFile filename
  print errors
  mapM_ print tokens
```

51. y ciertamente una función de cinco líneas es mejor que cuatro funciones en 24 líneas.

52. Noten que todo este trabajo de simplificación (y mucho más!) es realizado automáticamente por el compilador de *Haskell* como parte de su fase de optimización — salvo el de eliminar impresiones innecesarias, claro.

53. La posibilidad de hacer razonamiento ecuacional para analizar, comprender y simplificar código de esta manera es la fortaleza más grande de *Haskell*, y cuando la apliquen, con frecuencia verán que casi todo su código desaparece. Con el tiempo se acostumbrarán a ver estas transformaciones como algo natural.

---

1. La §3.7 del enunciado de la primera etapa requiere que el programa termine con un estado de salida distinto de cero si hubo errores, e igual a cero si no los hubo. Probablemente querrán usar algo como

```
if null errors
  then exitSuccess
  else exitFailure
```

2. aprovechando los cómputos definidos en el módulo `System.Exit`, y `Data.List.null`.

---

1. Usan el mismo constructor `TkBool` para representar los lexemas `true` y `false`, pero el constructor no recibe parámetros, por lo cual se pierde la información de cuál de los dos literales booleanos se reconoció.