kaikai

Programación simple con kaikai

Un lenguaje para humanos y agentes

Eduardo Díaz

Edición en español · Borrador

Copyright © 2026 · Eduardo Díaz
Borrador del libro *kaikai*. Distribución limitada.
El lenguaje *kaikai* y su documentación viven en github.com/kaikailang-org/kaikai.

Este libro se compone en dos ediciones: español e inglés. Ambas son ciudadanas de primera clase; ninguna es traducción de la otra. Esta es la edición en español.

Tipografía: New Computer Modern (cuerpo), Helvetica Neue (títulos), JetBrains Mono (código).
Compilado con Typst.

Índice

Prólogo 1
- Una nota sobre cómo se escribió este libro 3
- Convenciones 3
- Quién debería leer este libro 5
- Gracias 5

Capítulo 1 · Tour de kaikai 6
- 1.1 Hola, kaikai 6
- 1.2 Tipos algebraicos y `match`: FizzBuzz 7
- 1.3 Una calculadora con AST recursivo 8
- 1.4 Un efecto propio con su handler 9
- 1.5 Dos fibras cooperativas 10
- 1.6 Tipos a la medida con protocolos 11
- 1.7 Unidades de medida 12
- 1.8 Programación por contrato 13
- 1.9 Pruebas en el mismo archivo 14
- 1.10 Holes: agujeros que compilan 14
- 1.11 Dependencias y proyectos: `kai.toml` 15
- 1.12 Cómo instalar y correr `kai` 16
- 1.13 Cómo está organizado el resto del libro 17

Capítulo 2 · Pensar en kaikai 18
- 2.1 Expresiones, no sentencias 18
- 2.2 Inmutabilidad por defecto 20
- 2.3 `Option` y `Result` en vez de `null` y excepciones 21
- 2.4 Pattern matching como herramienta de control de flujo 22
- 2.5 Funciones puras y efectos visibles 23
- 2.6 Una breve genealogía 24
- Ejercicios 25

Capítulo 3 · Tipos básicos y expresiones 26
- 3.1 Los siete tipos primitivos 26
- 3.2 Literales e interpolación de strings 27
- 3.3 Operadores aritméticos, lógicos y de comparación 28
- 3.4 `let` y la propagación local de tipos 29
- 3.5 `if` como expresión 29
- 3.6 Bloques y el valor de un bloque 30
- 3.7 Tres formas de cuerpo de función 31
- Ejercicios 32

Capítulo 4 · Tipos compuestos 33
- 4.1 Records 33
- 4.2 Acceso a campos y destructuring 35
- 4.3 Listas 36
- 4.4 Strings, no listas de chars 37
- 4.5 `Option` y `Result`: el día a día 38
- 4.6 Tuplas 39

Ejercicios 40
Capítulo 5 · Sum types, uniones y `match` 41
5.1 Tipos suma con `|` 41
5.2 Constructores con y sin payload 42
5.3 Recursión en tipos 43
5.4 `match`: patrones, guardas, exhaustividad 44
5.5 Uniones de tipos existentes 45
5.6 Errores como uniones, sin wrappers 47
5.7 Caso de estudio: evaluador con errores tipados 49
Ejercicios 51
Capítulo 6 · Funciones y pipelines 53
6.1 Declaración 53
6.2 Lambdas 55
6.3 Funciones de orden superior 55
6.4 Pipes: `|>`, `|`, `||` 56
6.5 Trailing lambdas y otros azúcares 58
6.6 Recursión y TCO obligatoria 60
6.7 Caso de estudio: pipeline de transformación 61
Ejercicios 62
Capítulo 7 · Pruebas, propiedades y benchmarks 64
7.1 `test "..." { ... }` y `assert` 64
7.2 `kai test` y el ciclo corto de retroalimentación 65
7.3 `check "..."`: propiedades 66
7.4 `bench "..." { ... }`: medir, no adivinar 67
7.5 Cuándo usar cuál 68
7.6 Caso de estudio: pruebas para un mini-evaluador 69
Ejercicios 71
Capítulo 8 · Módulos, imports, organización del código 73
8.1 Un archivo, un módulo 73
8.2 `import` y nombres calificados 74
8.3 Visibilidad: el contrato del módulo 75
8.4 El stdlib que ya tienes 76
8.5 Proyectos: `kai.toml` 76
8.6 Dependencias: git, path, lock 77
8.7 Selección de versiones: MVS 79
8.8 Cache y `kai install` 79
8.9 Caso de estudio: refactor de un proyecto monolítico 79
8.10 Filosofía: simple y previsible 81
Ejercicios 81
Capítulo 9 · Protocolos 83
9.1 Por qué hay protocolos 83
9.2 Declarar un `protocol` y `impl` 84
9.3 Los cinco protocolos del stdlib 84
9.4 `#[derive(...)]` y cuándo usarlo 85
9.5 Protocolos propios 86
9.6 Por qué no hay typeclasses al estilo Haskell 87
9.7 Operadores: `+`, `==`, `<` como protocolos 88

Ejercicios ... 89
Capítulo 10 · Unidades de medida y branded types ... 90
10.1 `unit` y literales anotados ... 90
10.2 Aritmética con unidades ... 91
10.3 Álgebra de unidades: producto, cociente, potencia ... 91
10.4 Unidades genéricas ... 92
10.5 Conversiones explícitas ... 93
10.6 Branded types ... 93
10.7 Caso de estudio: cartera multi-moneda ... 94
Ejercicios ... 96
Capítulo 11 · Programación por contrato y refinement types ... 97
11.1 Por qué contratos y refinements van juntos ... 97
11.2 `requires` y `ensures` en una firma ... 98
11.3 `result` y los nombres en alcance dentro del `ensures` ... 99
11.4 Refinement types ... 99
11.5 Cuándo se prueba estáticamente, cuándo en runtime ... 100
11.6 Tres formas de garantía ... 101
11.7 La familia Design by Contract ... 101
11.8 Lo que kaikai no hace, y por qué ... 103
11.9 Caso de estudio: cuenta bancaria ... 103
Ejercicios ... 104
Capítulo 12 · Efectos algebraicos ... 106
12.1 La fricción que los efectos resuelven ... 106
12.2 Declarar un `effect` ... 108
12.3 Llamar a una operación: la firma cambia ... 108
12.4 Manejar un efecto con `handle ... with` ... 109
12.5 `resume`: el handler decide qué pasa después ... 110
12.6 Handlers con estado: el patrón `State` ... 112
12.7 `var`, `Ref[T]` y `Array[T]`: dos mecanismos distintos ... 113
12.8 Componer efectos: handlers anidados ... 115
12.9 Alias de filas de efectos ... 116
12.10 Default handlers: el efecto trae el suyo ... 116
12.11 Los handlers del stdlib son código kaikai ... 120
12.12 Caso de estudio: procesador de configuración ... 121
12.13 Filosofía: tres ideas que vale recordar ... 122
Ejercicios ... 123
Capítulo 13 · Concurrencia y memoria ... 125
13.1 El modelo: fibras aisladas ... 125
13.2 Perceus en una página ... 126
13.3 Crear y esperar fibras: las operaciones básicas ... 127
13.4 Nurseries: concurrencia estructurada ... 128
13.5 Cancelación cooperativa ... 131
13.6 Memoria mutable por fibra ... 132
13.7 Por qué las fibras no pueden escapar de su nursery ... 132
13.8 Caso de estudio: cola de tareas con pool de trabajadores ... 133
13.9 Filosofía: dos invariantes que vale recordar ... 135
Ejercicios ... 135

Capítulo 14 · Actores 137
Una fibra es un cómputo; un actor es un proceso vivo 137
Los actores no son primitivos del lenguaje 138
14.1 `Actor[Msg]`: el efecto 139
14.2 `with_mailbox`: dar mailbox a la fibra actual 139
14.3 `spawn_actor`: crear un actor nuevo 140
14.4 Policies de mailbox: qué pasa cuando se llena 141
14.5 Patrón request/reply 142
14.6 Supervisión: links y monitores 143
14.7 Caso de estudio: supervisor con reintentos 144
14.8 Filosofía: actores son una biblioteca 146
Ejercicios 147
Capítulo 15 · Holes y kaikai con agentes IA 148
15.1 Holes tipados: `?` y `?nombre` 148
15.2 La conversación con el compilador 149
15.3 Diseño top-down: empieza por la firma 150
15.4 Programas parciales: avanzar con el resto compilando 151
15.5 Holes en patrones: el match incompleto 151
15.6 La apuesta LLM: lenguaje diseñado para agentes 152
15.7 La salida JSON de los holes 152
15.8 Más allá de holes: información rica como interfaz 153
15.9 Un loop de trabajo con un agente 153
15.10 Lo que el lenguaje no automatiza 154
15.11 Caso de estudio: completar una función no trivial 154
15.12 Filosofía: tres ideas que vale recordar 155
Ejercicios 155
Capítulo 16 · Tooling: el binario `kai` 157
16.1 Compilar y correr: `kai run`, `kai build` 157
16.2 Tests, propiedades y benchmarks 158
16.3 Formateo: `kai fmt` 158
16.4 Gestión de paquetes: `init`, `add`, `install`, `update` 159
16.5 Modo de desarrollo: `kai watch` 159
16.6 Integración con editores: `kai lsp` 160
16.7 Variables de entorno 160
16.8 Estructura típica de un proyecto 160
16.9 Hablar con C: `extern "C"` y el efecto `Ffi` 161
16.10 Ediciones: estabilidad sin estancamiento 164
16.11 Filosofía: tres principios del tooling 166
Capítulo 17 · Caso de estudio: servidor HTTP 167
17.1 La forma del programa 167
17.2 El dominio: tipos puros 170
17.3 El almacén: actor con estado 170
17.4 Persistencia: actor de escritura 172
17.5 Parser HTTP 173
17.6 El main: armar todas las piezas 174
17.7 Lo que está ocurriendo, en términos del libro 175
17.8 Cómo extenderlo 176

17.9 Lo que muestra este caso 176
Capítulo 18 · Caso de estudio: ledger contable 178
18.1 La forma del programa 178
18.2 El dominio: unidades, branding, tipos algebraicos 179
18.3 La invariante central: cuadre 180
18.4 El almacén: actor con invariantes 181
18.5 El log de auditoría 182
18.6 El main: ejecutar un escenario 182
18.7 Lo que hace este caso distinto del cap. 17 183
18.8 Cómo extenderlo 184
18.9 Por qué fintech es un buen banco de pruebas 184
18.10 Filosofía: el cierre del libro 185
Apéndices 186
Apéndice A · Bootstrap de tres etapas 187
A.1 El problema del bootstrap 187
A.2 Stage 0: el compilador en C 188
A.3 Stage 1: el compilador en kaikai-minimal 188
A.4 Stage 2: kaikai completo, self-hosted 188
A.5 El punto fijo: validación del bootstrap 189
A.6 Reproducible desde una `cc` 190
A.7 Costos y trade-offs 190
A.8 Para profundizar 191
Apéndice B · Perceus a fondo 192
B.1 El paper, en una frase 192
B.2 Análisis paso a paso: dónde van los drops 192
B.3 Reuse in place 193
B.4 Comparación con `Rc<RefCell<T>>` de Rust 194
B.5 Y los ciclos, ¿qué? 195
B.6 Por qué la separación por fibra simplifica el RC 195
B.7 Lo que cuesta y lo que se gana 196
B.8 Para seguir 196
Apéndice C · Tabla de operadores y precedencia 197
C.1 Tabla principal 197
C.2 Operadores que **no** son operadores 198
C.3 Equivalencias útiles 198
C.4 Recordatorio: comparación no asociativa 199
Apéndice D · Catálogo de efectos del stdlib 200
D.1 IO básico 200
D.2 Tiempo y aleatoriedad 201
D.3 Red 201
D.4 Procesos y señales 202
D.5 Estado 202
D.6 Errores y control 203
D.7 Concurrencia 204
D.8 Interoperabilidad 205
D.9 Composición: el alias `Io` 205
D.10 Handlers por defecto 205

Apéndice E · Glosario ... 206
A ... 206
B ... 206
C ... 207
D ... 207
E ... 207
F ... 207
H ... 208
I ... 208
L ... 208
M ... 208
N ... 208
O ... 209
P ... 209
R ... 209
S ... 210
T ... 210
U ... 210
V ... 210
Y ... 211
Apéndice F · Para seguir ... 212
F.1 Efectos algebraicos ... 212
F.2 Perceus y reference counting ... 212
F.3 Modelo de actores y BEAM ... 213
F.4 Concurrencia estructurada ... 213
F.5 Diseño de lenguajes ... 213
F.6 Sistemas de tipos ... 213
F.7 Contratos y diseño por contrato ... 214
F.8 La comunidad y el código ... 214
F.9 Cierre ... 214

Prólogo

> *Un* kaikai *es un hilo único que dibuja una figura entre las manos. En la cultura rapanui, esto corresponde al arte completo: el hilo, la figura, y el* pata'u ta'u *— el canto que la acompaña.*
>
> *Este libro toma esa imagen prestada.* Simple *viene del latín* simplex*: "un solo pliegue, un solo hilo".* Complex *viene de* complectere*: "trenzado junto". El argumento del libro, y la apuesta del lenguaje, es que la simplicidad no es lo opuesto de lo difícil, es lo opuesto de lo entrelazado.*
>
> *La etimología y la distinción entre* simple *y* easy *las desarrolla Rich Hickey en su charla* Simple Made Easy *(Strange Loop, 2011), disponible en InfoQ.*
>
> *La tesis de Hickey atraviesa todo este libro: simple es una propiedad objetiva de las cosas (cuántos hilos las componen); easy es una propiedad relativa al observador (cuán cerca está de lo que ya sabe). Kaikai apuesta por lo primero aunque le cueste lo segundo.*

Cuando aprendí a construir compiladores en la universidad quedé fascinado con el diseño de lenguajes de programación. Luego en mi vida laboral resolví varios problemas creando pequeños lenguajes (hoy en día se conocen como lenguajes de dominio específico, o DSL por sus siglas en inglés).

Me encanta el diseño de lenguajes de programación, y me gusta aprenderlos, compararlos, entender por qué sus autores tomaron las decisiones que tomaron. También aprendí por qué hay tantos lenguajes de programación. Hace años hice una charla en una conferencia donde explico por qué pasa esto, la pueden encontrar en YouTube (https://www.youtube.com/watch?v=Hp9HwLPYkjI). En aquella charla presenté a Ogú, un lenguaje que inventé y desarrollé hace varios años.

Ogú es un personaje de caricatura, un cavernícola amigo de Mampato creado por el ilustrador y caricaturista chileno Themo Lobos. Conseguí el permiso para usar el personaje como mascota del lenguaje, una de las gestiones de las que estoy más orgulloso. El repositorio nació hacia 2010, anuncié intenciones en el blog, escribí parsers, volví a empezar, escribí una gramática, la cambié, leí y releí varios de los libros clásicos sobre el tema (el del dragón, *Modern Compiler Implementation*, *Engineering a Compiler*), y avancé muy poco.

En 2017 hice un asalto serio: durante dos meses, con sesenta horas totales repartidas entre el trabajo y las noches, construí una primera versión usando Clojure como backend. En esencia Ogú es un transpiler: el compilador traduce la sintaxis de Ogú a *S-expressions* que son interpretadas por Clojure, y la JVM hace el resto. Un *"fake it 'til you make it"* lo suficientemente sólido como para hacer una demostración a mi audiencia en la conferencia.

Pero Ogú quedó abandonado. El último commit es de 2021. La organización en GitHub sigue ahí para quien quiera leerla. Reescribí el parser en Scala, un ejercicio de persistencia y algo de masoquismo. Mi problema era la generación de código, pero también mi ambición.

Un amigo me dijo que si iba a crear un lenguaje, respondiera qué tenía de novedoso ese lenguaje.

Aprendí nuevos lenguajes, en particular la familia funcional, como Haskell o F#. Me enamoré de Rust, y luché con el borrow checker y llegué a dominar su sintaxis para los "lifetimes" y su promesa de gestión de memoria sin garbage collection.

También aprendí sobre teoría de categorías y tuve una epifanía, que documenté en su momento en el post *Revelaciones* (2015).

Fue ahí que descubrí la teoría de las mónadas y de repente, cuando estaba sumergido en esa pasta, cayó en mis manos un post sobre efectos algebraicos. Ahí leí el ensayo *"What Color is Your Function?"* de Bob Nystrom. Y tuve una segunda epifanía.

Fue así como nació la idea de un nuevo lenguaje, que llamé kaikai.

Inicialmente el nombre era por la mítica serpiente de la mitología mapuche, pero descubrí que kai kai en la cultura Rapa Nui, hace referencia a un juego en que se hacen figuras con cordel que se tejen con los dedos mientras se canta un *pata'u ta'u*, un verso recitado. En el kai kai la estructura y la narración van de la mano, como en un programa bien tipado.

Lo que distingue a kaikai de intentos anteriores, y lo que le da originalidad y novedad, es lo que está en este libro: te toca a ti descubrirlo. Lo que quiero comentar tiene que ver con un hecho central en la creación de kaikai: el uso de la IA.

El compilador que se describe en este libro fue creado en un mes. Así como Ogú fue construido en 60 horas, kaikai fue construido con ayuda de Claude Code en un periodo de un mes. Algo que podría haber tomado años.

Mi reflexión fue la siguiente: la IA ha leído y entiende mejor que yo todos los papers sobre efectos algebraicos, programación funcional, diseño de lenguajes, etc. Puedo usar eso a mi favor.

Yo actué como arquitecto, ya tenía mucho diseño previo, la IA me ayudó a plasmar ese diseño.

En el camino inventé una forma de trabajar con la IA en el desarrollo del lenguaje en tiempo récord.

Ese método lo documenté en lo que llamo ELP: *Empirical Lane Parallelism*. Esta forma de usar los agentes para amplificar tu proceso de desarrollo y construir productos de software robustos en poco tiempo está documentada en mi repo: https://github.com/lnds/elp. Te invito a leerlo para que entiendas cómo pude construir un compilador tan complejo como el de kaikai en tan solo un mes.

Una nota sobre cómo se escribió este libro

Este libro se escribió con la asistencia de un agente de IA (Claude Code de Anthropic) bajo mi dirección como autor y editor. Yo decidí qué capítulos había, en qué orden, con qué voz, con qué énfasis. Claude redactó borradores que yo edité, corregí, devolví y volví a editar. Los ejemplos los validamos contra el compilador. Las afirmaciones técnicas las verificamos contra la documentación del lenguaje. La voz del libro la calibramos contra mi blog.

Lo menciono no como disculpa, sino como reconocimiento de la realidad que vivimos los ingenieros de software en 2026. La IA escribió partes, mi juicio decidió todo. En *El fin del software artesanal* escribí que los telares de Jacquard ya están entre nosotros y que la pregunta no es si los usaremos sino cómo. Este libro es una respuesta concreta a esa pregunta: lo que la IA permite no es prescindir del autor, es escribir libros que antes uno no se habría atrevido a empezar. Sin ese apoyo este texto probablemente seguiría en mi cabeza, como el dibujo de Ogú se quedó en la mía durante años.

Si encuentras inconsistencias, ejemplos que no compilan, afirmaciones desactualizadas o cualquier pifia: el lapsus es mío, no del agente. El libro está en el repositorio `kaikailang-org/kaikai-book` y los reportes se aceptan como issues o pull requests.

Convenciones

Vale gastar unos párrafos en cómo está organizado el libro y cómo leerlo. Nada de esto es difícil, pero saberlo de antemano ahorra tropiezos.

Estructura

El libro está organizado en cuatro partes, dieciocho capítulos y seis apéndices.

- **Parte I (caps. 1–2)** es el aterrizaje: un tour del lenguaje con programas ejecutables y un capítulo corto sobre cómo *pensar* en kaikai.
- **Parte II (caps. 3–11)** cubre el núcleo: tipos, funciones, módulos, protocolos, unidades de medida, contratos. Lo que necesitas para leer y escribir kaikai cotidiano.
- **Parte III (caps. 12–15)** entra en lo distintivo: efectos algebraicos, concurrencia por fibras, actores, holes tipados con asistencia de IA.
- **Parte IV (caps. 16–18)** cierra con tooling y dos casos de estudio.
- **Apéndices A–F** quedan como referencia: bootstrap del compilador, Perceus, operadores, catálogo de efectos del stdlib, glosario, lecturas adicionales.

Si vienes de un mundo funcional y solo te interesa lo nuevo, parte por la Parte III y vuelve al núcleo cuando te haga falta.

Forma de cada capítulo

Cada capítulo abre con un párrafo o dos de contexto: por qué importa el tema, qué problema resuelve. Después viene el cuerpo técnico, denso, con ejemplos. Los capítulos clave cierran con un **caso de estudio** que integra los conceptos en un programa realista. Al final hay **ejercicios** numerados, entre tres y ocho según el peso del capítulo.

Numeración y citas

- **Capítulos** son números enteros (cap. 7, cap. 12).
- **Secciones** llevan el número del capítulo (§7.3, §12.10).
- **Ejercicios** se citan como *7.3* dentro del propio capítulo, *cap. 7, ejercicio 3* desde otro.
- **Apéndices** son letras (apéndice A, apéndice D), con secciones tipo §A.1.

Tipografía

- **Negrita** se usa para introducir términos nuevos la primera vez que aparecen. Si una palabra está en negrita, es la definición.
- *Cursiva* se usa para énfasis y para títulos de obras citadas (*The Go Programming Language*, *Learn You a Haskell*).
- `Tipo monoespaciado` se usa para identificadores, código inline, nombres de archivo y comandos.

Código

Todos los bloques marcados como `kai` son ejecutables. Puedes copiarlos a un archivo `.kai`, compilarlos con `kai run`, y producirán la salida que el texto promete. Si un bloque aparece *sin* marca de lenguaje, es porque muestra una sesión de terminal: lo que viene después del `$` es lo que tipeas, lo que viene debajo es la salida.

```
$ kai run ejemplos/cap01/01_hola.kai
Hola, kaikai
```

Los ejemplos largos viven bajo `ejemplos/capNN/` en el repositorio del libro, y el texto los referencia por nombre cuando vale la pena bajarse el archivo entero.

Notación

El **lenguaje** se llama `kaikai`, siempre en minúscula, incluso al inicio de oración. La **herramienta de línea de comandos** se llama `kai`: la usas para compilar (`kai run`), correr tests (`kai test`), buscar propiedades (`kai check`), medir (`kai bench`).

Identificadores del lenguaje, palabras clave, mensajes del compilador y nombres de archivo se quedan siempre en inglés en ambas ediciones. Las palabras técnicas que ya forman parte del léxico de la profesión —*handler*, *fiber*, *effect row*, *pattern matching*— se usan en inglés sin cursiva. Los conceptos con traducción limpia al español (*tipo suma*, *unidad de medida*, *contrato*) sí se traducen.

Idioma

El libro se publica en dos ediciones: español e inglés. Ambas viven en el mismo repositorio, en árboles paralelos. Esta es la edición en español, redactada en español neutro latinoamericano informal. La edición en inglés no es traducción: se escribió en paralelo, con la misma estructura y los mismos ejemplos pero en la voz nativa de cada idioma.

Software vivo

El compilador, el stdlib y este libro están en evolución. Las versiones avanzan, los ejemplos a veces dejan de compilar entre versiones, y los apéndices se desactualizan. Si encuentras una discrepancia entre el libro y el compilador que tienes instalado, manda un issue al repo del libro. El libro indica al inicio de cada edición contra qué versión del compilador se validó.

Quién debería leer este libro

Programadores con experiencia en algún lenguaje: Python, JavaScript, Go, Java, C#, Rust, lo que sea, que tengan curiosidad por uno nuevo. No asumo background funcional: si nunca tocaste Haskell, OCaml o Elixir, los capítulos introductorios te apoyan. Si vienes de un mundo funcional, puedes saltar a la Parte III y leer directo lo distintivo de kaikai.

No es un libro para principiantes absolutos en programación. Asumo que sabes qué es una función, una lista, un tipo, un test.

Gracias

A quien lee mi blog desde hace más de veinte años: este libro existe porque ese diálogo existió. La constancia de los lectores, los comentarios, los correos, las correcciones, las discusiones que se prolongaron en Twitter primero y en newsletters después, me confirmaron una y otra vez que valía la pena escribir. Sin esa audiencia paciente kaikai habría seguido en mi cabeza con Ogú.

A Themo Lobos, que ya no está, por Ogú el cavernícola y por darle a tres generaciones de chilenos la convicción de que los mundos imaginarios se construyen con las manos. Recomiendo leer su novela gráfica *Mata-ki-te-rangi* (que se convirtió en el primer largometraje animado chileno: *Ogú y Mampato en Rapa Nui*). A los autores cuyas ideas kaikai recoge: Daan Leijen por Koka, Andreas Rossberg y Jonathan Brachthäuser por Effekt, Joe Armstrong por el espíritu BEAM, y a la comunidad académica que llevó los efectos algebraicos desde Plotkin y Pretnar hasta una herramienta usable. A los amigos que me bromeaban con Ogú y ahora me preguntan por kaikai.

Y a Anthropic, por construir una herramienta que me permitió escribir el compilador y este libro en un mes y no en la próxima década.

El compilador está vivo, el lenguaje está evolucionando, y la comunidad es pequeña pero atenta. Hay lugar para más.

Capítulo 1 · Tour de kaikai

La mejor forma de conocer un lenguaje es leerlo y correrlo. Este capítulo es un recorrido panorámico por kaikai en diez programas cortos. Ninguno pasa de unas treinta líneas, todos compilan, y juntos cubren las formas que vas a ver una y otra vez en el resto del libro: declaraciones, tipos algebraicos, pattern matching, efectos, fibras, protocolos, unidades de medida, contratos, pruebas inline y holes tipados.

No vamos a explicar cada detalle todavía. La idea es que termines el capítulo con el lenguaje mirado desde arriba y la sensación de que ya puedes leer código kaikai aunque te falten precisiones. Esas precisiones llegan en los capítulos siguientes.

Si quieres seguir los ejemplos en tu computador, los archivos están en `ejemplos/cap01/` del repositorio del libro. La instalación de `kai` viene al final del capítulo, en §1.12; si te urge, salta ahí primero y vuelve.

1.1 Hola, kaikai

Empezamos por lo más viejo del repertorio.

```
fn main() {
  println("Hola, kaikai")
}
```

```
$ kai run ejemplos/cap01/01_hola.kai
Hola, kaikai
```

Cuatro cosas que mirar antes de seguir:

- Todo programa kaikai parte en `fn main()`. No hay archivo de configuración, no hay `package main`, no hay clase contenedora. Una función con ese nombre, en algún archivo, alcanza.
- `fn` declara funciones. Es una palabra clave corta a propósito; vas a escribirla mucho.
- Las llaves `{ … }` agrupan un bloque de instrucciones, pero un bloque también es una expresión: el último valor que produce es el valor del bloque. Acá no nos interesa, pero lo vas a usar.
- `println` no requiere `import`. Está disponible en todos los programas porque escribe a la salida estándar mediante un efecto que kaikai trae por defecto. En el capítulo 12 vamos a abrir esa caja; por ahora basta con que funciona.

No hay punto y coma al final de la línea. No hay `return` para funciones que no devuelven nada. Tampoco hace falta declarar el tipo de retorno de `main` cuando no devuelve un valor útil. Todo eso es por diseño: kaikai trata de no pedirte que escribas lo obvio.

1.2 Tipos algebraicos y `match`: FizzBuzz

El típico ejercicio de la entrevista, escrito en kaikai, se ve así:

```
type Tag
  = Both
  | Fizz
  | Buzz
  | Other(Int)

fn classify(n: Int) : Tag {
  if n % 15 == 0     { Both }
  else if n % 3 == 0 { Fizz }
  else if n % 5 == 0 { Buzz }
  else               { Other(n) }
}

fn label(c: Tag) : String {
  match c {
    Both     -> "FizzBuzz"
    Fizz     -> "Fizz"
    Buzz     -> "Buzz"
    Other(n) -> int_to_string(n)
  }
}

fn loop(i: Int, n: Int) : Unit / Stdout {
  if i <= n {
    println(label(classify(i)))
    loop(i + 1, n)
  }
}

fn main() {
  loop(1, 15)
}
```

Lo interesante de esta versión no es que imprima `1, 2, Fizz, 4, Buzz, ...`. Eso lo hace cualquier lenguaje. Lo interesante es qué hicimos para llegar ahí.

Definimos un **tipo suma**: `Tag` es uno de cuatro constructores. Tres son nombres pelados (`Both`, `Fizz`, `Buzz`) y uno carga un dato (`Other(Int)`). Si vienes de un lenguaje imperativo, esto se parece a un `enum` con datos. Si vienes de un lenguaje OO, se parece a una jerarquía sellada de subclases. La diferencia es que en kaikai esta declaración no trae herencia, no trae métodos virtuales, no trae nada salvo lo que ves: cuatro maneras de construir un valor de tipo `Tag`.

`classify` decide cuál de los cuatro construir. Fíjate en el `if`: no tiene `then`, no tiene paréntesis alrededor de la condición, y cada rama es un bloque que produce un valor. El `if` mismo

es una expresión que devuelve `Tag`, y el cuerpo de la función es esa expresión, sin `return` ni asignación intermedia. Esto es lo que en el capítulo 2 vamos a llamar **expresión, no sentencia**, y es uno de los pocos cambios de hábito que vas a tener que hacer.

`label` consume un `Tag` con `match`. Cada rama es un **patrón** seguido de `->` y la expresión que devuelve. El patrón `Other(n)` no solo dice "es del constructor `Other`", también desempaca el dato y lo amarra al nombre `n`, listo para usarse a la derecha. Es destructurar, comparar y declarar una variable en un solo paso.

`loop` es recursivo. No hay `while`, no hay `for`. Bueno, no para esto: en el capítulo 6 vamos a ver que kaikai sí tiene formas más cómodas para iterar, pero la base es la recursión. Y para que esa base no le cueste a tu programa, el lenguaje garantiza **tail-call optimisation**: una llamada recursiva en posición de cola no consume stack. `loop(1, 1_000_000)` funciona sin reventar.

Una sola cosa que va a parecer rara y que dejamos para el capítulo 12: la firma de `loop` dice `: Unit / Stdout`. La parte después del `/` es el conjunto de **efectos** que la función usa. `Stdout` significa "esta función escribe al terminal". Si no estuviera ahí, el compilador no te dejaría llamar a `println` adentro. Por ahora no te preocupes; el detalle viene completo más adelante.

1.3 Una calculadora con AST recursivo

Pasemos a algo con un poco más de chicha. Una calculadora muy simple, con expresiones aritméticas representadas como árbol.

```
type Expr
  = Lit(Int)
  | Add(Expr, Expr)
  | Mul(Expr, Expr)
  | Neg(Expr)

fn eval(e: Expr) : Int {
  match e {
    Lit(n)    -> n
    Add(l, r) -> eval(l) + eval(r)
    Mul(l, r) -> eval(l) * eval(r)
    Neg(x)    -> -eval(x)
  }
}

fn main() {
  let e = Add(Lit(2), Mul(Lit(3), Lit(4)))
  println(int_to_string(eval(e)))
}
```

```
$ kai run ejemplos/cap01/03_calculadora.kai
14
```

`Expr` es un tipo suma como el de FizzBuzz, con una diferencia: **se menciona a sí mismo en sus propios constructores**. `Add` recibe dos `Expr`. `Mul` también. `Neg` recibe uno. El resultado es que un valor de tipo `Expr` puede ser un árbol de cualquier profundidad.

Esa es la herramienta clave para representar lenguajes, configuraciones, consultas, comandos, casi cualquier estructura con anidamiento. La vas a ver mucho. En el capítulo 5 dedicamos una sección entera a este patrón.

`eval` recorre el árbol con `match`. Cada caso recurre sobre los hijos. La exhaustividad la verifica el compilador: si agregas un constructor a `Expr` y se te olvida una rama en `eval`, no compila. Esto es enorme y va a salvarte muchas horas. El capítulo 5 lo explora con calma; por ahora confía.

`let` introduce un binding local. El tipo se infiere del lado derecho. No hay `var`, no hay `mutable`, no hay reasignación: `let e = ...` ata `e` a un valor y ese valor no cambia. Si necesitas mutar algo, kaikai te lo permite, pero te pide declararlo (capítulo 13). Esta es la otra mitad del cambio de hábito: **inmutabilidad por defecto**.

1.4 Un efecto propio con su handler

Hasta ahora todo el "efecto" que vimos fue `println`, que funciona porque kaikai trae un handler por defecto. Veamos qué pasa cuando declaramos un efecto nuevo.

```
effect Log {
  log(msg: String) : Unit
}

fn greet(name: String) : Unit / Log {
  Log.log("hola, " ++ name)
}

fn main() {
  handle {
    greet("kaikai")
    greet("mundo")
  } with Log {
    log(msg, resume) -> {
      println("[INFO] " ++ msg)
      resume(())
    }
  }
}
```

```
$ kai run ejemplos/cap01/04_efecto.kai
[INFO] hola, kaikai
[INFO] hola, mundo
```

Este es el ejemplo que más probablemente te haga frenar la lectura. Es deliberado. Los efectos algebraicos son la apuesta distintiva de kaikai y queremos que los veas funcionando antes de que te los expliquemos en serio.

Lo que está pasando es lo siguiente:

- `effect Log { log(msg: String) : Unit }` declara un nuevo efecto llamado `Log` con una operación, `log`, que recibe un string y devuelve nada.
- `greet` usa esa operación. Su firma, `: Unit / Log`, declara que la función tiene el efecto `Log`, sin decir cómo se realiza ese efecto. `greet` es agnóstica: no sabe si los mensajes van al terminal, a un archivo, a la nada.
- Quien decide es `handle ... with Log { ... }`. Ahí, en el cuerpo de `main`, decimos: "para este bloque, cuando alguien invoque `Log.log(msg)`, ejecuta este código". El handler imprime el mensaje con un prefijo `[INFO]` y le pasa la ejecución a `resume(())`, que continúa el programa donde había quedado.

Esto se parece a try/catch, a un dependency injection container, a un middleware, a callbacks. Pero **es una sola idea** que subsume a las cuatro. Si la primera vez te confunde, está bien. Volvemos en el capítulo 12 con tiempo y con varios ejemplos antes de pedirte que escribas un handler tuyo.

Lo que sí conviene retener desde ya: el tipo de `greet` te dice que necesita `Log`. El compilador no te deja llamarla desde un contexto donde `Log` no esté manejado. Los efectos son **visibles en el tipo**, no escondidos. Esto resuelve una incomodidad vieja de los lenguajes que tienen excepciones invisibles.

1.5 Dos fibras cooperativas

El quinto programa del tour usa concurrencia.

```
import spawn

fn worker(tag: String, n: Int) : Unit / Stdout + Spawn {
  if n > 0 {
    println(tag)
    spawn.yield()
    worker(tag, n - 1)
  }
}

fn main() {
  let f = spawn.spawn(() => worker("B", 3))
  worker("A", 3)
  spawn.await(f)
}
```

```
$ kai run ejemplos/cap01/05_concurrente.kai
A
B
A
B
A
B
```

Una **fibra** es una unidad de ejecución cooperativa. Pesa mucho menos que un thread del sistema operativo y vive dentro del proceso. `spawn.spawn` agenda una fibra nueva pero no la arranca de inmediato; el scheduler la levanta en el próximo punto de cooperación.

`spawn.yield` es justamente eso: un punto donde la fibra actual dice "puedo esperar, dale paso a otra".

Sin los `spawn.yield`, el `worker` "A" correría sus tres iteraciones antes de soltarle el turno a "B". Con ellos, la salida queda alternada.

La firma de `worker` es `: Unit / Stdout + Spawn`. Dos efectos: el que ya conocíamos para imprimir, y `Spawn` para levantar y coordinar fibras. El operador `+` compone efectos: una función puede tener varios al mismo tiempo, declarados en su tipo.

`(() => worker("B", 3))` es una **lambda**: una función anónima sin parámetros que llama a `worker`. La pasamos como argumento a `spawn.spawn` para que la corra dentro de la fibra nueva.

Hay mucho que decir sobre el modelo de concurrencia de kaikai (por qué las fibras son aisladas, cómo se cancelan, qué pasa con la memoria), pero todo eso vive en el capítulo 13. Lo que importa para el tour es que el lenguaje tiene concurrencia estructurada de primera clase y que se trata, una vez más, como un efecto.

1.6 Tipos a la medida con protocolos

A esta altura ya viste tipos primitivos y tipos suma. Falta una construcción más: los **records**, que son lo que en la mayoría de los lenguajes llamarías un *struct*: un agregado con campos nombrados.

```
type Punto = { x: Int, y: Int }
```

Y con eso aparece la pregunta natural: ¿cómo se le "agregan operaciones" a un tipo? Por ejemplo, ¿cómo le decimos al compilador que mi `Punto` sabe imprimirse como string?

La respuesta de kaikai son los **protocolos**: un contrato con nombre y un puñado de operaciones, que cualquier tipo puede satisfacer. Es el equivalente conceptual a las interfaces de Go, los traits de Rust o los protocols de Clojure y Elixir.

```
#[derive(Show)]
type Punto = { x: Int, y: Int }

fn main() {
  let p = Punto { x: 3, y: 4 }
  println(show(p))
}
```

```
$ kai run ejemplos/cap01/07_protocolos.kai
Punto { x: 3, y: 4 }
```

`Show` es uno de los protocolos del stdlib (`Eq`, `Ord`, `Hash`, `Show`, `Serialize`). Su contrato es una sola operación: dado un valor, devolver un `String`. La línea `#[derive(Show)]` arriba del record le dice al compilador que **genere automáticamente** una implementación de `Show` para `Punto`, recorriendo los campos y delegando en el `Show` de cada uno. Como `Int` ya implementa `Show` en el stdlib, el record entero queda cubierto sin que tengamos que escribir nada más.

Si quisieras una implementación a mano, en vez de `#derive` escribirías:

```
impl Show for Punto {
  fn show(p: Punto) : String =
    "(" ++ show(p.x) ++ ", " ++ show(p.y) ++ ")"
}
```

Y `show(Punto { x: 3, y: 4 })` ahora devolvería `"(3, 4)"` en vez del formato del record.

Lo importante para el tour: **kaikai elige single-dispatch explícito**, no typeclasses al estilo Haskell. No hay inferencia de constraints, no hay tipos de orden superior, no hay polimorfismo paramétrico ad-hoc encadenado. Una sola mecánica simple, igual que en Go o Clojure. El capítulo 9 desarrolla la idea.

1.7 Unidades de medida

kaikai trae una herramienta poco común en lenguajes *mainstream*: las **unidades de medida**. F# las tiene desde 2010 y prácticamente ningún otro lenguaje las ofrece de fábrica. La idea es marcar un número con una unidad (`Real<USD>`, `Real<m/s>`, `Int<Seconds>`) y dejar que el compilador rechace mezclas incompatibles.

```
unit USD
unit EUR

fn main() {
  let precio : Real<USD> = 1.50<USD>
  let total  : Real<USD> = precio + 2.00<USD>
  println("total = #{total}")
}
```

```
$ kai run ejemplos/cap01/08_unidades.kai
total = 3.5 USD
```

`unit USD` declara una unidad. `1.50<USD>` es un literal anotado. `Real<USD>` es el tipo de un real con esa unidad. Y si intentas:

```
let mezcla = precio + 1.00<EUR>     # error: USD ≠ EUR
```

el compilador se queja antes de que el programa corra. Esto captura una clase entera de bugs que normalmente se descubren en producción: el clásico de Mars Climate Orbiter[1], el de sumar saldos en monedas distintas, el de pasar un timeout en milisegundos donde se esperaban segundos.

Lo más bonito del esquema es que **las unidades se borran en tiempo de compilación**. El binario que produce `kai build` opera con `Real` plano, sin overhead. Es la misma promesa de los efectos: información en el tipo, costo cero en runtime.

[1] La sonda Mars Climate Orbiter de la NASA se perdió en septiembre de 1999 al entrar en la atmósfera marciana. La causa raíz: un módulo de software calculaba el empuje en libras-fuerza por segundo (unidades imperiales) y el otro leía ese valor como newtons por segundo (unidades métricas). Nadie había anotado las unidades en la interfaz. La misión costó 327 millones de dólares.

El tema da para mucho más: unidades genéricas, álgebra de unidades (`m/s^2`, `kg * m / s^2`), conversiones explícitas, y una variante muy útil llamada *branded types* que marca strings y enteros con tags como `UserId` o `OrderId` para que el compilador no te deje confundirlos. Todo eso aparece en el capítulo 10. Por ahora basta saber que existe.

1.8 Programación por contrato

kaikai trae otra herramienta heredada de pocos lenguajes (Eiffel en los ochenta, Ada 2012, D) para declarar lo que una función espera de quien la llama y lo que garantiza a cambio: las **precondiciones** y **postcondiciones**.

```
fn divide(a: Int, b: Int) : Int
  requires b != 0
  ensures  result * b + (a % b) == a
= a / b

fn main() {
  println("10 / 2 = #{divide(10, 2)}")
  println("17 / 3 = #{divide(17, 3)}")
}
```

```
$ kai run ejemplos/cap01/09_contratos.kai
10 / 2 = 5
17 / 3 = 5
```

`requires b != 0` dice "esta función exige que `b` no sea cero al momento de llamarla". El compilador hace dos cosas con esa precondición: si puede **probarla en tiempo de compilación** (porque los argumentos son literales o porque ya conoce los rangos posibles), rechaza la llamada antes de emitir código: `divide(10, 0)` literal es un error de compilación, no de ejecución. Si los argumentos son dinámicos y el compilador no alcanza a decidir, inserta un assert que se verifica **al entrar** a la función, y el programa aborta si la precondición falla.

`ensures result * b + (a % b) == a` dice "esta función garantiza que la identidad fundamental de la división entera se cumple al salir". `result` es un nombre reservado dentro del `ensures` que se refiere al valor de retorno. Esta postcondición se verifica **al salir** del cuerpo: si por algún bug interno la función devolviera algo que no cumple, el programa también aborta.

Los contratos no son comentarios. Son código que el compilador emite como verificaciones reales: estáticas cuando puede, dinámicas cuando hace falta. El día que algo viole un contrato, vas a saberlo en el lugar exacto.

¿En qué se diferencian de las pruebas que vienen en la próxima sección? Una prueba dice "para esta entrada específica, espero esta salida específica". Un contrato dice "para **toda** entrada que cumpla esta precondición, la salida cumple esta postcondición". Una es un caso fijo; el otro, una promesa universal documentada en la firma.

Hay un mecanismo hermano que kaikai usa para extender la misma idea a los **valores**, no a las operaciones: los **refinement types**, que te dejan declarar tipos como `Int where >= 0` o `Real where 0.0 <= self <= 1.0`. Son la contraparte estructural de los contratos: el tipo describe

qué valores son válidos, el contrato describe qué hacen las operaciones con ellos. Los dos viven juntos en el capítulo 11.

1.9 Pruebas en el mismo archivo

kaikai trata las pruebas como ciudadanas de primera: viven en el mismo archivo que el código que prueban, con su propia sintaxis al lado de las funciones.

```
fn cuadrado(n: Int) : Int = n * n

test "cuadrado de cero" {
  assert cuadrado(0) == 0
}

test "cuadrado preserva positivos" {
  assert cuadrado(7) == 49
}

test "cuadrado de negativos" {
  assert cuadrado(-5) == 25
}
```

```
$ kai test ejemplos/cap01/06_pruebas.kai
  ok   cuadrado de cero
  ok   cuadrado preserva positivos
  ok   cuadrado de negativos

3/3 tests passed
```

`test "..." { ... }` es un bloque top-level. Adentro usas `assert` para escribir aserciones: si una falla, el test falla y el runner sigue con los siguientes. En una build normal (`kai run`, `kai build`) los bloques `test` se ignoran: no agregan peso al binario que despliegas.

Hay dos parientes cercanos que usan la misma forma:

- **`check "..." with x: T { ... }`** declara una **propiedad** que el runner verifica con valores generados al azar. Es lo que en otros lenguajes se llama property-based testing.
- **`bench "..." { ... }`** es un benchmark: el runner ejecuta el bloque muchas veces y reporta nanosegundos por iteración.

Las tres formas se complementan: `test` para casos fijos, `check` para invariantes que deben valer sobre cualquier entrada, `bench` para medir rendimiento sin adivinar. El capítulo 7 entra en cada una con tiempo.

1.10 Holes: agujeros que compilan

Hay una última construcción del lenguaje que vale la pena ver en el tour, porque cambia un poco la forma de escribir código. kaikai te permite dejar **agujeros** en lugares donde todavía no sabes qué poner, y el programa compila igual.

```
fn area_circulo(r: Real) : Real = ?formula

fn perimetro_circulo(r: Real) : Real = ?

fn main() {
  println("compiló: el cuerpo está pendiente")
}
```

```
$ kai run ejemplos/cap01/10_holes.kai
compiló: el cuerpo está pendiente
```

`?` y `?nombre` son **expresiones tipadas**. El compilador acepta el programa, infiere el tipo esperado en cada agujero (en `?formula`, sabe que tienes que producir un `Real`), y te deja el resto del archivo compilando. Si alguien llama a `area_circulo` en runtime sin haber rellenado el agujero, el programa aborta con `panic: unfilled hole`. Como `main` no la llama, el programa de arriba termina bien.

¿Para qué sirve esto? Para tres cosas:

- **Diseñar de arriba hacia abajo.** Escribes la firma de la función, dejas el cuerpo en `?`, y compilas. El compilador te dice qué tipo se espera ahí y qué valores tienes en alcance. Conversas con el compilador antes de escribir el cuerpo.
- **Avanzar con un programa parcial.** Tienes diez funciones por escribir, pero quieres que el programa compile y correr la primera para ver si la idea va. Las otras nueve quedan en `?`; el código compila; pruebas la primera; el resto espera.
- **Trabajar con un agente IA.** Le pasas la firma con holes al agente y le pides que los rellene. La doc del compilador se diseñó para que esa información (tipo esperado, bindings en alcance, candidatos posibles) se pueda emitir como JSON estructurado, listo para alimentar al agente.

Las dos primeras son útiles para el programador humano. La tercera es la apuesta más estratégica del lenguaje: kaikai quiere ser un lenguaje en el que un LLM pueda escribir bien aun cuando su corpus de entrenamiento contenga poco kaikai, porque el compilador hace gran parte del trabajo. El capítulo 15 entra en esa apuesta con tiempo.

1.11 Dependencias y proyectos: `kai.toml`

Hasta acá los ejemplos del tour fueron archivos sueltos. La realidad de un proyecto es distinta: vas a tener varios archivos, vas a depender de bibliotecas que otros publicaron, y vas a querer que tu colega clone el repo y obtenga exactamente la misma compilación que la tuya.

kaikai resuelve esto con un manifest mínimo, **`kai.toml`**:

```
name = "mi_app"
version = "0.1.0"

[dependencies]
manutara = "github.com/kaikailang-org/manutara@v0.1"
local    = { path = "../local-thing" }
```

El flujo del día a día son tres comandos:

```
$ kai init                                   # crea kai.toml en el directorio actual
$ kai add github.com/kaikailang-org/manutara@v0.1         # agrega una dependencia
$ kai run main.kai                              # compila y corre
```

`kai add` clona el repositorio de la dependencia, lo cachea bajo `~/.cache/kai/pkg/` direccionado por SHA, y actualiza dos archivos: `kai.toml` (qué se quería) y **`kai.lock`** (qué versión exacta se resolvió). El lockfile es lo que garantiza **builds reproducibles**: cuando tu colega clone el repo y corra `kai install`, va a obtener bit-por-bit las mismas dependencias que tú.

La resolución de dependencias es **git-first**: una URL puede ser un tag (`@v0.1`), una rama (`@main`) o un commit específico (`@abc123`). No hace falta registry centralizado, no hace falta TLS, no hace falta autenticarse. Si el repo está en GitHub, GitLab o tu servidor privado de git, kaikai sabe ir a buscarlo.

El capítulo 8 cubre esto con detalle: cómo organizar un proyecto en módulos, qué entra en `[dependencies]`, cómo funciona la selección de versiones cuando dos dependencias piden cosas distintas. Por ahora basta saber que existe y que **no necesitas un sistema de build externo**: `kai` es el único binario.

1.12 Cómo instalar y correr `kai`

Para correr cualquiera de los programas anteriores necesitas el binario `kai`. El proyecto está en github.com/kaikailang-org/kaikai.

Desde un checkout fresco, lo único que se requiere es un compilador de C:

```
$ make all
$ make test
```

`make all` construye stage 0 (escrito en C), stage 1 (escrito en kaikai-minimal y compilado por stage 0) y el binario `kai` del directorio raíz. `make test` corre las baterías de stage 0, stage 1 y phase 4 para confirmar que la compilación quedó bien.

A partir de ahí, los comandos que vas a usar a lo largo del libro son tres:

```
$ kai run archivo.kai     # compila y ejecuta
$ kai build archivo.kai -o nombre   # produce un binario nativo
$ kai test archivo.kai    # ejecuta los bloques `test "..." { ... }` del archivo
```

`kai run` es el caballo de batalla mientras lees el libro. Edita un archivo, córrelo, mira la salida, vuelve a editar.

El capítulo 16 cubre el resto del tooling: `fmt`, `lsp`, integración con editores, y los comandos de paquetes (`init`, `add`, `install`, `update`). Por ahora con `run` te basta.

1.13 Cómo está organizado el resto del libro

Vimos en este capítulo, sin profundizar, prácticamente todo lo que hace distinto a kaikai. El resto del libro toma cada cosa y la trata en serio.

- **Parte II: El lenguaje** (capítulos 3 a 11) cubre los tipos básicos, los tipos compuestos, los tipos suma y `match`, las funciones, las pruebas y benchmarks, los módulos, los protocolos, las unidades de medida y la programación por contrato. Es la mitad sólida y predecible.
- **Parte III: Lo distintivo** (capítulos 12 a 15) toma los efectos algebraicos, la concurrencia con fibras, los actores y la apuesta del lenguaje en torno a los LLMs. Es la mitad donde kaikai paga su novedad.
- **Parte IV: Práctica** (capítulos 16 y 17) se ocupa del tooling y cierra con un caso de estudio integrador.
- Antes, el **capítulo 2** te ablanda algunas asunciones si vienes de un mundo imperativo: expresiones vs sentencias, inmutabilidad por defecto, `Option` en vez de `null`, efectos visibles. Es un capítulo corto pero útil.

Si vienes de Haskell, OCaml, Elixir o Scala, puedes saltarte el capítulo 2 e incluso ojear rápido la Parte II; lo nuevo para ti vive en la Parte III. Si vienes de Python, Go, Java, JavaScript o C#, lee el capítulo 2 sin prisa y haz los ejercicios de la Parte II.

En cualquier caso: el código de los ejemplos está en `ejemplos/` del repositorio del libro. Compila todo. Corre todo. La única forma de aprender un lenguaje es escribirlo.

Capítulo 2 · Pensar en kaikai

El capítulo 1 te mostró el lenguaje desde arriba. Antes de bajar al detalle de tipos, funciones y módulos, conviene detenerse un momento en algunos hábitos que kaikai pide y que probablemente no traigas si vienes de Python, Java, Go, JavaScript o C#.

Este es el capítulo más corto del libro, y se puede saltar. Si ya programaste en Haskell, OCaml, Elixir o Scala, esto te va a sonar familiar; pasa a la Parte II y nos vemos en el capítulo

3. Si vienes de un mundo imperativo, dedícale los veinte minutos

que pide. Te van a ahorrar incomodidades en los siguientes ciento cincuenta páginas.

No vamos a abrir la teoría de cada idea: eso es trabajo de los capítulos siguientes. Lo que queremos es nombrar el cambio de hábito, mostrarlo, y darle al lector las palabras para reconocerlo cuando aparezca.

2.1 Expresiones, no sentencias

En la mayoría de los lenguajes que probablemente conoces, el código se construye con dos tipos distintos de pieza:

- **Expresiones**, que producen un valor: `x + 1`, `f(2)`, `a == b`.
- **Sentencias**, que no producen valor sino que ejecutan algo: un `if` con dos ramas, un `for`, un `return`, una asignación.

Las sentencias necesitan armarse en una secuencia. Las expresiones, no: se componen anidándose.

kaikai borra esa frontera. **Casi todo es expresión.** Un `if` produce un valor. Un `match` produce un valor. Un bloque `{ … }` produce un valor: el de la última expresión adentro. Una función no necesita `return` porque su cuerpo *es* la expresión que devuelve.

Compara dos formas de escribir lo mismo. En el estilo imperativo clásico:

```
String s;
if (x > 0) {
  s = "positivo";
} else {
  s = "no positivo";
}
print(s);
```

En kaikai:

```
let s = if x > 0 { "positivo" } else { "no positivo" }
println(s)
```

La diferencia no es de líneas. Es de pensamiento. En la versión imperativa hay que **declarar `s` primero**, porque el `if` no sabe devolver nada; después hay que **mutar `s` en cada rama**. En kaikai, el `if` *es* el valor, y `s` se ata directo al resultado. No hay declaración separada de la asignación. No hay asignación en absoluto: `s` se ata una vez y no cambia.

Esto tiene consecuencias prácticas que vas a notar pronto:

- **Menos líneas, sin perder claridad.** Lo que antes eran tres pasos (declarar, decidir, asignar) se convierte en uno.
- **Menos variables temporales.** Si solo necesitas un valor para pasarlo a la siguiente función, lo armas inline.
- **Menos errores de "olvidé inicializar".** No hay variables declaradas-pero-sin--valor.
- **Refactor más fluido.** Una expresión se puede extraer a una función o reemplazar por otra expresión sin tocar el contexto alrededor; una sentencia, no tanto.

Vas a ver lo mismo en `match`. En la mayoría de los lenguajes con `switch`, cada `case` es una sentencia que ejecuta un bloque y después rompe (o sigue, según las reglas del lenguaje). En kaikai, `match` es una expresión que devuelve un valor, y cada rama es la expresión que ese valor podría ser. Lo viste en el capítulo 1, en `label` y en `eval`. Volverás a verlo constantemente.

Una pieza relacionada: el cuerpo de una función puede tomar dos formas, y la elección entre ellas comunica intención.

```
fn doble(x: Int) : Int = x * 2

fn clasificar(x: Int) : String {
  if x < 0 { "negativo" }
  else if x == 0 { "cero" }
  else { "positivo" }
}
```

Con `=` y una sola expresión, cuando la función es directa. Con `{ ... }` cuando hay varios pasos o conviene visualmente separar. El compilador acepta ambas; la diferencia es para quien lee.

Una consecuencia útil de tener todo como expresión es que las **transformaciones encadenadas** se vuelven cómodas. kaikai trae del mundo Elixir el operador pipe `|>`:

```
xs |> filter(es_par) |> map(doble) |> list.length
```

equivale a `list.length(map(filter(xs, es_par), doble))`. La versión con pipe se lee de izquierda a derecha, en el orden en que ocurren las transformaciones, y no requiere variables intermedias. Solo es posible porque cada paso es una expresión que se puede componer con la siguiente.

`|>` es el más general de cuatro operadores que kaikai ofrece para encadenar. Los otros tres (`|` map sobre listas, `||` flat-map y `|?` filter) son atajos para los casos que aparecen una y otra vez. Los vemos en detalle en el capítulo 6.

2.2 Inmutabilidad por defecto

`let x = 5` ata `x` al valor `5`. Eso es todo. No hay forma de escribir `x = 6` después. Si tu código necesita "cambiar `x`", en kaikai eso significa una de dos cosas:

- En realidad necesitas un valor nuevo, derivado del primero. La forma correcta es atar otro nombre, o redefinir `x` en un ámbito interno.
- En realidad necesitas mutación visible. Eso es un caso real pero pequeño, y kaikai te lo da, pero te pide declararlo. La mutación de un array, por ejemplo, vive bajo el efecto `Mutable`, que aparece en la firma de cualquier función que la use. Lo veremos con calma en el capítulo 13.

¿Por qué tomar este camino? Porque la mayoría de las veces "cambiar una variable" es una limitación heredada del modelo imperativo, no una necesidad real. Cuando programas con valores que no cambian:

- **El razonamiento se vuelve local.** Si `x` se ató a `5` en la línea 12, sigue siendo `5` en la línea 30. Punto. No hay que buscar quién lo modificó en el medio.
- **La concurrencia se simplifica.** Dos fibras pueden mirar el mismo valor sin sincronización; nadie va a sobrescribirlo.
- **Los bugs disminuyen.** Una clase entera de errores ("esperaba X, pero al final del método era Y") simplemente no existe.
- **Los tests son más simples.** Una función pura (entrada, salida, sin estado escondido) se prueba dándole entradas y comparando salidas. Eso es todo.

Una nota sobre vocabulario. La forma habitual de atar un nombre es `let`, que es inmutable. Para los casos en que de verdad necesitas una celda mutable local (un contador, un acumulador, un cursor), kaikai te da `var`, junto a dos azúcares que acompañan: `@nombre` para leer la celda y `nombre := v` para escribirla.

```
var n = 0
n := @n + 1
println(int_to_string(@n))   # 1
```

Acá está la cosa interesante: `var` no es realmente una construcción nueva del lenguaje, es **azúcar sintáctico** sobre el efecto `State`. La línea `var n = 0` se reescribe a un `handle ... with State[Int](0) as n { ... }` que abarca el resto del bloque. `@n` se reescribe a `n.get()`, y `n := v` a `n.set(v)`. El lenguaje base es el mismo de los efectos algebraicos del capítulo 12; lo que cambia es la cara que muestra para los casos comunes.

Lo importante para tu modelo mental es que esa traducción ocurre **dentro del bloque**: el `handle` se abre y se cierra ahí mismo, así que el efecto `State` no se asoma a la firma de la función. Una función con `var` adentro tiene la misma firma que si no lo tuviera.

Mutaciones más visibles (escribir un array que vive más allá del bloque, enviar a la mailbox de otro actor, modificar memoria que se observa desde fuera) sí aparecen en la

firma, bajo efectos como `Mutable`, `Actor` o los que correspondan. Esa distinción la veremos en el capítulo 13.

La regla práctica es simple: usa `let` por defecto; si necesitas una variable local que cambia, `var` con `@` y `:=`; si lo que quieres mutar es algo visible desde afuera, ya estamos en territorio de efectos y vas a tener que declararlos.

2.3 `Option` y `Result` en vez de `null` y excepciones

La pregunta más vieja al diseñar un lenguaje: ¿qué hace una función cuando no puede devolver lo que prometió?

La respuesta de C, Java, Python, JavaScript y un largo etcétera es **mentir**: la función dice que devuelve un `Usuario`, pero en algunos casos devuelve una variable mágica llamada `null` (o `None`, o `nil`) que **no es un usuario** y que el sistema de tipos no distingue del valor real. El que llama tiene que recordar comprobarlo. Tony Hoare, que inventó la referencia nula en 1965, llamó a esa decisión su "error de mil millones de dólares".

La otra respuesta tradicional es lanzar una excepción. La función no devuelve nada y, sin avisar al sistema de tipos, desvía el control a algún `catch` lejano. Cuál `catch`, eso depende. A veces no hay ninguno y el programa muere.

kaikai elige una tercera vía, vieja en la familia ML pero todavía poco común fuera de ella: **codificar la posibilidad de fallar en el tipo de retorno**.

```
type Option[a] = None | Some(a)
type Result[e, a] = Err(e) | Ok(a)
```

Una función que puede no encontrar el resultado devuelve `Option[Usuario]`: o `Some(usr)` cuando lo encuentra, o `None` cuando no. Una función que puede fallar de varias maneras devuelve `Result[Error, Usuario]`: o `Ok(usr)`, o `Err(razón)`. En ambos casos, el tipo te obliga a considerar las dos posibilidades.

Compara:

```
# Python: el que llama tiene que recordar
def buscar(id: int) -> Usuario:
    ...  # a veces devuelve None, a veces no, mira la doc

usr = buscar(7)
print(usr.nombre)  # crash si usr es None
```

```
# kaikai: el tipo lo dice
fn buscar(id: Int) : Option[Usuario] = ...

let r = buscar(7)
match r {
  Some(usr) -> println(usr.nombre)
  None      -> println("no encontrado")
}
```

En la versión kaikai, el `match` es exhaustivo: si te olvidas del caso `None`, no compila. El compilador te recuerda lo que en Python depende de tu memoria.

¿Y las excepciones? kaikai tiene un mecanismo equivalente (el efecto `Fail` y, más en general, los efectos algebraicos del capítulo 12), pero también ahí lo que puede fallar aparece en el tipo. Las "excepciones invisibles" que en Java o Python pueden brotar de cualquier llamada, en kaikai no existen. Si una función puede saltar a otro lado, su firma lo declara.

Esto cambia la sensación de programar. En vez de envolver cada llamada externa en un `try` por las dudas, lees la firma, ves qué puede fallar, y decides ahí mismo qué hacer.

Una nota sobre `!`

En kaikai, el operador postfix `!` aplica a un `Option` o un `Result` y propaga el caso negativo: si el valor es `Ok(v)` o `Some(v)`, la expresión vale `v` y el programa sigue; si es `Err(e)` o `None`, la función actual termina ahí mismo devolviendo ese `Err` o `None` a quien llama.

```
fn cargar() : Result[Error, Usuario] {
  let id = parsear_id(input)!       # si falla, propaga
  let datos = leer_archivo(id)!     # idem
  Ok(armar_usuario(id, datos))
}
```

Es lo mismo que el `?` de Rust. Y conviene mencionar lo que **no** es: en Elixir hay una convención de nombrar `File.read!` a la versión que lanza excepción en vez de devolver un tuple `{:ok, _} | {:error, _}`. En kaikai la convención es la inversa: `!` no es parte de un nombre, es un operador, y **nunca lanza una excepción**. Solo desempaca el caso feliz y propaga el caso negativo a través del tipo de retorno. Si viste muchos `File.read!` en código Elixir y te pone nervioso, puedes relajarte: en kaikai esa misma sintaxis está del lado seguro.

2.4 Pattern matching como herramienta de control de flujo

En un lenguaje imperativo, decidir qué hacer según la forma de un dato suele tomar tres pasos:

1. **Comprobar la forma** con `if`, `instanceof`, `is`, `typeof`, o un campo discriminador.
2. **Acceder a los datos** que vienen con esa forma, con casts, `as`, o accesos por nombre que asumen lo del paso 1.
3. **Hacer algo** con esos datos.

kaikai junta los tres en una sola construcción: `match`.

```
match expr {
  Lit(n)    -> n
  Add(l, r) -> eval(l) + eval(r)
  Mul(l, r) -> eval(l) * eval(r)
  Neg(x)    -> -eval(x)
}
```

Cada rama es un **patrón** seguido de la expresión que ese patrón produce. `Lit(n)` no solo dice "esto fue construido con `Lit`", también declara que `n` es el `Int` que vino adentro, listo para usarse a la derecha. Comprobar, desempacar y nombrar en una sola pieza.

Los patrones se anidan. Si tienes una `Option[Result[Error, Int]]` y quieres distinguir las tres formas posibles, lo escribes así:

```
match x {
  None             -> "no había"
  Some(Err(razón))  -> "falló: " ++ razón
  Some(Ok(valor))   -> "ok: " ++ int_to_string(valor)
}
```

Los patrones también pueden traer **guardas** (condiciones adicionales que se evalúan después del match estructural) y caracteres comodín (`_`) cuando no te interesa el dato.

Lo importante: el compilador verifica **exhaustividad**. Si `Tag` tiene cuatro constructores y tu `match` cubre tres, no compila. Si agregas un quinto constructor a un tipo y existen treinta `match` en el código, los treinta se vuelven errores de compilación que te indican exactamente dónde tienes que volver. Esto convierte una refactorización temida en una tediosa pero segura.

Sin pattern matching, los lenguajes resuelven este escenario con visitor patterns, jerarquías de clases, o cadenas de `if / else if / else`. Cada una de esas soluciones funciona, pero todas pierden la conexión que tiene `match` entre el tipo del dato y la forma de decidir sobre él.

Cuando llevas algunas semanas con kaikai, el `match` se vuelve una de esas herramientas que no quieres soltar.

2.5 Funciones puras y efectos visibles

Las cuatro ideas anteriores convergen en una más grande, que es la apuesta central del lenguaje: **separar lo puro de lo que toca el mundo**, y tener al sistema de tipos cuidando esa distinción.

Una función *pura* es una función cuyo resultado depende solo de sus argumentos. Llamarla con los mismos argumentos siempre devuelve el mismo valor. No imprime. No lee del disco. No manda mensajes. No mira un reloj. No lanza un dado.

Las funciones puras son fáciles de probar, fáciles de razonar, fáciles de paralelizar, fáciles de cachear. El problema es que un programa que solo tiene funciones puras no hace nada útil: nunca habla con el mundo.

kaikai pone los efectos en el sistema de tipos. Lo viste en el capítulo 1: una función que escribe a stdout dice `: Unit / Stdout` en su firma. Una función que usa `Log` dice `: ... / Log`. Una función que puede fallar y abortar dice `: ... / Fail`. Y una función pura, que no toca el mundo, no dice nada después del `/`. Su firma es `fn f(x: Int) : Int`, sin más.

El efecto del lado derecho del `/` no es decoración. Es una restricción: el compilador no te deja llamar a una función con efectos desde un contexto donde esos efectos no estén siendo manejados. Un handler en algún lado de la pila tiene que tomarse el problema. Esto resuelve, de una vez y a nivel del lenguaje, varias incomodidades viejas:

- Las **excepciones invisibles** que Java y Python permiten porque ninguna firma las declara.
- El **manejo de cancelación** que en lenguajes con `async` / `await` se vuelve un hilo por dentro de la lógica de negocio, en vez de un mecanismo aparte.
- La **inyección de dependencias** que en lenguajes OO requiere contenedores enteros para hacer lo que un handler de efecto hace en cinco líneas.
- El **logging**, el **acceso a configuración**, el **reloj**, la **base de datos**: todo lo que tradicionalmente se cuela como dependencia escondida puede ser un efecto, declararse en el tipo, y la elige quien llama.

Si nunca has visto esto, suena demasiado ambicioso para ser cierto. Lo es y no lo es. El capítulo 12 le dedica todo el espacio que merece. Por ahora basta con saber que las firmas que ves con `/ algo` no son ruido: son información sobre lo que esa función puede hacerle a tu programa.

2.6 Una breve genealogía

kaikai no salió de la nada. Hereda decisiones de varias familias de lenguajes, y conviene saber cuáles para entender por qué se ven como se ven.

- **De ML (1973), OCaml y Haskell** vienen los tipos algebraicos, el pattern matching, la inferencia de tipos estilo Hindley-Milner, e `Option` / `Result`. Son ideas viejas, buenas, y hoy se redescubren en lenguajes "modernos" como Rust y Swift sin reconocer del todo a sus abuelos.
- **De Erlang (1986) y Elixir** vienen los procesos aislados con memoria privada, la idea de que la concurrencia se basa en pasar mensajes y no en compartir memoria, y el modelo de supervisión con `link` y `monitor`. En kaikai esos procesos se llaman fibras y actores.
- **De Elixir** además, el operador pipe `|>` que vimos arriba.
- **De Koka (Daan Leijen, Microsoft Research)** y **Effekt (Andreas Rossberg, Jonathan Brachthäuser)** vienen los efectos algebraicos con filas de efectos en el sistema de tipos. kaikai sigue de cerca a Effekt en cómo los handlers se expresan, y a Koka en algunas decisiones internas.
- **De Koka** también viene **Perceus**, el esquema de reference counting optimizado en compilación que kaikai usa para administrar memoria sin garbage collector ni borrow checker.
- **De Go** se rescata la decisión de tener un solo binario como compilador, formateador y test runner; la primacía de programas que se construyen y corren rápido; y la disciplina de mantener pocas formas en el lenguaje.
- **De Rust** kaikai aprende qué *no* hacer: los tipos suma y el pattern matching son lecciones que Rust enseña bien. El borrow checker, en cambio, kaikai prefiere evitarlo: Perceus + fibras aisladas resuelven el problema sin pedirle al programador que entienda lifetimes.

Ninguna de estas decisiones es nueva. Lo que kaikai intenta es una combinación coherente: tipos algebraicos + efectos algebraicos + Perceus + fibras BEAM, en un lenguaje que se compila rápido a código nativo y que un programador con experiencia puede leer sin un curso previo.

El resto del libro entra en detalle en cada una de esas decisiones. Si llegaste hasta acá, ya tienes el mapa.

Ejercicios

2.1. Vuelve al programa `02_fizzbuzz.kai` del capítulo 1. Identifica todas las **expresiones** que devuelven un valor. ¿Cuántas hay? ¿Hay alguna **sentencia** estricta (algo que se ejecute por su efecto sin producir nada útil) fuera de las llamadas a `println`?

2.2. En tu lenguaje habitual, escribe una versión corta del siguiente caso: una función `clasificar_edad(n)` que devuelve un string `"niño"`, `"joven"`, `"adulto"` o `"mayor"` según rangos. ¿Cuántas variables locales usaste? ¿Cuántas asignaciones? Reescríbelo después en pseudo-kaikai (no hace falta que compile) usando `if` como expresión.

2.3. Encuentra en el código de tu trabajo una función que devuelva `null` o lance una excepción para señalar "no hay resultado". Escríbe en un comentario cuál sería su firma en kaikai usando `Option` o `Result`. ¿Qué información ganas? ¿Qué información pierdes?

Capítulo 3 · Tipos básicos y expresiones

Vamos a las primitivas. Este capítulo describe los siete tipos básicos de kaikai, la forma de los literales, los operadores que los manipulan, y cómo se atan a nombres con `let`. También introduce algo que ya viste de pasada y vamos a fijar bien: **`if` y los bloques son expresiones**, no sentencias.

Si leíste el capítulo 2, lo que viene es el contenido concreto de aquellas advertencias. Si no lo leíste y vienes de un mundo imperativo, vuelve. Te ahorra fricción.

3.1 Los siete tipos primitivos

kaikai tiene exactamente siete tipos primitivos:

Tipo	Para qué sirve	Literal de ejemplo
`Int`	Enteros con signo, 64 bits	`42`, `-7`, `1_000_000`
`Real`	Reales de doble precisión, IEEE 754	`3.14`, `-0.5`, `1e10`
`Bool`	Verdadero / falso	`true`, `false`
`String`	Cadenas de texto Unicode	`"hola"`, `"α + β"`
`Char`	Un único carácter Unicode	`'a'`, `'\n'`, `'\u{2603}'`
`Unit`	El tipo con un solo valor	`()`
`Nothing`	El tipo vacío, sin habitantes	(no tiene literal)

Los primeros cinco son lo que esperarías de cualquier lenguaje. Los últimos dos merecen una explicación.

`Unit` es el tipo del valor `()`. Tiene un solo habitante. Es lo que devuelve una función que "no devuelve nada útil": el equivalente al `void` de C, pero es un tipo de verdad, con un valor de verdad, que se puede pasar como argumento o guardar en una variable. Un `println(...)` devuelve `Unit`. Un bloque que termina sin un valor explícito devuelve `Unit`.

`Nothing` es el opuesto. **Cero habitantes.** Ninguna expresión produce un `Nothing` que el programa pueda usar. ¿Para qué sirve, entonces? Para describir el tipo de retorno de cosas que **nunca terminan normalmente**: un `panic(...)` que aborta el proceso, un loop infinito, una función `forever` cuyo cuerpo no puede salir. Una función `fn loop_eterno() : Nothing` le dice al sistema de tipos que cualquier código después de llamarla es inalcanzable, y por eso una

expresión de tipo `Nothing` calza en cualquier contexto donde se espere otro tipo. Te vas a topar con `Nothing` rara vez, pero conviene tener el nombre.

3.2 Literales e interpolación de strings

Los literales numéricos admiten guion bajo como separador visual:

```
let poblacion = 19_000_000
let pi        = 3.141_592
let escala    = 1e6
```

Los `String` se escriben con comillas dobles. La sintaxis más útil del día a día es la **interpolación**:

```
let nombre = "kaikai"
let edad   = 1
println("Hola, #{nombre}. El lenguaje tiene #{edad} año.")
```

Salida:

```
Hola, kaikai. El lenguaje tiene 1 año.
```

Cualquier expresión cabe dentro de `#{...}`, no solo nombres. Su valor se convierte a `String` automáticamente:

```
let a = 7
let b = 2
println("La suma de #{a} y #{b} es #{a + b}.")
```

Para concatenar strings sin interpolación se usa `++`:

```
let saludo = "Hola, " ++ nombre
```

El operador es `++`, no `+`, para no confundirlo con suma. Es un detalle pequeño que evita un bug clásico.

Para mensajes que ocupan varias líneas, kaikai tiene **strings con triple comilla**:

```
let mensaje = """
  Esto es un mensaje
  de varias líneas, que mantiene
  la indentación relativa al cierre.
  """
```

La indentación de cada línea se mide respecto al cierre `"""`, así que puedes formatear el string visualmente sin que aparezcan espacios extra en la salida. Las secuencias de escape (`\n`, `\t`, `\u{HHHH}`, etc.) funcionan igual que en los strings normales.

Los `Char` se escriben con comilla simple: `'a'`, `'\n'`, `'\u{2603}'`. Un `Char` no es un `String` de longitud uno, son tipos distintos. Eso evita una colección de bugs típicos de lenguajes que confunden los dos.

3.3 Operadores aritméticos, lógicos y de comparación

Los operadores aritméticos:

```
+   suma              (Int o Real)
-   resta / negación  (Int o Real)
*   producto          (Int o Real)
/   división          (Int o Real)
%   módulo            (Int o Real)
```

Los cinco están **sobrecargados por tipo**: funcionan tanto con `Int` como con `Real`, y el resultado tiene el tipo de los operandos. Lo que **no** existe es coerción implícita entre `Int` y `Real`: no puedes mezclarlos en una misma expresión. Si lo necesitas, conviertes explícitamente con `int_to_real(...)` o `real_to_int(...)`.

El detalle que conviene fijar: `/` con dos `Int` ya trunca el resto. Para obtener un cociente con parte fraccionaria, los dos operandos tienen que ser `Real`.

```
let a : Int = 7
let b : Int = 2
println("a / b  = #{a / b}")     # 3: sobre Int, / trunca

let x : Real = 7.0
let y : Real = 2.0
println("x / y = #{x / y}")      # 3.5: sobre Real, hay parte
                                 #      fraccionaria
```

Si vienes de Python 3, hay un cambio de hábito. Allá `/` siempre devuelve flotante; en kaikai el tipo manda: `Int / Int` es `Int`, y si quieres parte fraccionaria conviertes explícitamente o trabajas con `Real` desde el principio.

Los operadores lógicos son palabras, no símbolos:

```
and   or   not
```

```
if x > 0 and x < 10 { ... }
if not vacio { ... }
```

A la mayoría de los lenguajes que conoces les da por usar `&&`, `||` y `!`. kaikai eligió palabras porque son más legibles, y porque los símbolos están reservados para otras cosas (`||` es flat-map en pipes, capítulo 6).

Los operadores de comparación son los habituales:

```
== != < > <= >=
```

Devuelven `Bool`. Funcionan sobre cualquier tipo que implemente los protocolos `Eq` (para `==` / `!=`) y `Ord` (para los demás). Eso lo veremos en el capítulo 9; por ahora, todos los tipos primitivos los implementan.

3.4 `let` y la propagación local de tipos

`let` ata un nombre a un valor. El tipo se infiere del lado derecho:

```
let x = 42              # x : Int
let y = 3.14            # y : Real
let nombre = "kaikai"   # nombre : String
```

Si quieres dejar el tipo explícito, lo anotas con `:`:

```
let x : Int  = 42
let y : Real = 3.14
```

La anotación no es solo decoración. Sirve para dos cosas: documentar la intención cuando el tipo no es obvio, y guiar al inferidor en los pocos casos en que la inferencia local no alcanza. La regla práctica es **anotar los argumentos y el retorno de las funciones públicas, dejar los `let` locales sin anotación**. Así el tipo viaja por las firmas y el cuerpo se queda limpio.

Una vez que un nombre se ata, no se puede volver a atar al mismo nombre en el mismo bloque. La línea siguiente daría un error:

```
let x = 42
let x = 7      # error: nombre ya definido en este ámbito
```

Lo que sí puedes hacer es atar el mismo nombre **en un ámbito interno**, lo que produce un *shadowing*: el nombre local oculta al externo:

```
let x = 42
{
  let x = 7      # OK: shadowing dentro del bloque
  println("#{x}")    # 7
}
println("#{x}")      # 42: el de afuera no cambió
```

Esto es ortogonal a la mutación. No estás "cambiando `x`": estás introduciendo un `x` distinto en un ámbito anidado, que deja de existir cuando el ámbito se cierra. El de afuera nunca fue tocado.

Para los casos donde necesitas una celda mutable de verdad, kaikai te da `var`, junto con `@nombre` y `nombre := v`. Lo viste en §2.2 y volveremos a ello en el capítulo 12 cuando hablemos de efectos.

3.5 `if` como expresión

Un `if` en kaikai produce un valor:

```
let s = if x > 0 { "positivo" } else { "no positivo" }
```

Las dos ramas tienen que producir valores del mismo tipo, y ese tipo es el del `if`. La sintaxis no usa `then`, no usa paréntesis alrededor de la condición, y cada rama es un bloque.

Si la condición tiene varias ramas, se encadenan con `else if`:

```
fn signo(n: Int) : String =
  if n < 0 { "negativo" }
  else if n == 0 { "cero" }
  else { "positivo" }
```

Cada llave abre un bloque cuyo valor es el resultado de la rama. La función entera es una sola expresión, atada a la firma con `=`. No hay `return`.

Una variante que conviene fijar: **un `if` sin `else` siempre tiene tipo `Unit`**. El compilador no completa la rama faltante con un valor sintetizado; toma la decisión más simple posible y dice "el tipo del `if` es `Unit`, y si la condición es falsa, el valor es `()`".

Por eso la forma usual de un `if` sin `else` es un cuerpo que se ejecuta por sus efectos:

```
if i <= n {
  println(label(classify(i)))
  loop(i + 1, n)
}
```

Es la forma habitual de "haz algo o sigue". Y si la rama del `then` produce un valor de otro tipo, ese valor **se descarta en silencio**: el `if` sigue siendo `Unit`:

```
if x > 0 {
  x + 1    # Int, pero se tira a la basura
}
```

Esto compila, no produce advertencia, y rara vez es lo que querías. El error suele aparecer un poco más allá, cuando intentas usar el resultado del `if`:

```
let r = if x > 0 { 42 }
println(int_to_string(r))   # error: r es Unit, no Int
```

La regla práctica es simple: si **te interesa el valor**, escribe un `if/else` exhaustivo; si **te interesa el efecto**, escribe un `if` solo, sin atarlo a nada.

3.6 Bloques y el valor de un bloque

Un bloque `{ ... }` es una **expresión** cuyo valor es el de la última expresión adentro. Las líneas anteriores se ejecutan en orden y sus valores se descartan, salvo cuando los atas con `let`.

```
fn cuadrado_mas_uno(x: Int) : Int = {
  let cuadrado = x * x
```

```
  cuadrado + 1
}
```

`cuadrado` es un binding local. La última línea, `cuadrado + 1`, es la expresión de retorno. No hay `return`; el valor del bloque es el valor de la función.

Esto se compone con todo lo demás: una rama de `if` puede ser un bloque, una rama de `match` también, el cuerpo de una lambda también. Todo el lenguaje se reduce a expresiones que devuelven valores.

3.7 Tres formas de cuerpo de función

El cuerpo de una función puede tomar **tres formas**. Lo viste en el capítulo 1 y lo fijamos acá; el cap. 6 vuelve sobre ellas con más profundidad.

Forma corta, con `=` y una sola expresión:

```
fn doble(x: Int) : Int = x * 2

fn signo(n: Int) : String =
  if n < 0 { "negativo" }
  else if n == 0 { "cero" }
  else { "positivo" }
```

Forma larga, con `{ ... }` y bindings intermedios:

```
fn cuadrado_mas_uno(x: Int) : Int = {
  let cuadrado = x * x
  cuadrado + 1
}
```

Forma multi-clause, con `case` arms cuando la función decide por la forma de sus argumentos:

```
fn signo(n: Int) : String {
  case 0           -> "cero"
  case k when k > 0 -> "positivo"
  case _           -> "negativo"
}
```

El compilador acepta las tres. La diferencia es para quien lee:

- **Forma corta** cuando la función es una expresión directa, sin pasos intermedios. Lee como una definición matemática.
- **Forma larga** cuando hay bindings intermedios o varios pasos que ayuda separar visualmente.
- **Multi-clause** cuando la función dispatcha por patrones sobre sus argumentos. Es lo natural para muchas funciones recursivas y para deciders sobre tipos suma.

No hay forma "preferida": elige la que comunica mejor la intención de esa función en particular.

Ejercicios

3.1. Escribe una función `fn fahrenheit_a_celsius(f: Real) : Real` usando solo lo de este capítulo. Verifica que `fahrenheit_a_celsius(32.0)` da `0.0` y que `fahrenheit_a_celsius(212.0)` da `100.0`. Hazlo con `=` y una sola expresión.

3.2. Escribe `fn es_par(n: Int) : Bool` y `fn paridad(n: Int) : String`, donde `paridad` devuelve `"par"` o `"impar"` según el caso. La segunda función debería usar la primera, no repetir la lógica.

3.3. Dada la función:

```
fn precio_total(unidades: Int, precio_unitario: Int) : Int = {
  let subtotal = unidades * precio_unitario
  let iva = subtotal * 19 / 100
  subtotal + iva
}
```

¿Qué imprime `precio_total(3, 100)`? Modifica la función para que el IVA se calcule correctamente como un porcentaje real (con parte fraccionaria), no entero. ¿Qué tipos cambias?

3.4. Escribe `fn maximo(a: Int, b: Int, c: Int) : Int` que devuelve el mayor de los tres argumentos, usando `if` como expresión y sin `match` ni funciones del stdlib. ¿Cuántos `else if` necesitas?

3.5. Lee la documentación del operador `++` y averigua qué pasa si intentas escribir `"hola, " ++ 42`. ¿Compila? Si no, ¿cómo lo arreglas? Razona la respuesta antes de probar.

Capítulo 4 · Tipos compuestos

Los siete primitivos del capítulo anterior te dan piezas sueltas. Para construir programas de verdad, las pegas en estructuras: agregados con campos nombrados, listas, tuplas y las dos joyas del stdlib que vas a ver más que cualquier otro tipo, `Option` y `Result`.

Este capítulo cubre todo eso. Los **sum types** (`type Tag = Foo | Bar(Int)`) que viste en el tour merecen su propio capítulo y los tratamos en el 5.

4.1 Records

Un **record** es un agregado con campos nombrados. Lo declaras con `type` y llaves:

```
type Punto = { x: Int, y: Int }

type Empleado = {
  nombre: String,
  edad: Int,
  sueldo: Int,
}
```

La coma final del último campo es opcional pero idiomática: hace que agregar un campo nuevo no toque la línea anterior, lo que mantiene los diffs de git limpios.

Para construir un valor del record, escribes el nombre del tipo seguido de las llaves con los campos:

```
let origen = Punto { x: 0, y: 0 }
let p      = Punto { x: 3, y: 4 }
let ada    = Empleado { nombre: "Ada", edad: 30, sueldo: 1500 }
```

Y para leer un campo, usas `.`:

```
println("p está en (#{p.x}, #{p.y})")
println("#{ada.nombre} tiene #{ada.edad} años")
```

Eso es lo que necesitas para el día a día. Hay tres detalles que conviene tener presentes:

- **Los records son inmutables.** No existe `p.x = 7`. Si necesitas un punto con un campo distinto, construyes uno nuevo a partir del original con la sintaxis de **spread**:

```
let p  = Punto { x: 3, y: 4 }
let p2 = Punto { ...p, x: 7 }   # Punto { x: 7, y: 4 }
```

El `...p` copia todos los campos de `p`, y los inicializadores que vienen después (acá `x: 7`) reemplazan los que se repiten. Es la misma idea del spread sobre listas que verás en §4.3, aplicada a records.

- **Los records son nominales.** `Punto { x: Int, y: Int }` y otro tipo `Posicion { x: Int, y: Int }` con los mismos campos son distintos. El compilador no los confunde aunque tengan la misma forma. Esto es deliberado: si quieres una posición, di posición.
- **El spread tiene reglas.** Solo un spread por literal, y tiene que ir primero: `Punto { x: 7, ...p }` es un error de parseo. Los inicializadores que sigan al spread tienen que ser nombrados (`x: expr`), no abreviados ni posicionales. Estas restricciones son a propósito: hacen obvio quién gana cuando hay duplicados.

4.1.1 Campos privados

Por defecto los campos de un record son públicos: cualquier módulo que importe el tipo los puede leer y mencionar al construir el record. La palabra `priv` antes del nombre del campo invierte ese default:

```
# módulo `caja`
pub type Cuenta = {
  nombre:    String,      # público por defecto
  priv saldo: Real,       # privado al módulo `caja`
}

pub fn abrir(nombre: String) : Cuenta =
  Cuenta { nombre: nombre, saldo: 0.0 }

pub fn depositar(c: Cuenta, monto: Real) : Cuenta =
  Cuenta { ...c, saldo: c.saldo + monto }

pub fn saldo_de(c: Cuenta) : Real = c.saldo
```

Desde el módulo `caja` el campo `saldo` se lee y escribe como cualquier otro. Desde fuera, no:

```
import caja

fn main() {
  let c = caja.abrir("ahorros")
  println("#{c.nombre}")          # OK, `nombre` es pública
  println("#{caja.saldo_de(c)}")  # OK, paso por el getter

  # Las dos líneas siguientes no compilan:
  # println("#{c.saldo}")          # ← field `saldo` is private to module `caja`
  # let d = caja.Cuenta {          # ← cannot construct `Cuenta` from outside ...
  #   nombre: "x",
  #   saldo: 1000.0,
  # }
}
```

La regla es estricta: ni lectura desde afuera, ni mención dentro de un literal de construcción. Si el módulo `caja` quiere que un consumidor pueda crear cuentas, expone constructores (`abrir`) y operaciones (`depositar`) que mantengan los invariantes; el campo crudo queda escondido.

Esto convierte el record en un tipo abstracto liviano: forma pública, interior bajo control del autor. Lo usaremos así en el cap. 17 para esconder el estado del actor del almacén, y en el cap. 18 para que los saldos del libro mayor no se construyan por fuera del módulo de dominio.

4.2 Acceso a campos y destructuring

Acceder con `.` está bien para uno o dos campos. Cuando necesitas varios campos a la vez, **destructuring** es más limpio:

```
fn distancia_cuadrada(a: Punto, b: Punto) : Int = {
  let Punto { x: ax, y: ay } = a
  let Punto { x: bx, y: by } = b
  let dx = ax - bx
  let dy = ay - by
  dx * dx + dy * dy
}
```

Cuando los nombres de los campos te sirven tal cual, puedes omitir el `:` y dejar solo el nombre, atando el campo a una variable del mismo nombre:

```
fn describir(p: Punto) : String = {
  let Punto { x, y } = p
  "(" ++ int_to_string(x) ++ ", " ++ int_to_string(y) ++ ")"
}
```

`let Punto { x, y } = p` es equivalente a `let Punto { x: x, y: y } = p`, solo que más conciso.

El destructuring también funciona en `match`. Esa es su forma más útil: decidir qué hacer **según los valores específicos de los campos**.

```
fn clasificar(p: Punto) : String =
  match p {
    Punto { x: 0, y: 0 } -> "origen"
    Punto { x: 0, y: _ } -> "eje Y"
    Punto { x: _, y: 0 } -> "eje X"
    Punto { x, y }       -> "punto en (#{x}, #{y})"
  }
```

El compilador verifica exhaustividad, igual que con sum types: si quitas la última rama, no compila. La exhaustividad sobre campos `Int` la cubre el patrón final `Punto { x, y }`, que calza con cualquier `Punto`. Sin esa rama, el `match` no es total.

4.3 Listas

Una lista es una secuencia inmutable y enlazada de valores del mismo tipo. El tipo se escribe con corchetes alrededor del tipo del elemento: `[Int]`, `[String]`, `[Punto]`, `[[String]]` (lista de listas).

Construir literales:

```
let primos = [2, 3, 5, 7, 11]
let vacia : [Int] = []
```

kaikai trae también **literales de rango**, que son azúcar sintáctica que produce una lista:

```
let r1 = [1..10]       # [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
let r2 = [1..10..2]     # [1, 3, 5, 7, 9]
let r3 = [10..1..-1]    # [10, 9, 8, ..., 1]
```

Para extender una lista existente, usas `...` (spread):

```
let xs = [1, 2, 3]
let ys = [0, ...xs, 99]      # [0, 1, 2, 3, 99]
```

Y para descomponerlas, los patrones de `match`:

```
fn suma(xs: [Int]) : Int =
  match xs {
    [] -> 0
    [h, ...t] -> h + suma(t)
  }
```

`[]` calza con la lista vacía. `[h, ...t]` calza con cualquier lista no vacía, atando `h` al primer elemento (la "cabeza") y `t` al resto (la "cola"). Estos dos patrones cubren todos los casos posibles, lo que hace al `match` exhaustivo.

Patrones más específicos también son legales:

```
match xs {
  []                     -> "vacía"
  [unico]                -> "uno: #{unico}"
  [primero, segundo, ...]  -> "al menos dos"
}
```

El compilador requiere que cubras todos los casos, así que si escribes solo `[]` y `[h, ...t]` te alcanza para cualquier lista; pero si quieres distinguir el caso de "exactamente un elemento", escribes `[unico]` antes del catch-all.

Una convención del lenguaje: si la cola **te interesa**, le das un nombre, `[h, ...t]`, y después usas `t`. Si **no te interesa**, escribes `...` solo, sin nombre: `[h, ...]`. Es la diferencia entre "tomo la cabeza, el resto lo guardo para después" y "tomo la cabeza, el resto lo descarto". Un nombre inventado que no se usa es ruido visual; la forma corta comunica la intención sin pedirte un nombre que no aporta.

Para acceder por índice, el stdlib expone `list.nth`:

```
let primero = list.nth(xs, 0)   # Option[Int]
let tercero = list.nth(xs, 2)   # Option[Int]
```

Fíjate en el tipo de retorno: **`Option[a]`**, no `a`. Una lista enlazada no garantiza que un índice exista: si pides el elemento número 99 de una lista de tres, no hay valor que devolver. El tipo te obliga a considerarlo. Esto es coherente con `Option` y `Result` que vimos en §4.5: kaikai prefiere encerrar la posibilidad de fallar en el tipo antes que abortar en runtime.

Hay dos cosas más que conviene saber sobre el acceso por índice. Una es que es **`O(n)`**: las listas son enlazadas, no indexadas; recorrer hasta la posición `i` cuesta `i` pasos. Para acceso aleatorio rápido kaikai tiene `Array[T]`, que veremos en el capítulo 13.

La otra es que la sintaxis `xs[i]` que algunos lenguajes usan para listas, en kaikai está reservada para `Array[T]`. Si la escribes sobre una lista te la rechaza el typer. La razón es la misma de antes: la sintaxis `xs[i]` sugiere acceso barato y con resultado garantizado, lo que sería mentir sobre una lista enlazada.

Para la mayoría del código, tampoco vas a querer indexar a mano. Recursión sobre `[h, ...t]` o las funciones de orden superior del capítulo 6 (`map`, `filter`, `reduce`) son la forma natural de procesar listas.

Las listas son **inmutables**. No hay `xs[0] = 99`. Si necesitas una lista modificada, construyes una nueva.

4.4 Strings, no listas de chars

Vale la pena detenerse un momento en algo que muchos lenguajes mezclan: en kaikai, **un `String` no es una lista de `Char`**. Son tipos distintos:

```
let s : String = "hola"
let cs : [Char] = ['h', 'o', 'l', 'a']
```

`s` y `cs` no son intercambiables. No puedes escribir `s[0]` esperando un `Char`, y no puedes pasar un `String` donde se espera `[Char]`.

¿Por qué? Porque en Unicode no hay una correspondencia simple entre "carácter" e "índice". Un emoji puede ocupar varios code points; una letra acentuada puede tener una o dos representaciones; un grafema puede saltar bytes y code points arbitrariamente. Tratar a un string como lista de chars te obliga a tomar una decisión sobre qué cuenta como "carácter", y todas las decisiones son malas para algún caso.

kaikai opta por hacer al `String` **opaco**: las operaciones que tienen sentido se exponen en el módulo `string` del stdlib (`length`, `starts_with`, `ends_with`, `trim`, `repeat`, `join`) y no se confunde texto con listas. `string.length(s)` cuenta **bytes**, no caracteres ni grafemas. Para `"á"` devuelve 2 (porque "á" en UTF-8 ocupa dos bytes); para `"☃"` devuelve 3. Es una elección consciente: la representación interna del string es UTF-8, y kaikai prefiere una respuesta predecible y barata sobre una respuesta filosóficamente correcta pero costosa. Si necesitas contar grafemas o caracteres lógicos, vas a usar funciones del módulo distintas que decodifican Unicode con su sutileza.

Para concatenar, ya lo viste en el capítulo 3, usas `++`:

```
let saludo = "hola, " ++ nombre
```

Y para interpolar, `#{...}` dentro de un literal `"..."`.

4.5 `Option` y `Result`: el día a día

Ya viste estos dos en el capítulo 2:

```
type Option[a] = None | Some(a)
type Result[e, a] = Err(e) | Ok(a)
```

Acá los miramos en uso. Los dos son tipos suma genéricos del stdlib que vas a usar **constantemente**. La idea, recordando: `Option` representa "puede no haber valor"; `Result`, "puede no haber valor o haber un error".

Una función que busca el primer elemento par de una lista:

```
fn primer_par(xs: [Int]) : Option[Int] =
  match xs {
    [] -> None
    [h, ...t] -> if h % 2 == 0 { Some(h) } else { primer_par(t) }
  }
```

Y quien la llama tiene que considerar los dos casos explícitamente:

```
match primer_par(xs) {
  Some(n) -> println("encontré: #{n}")
  None    -> println("no había pares")
}
```

Una función que parsea una edad de un string puede fallar de dos formas distintas, y eso es exactamente para lo que sirve `Result`:

```
type ErrorEdad = NoNumerica | FueraDeRango

fn parsear_edad(s: String) : Result[ErrorEdad, Int] =
  match string_to_int(s) {
    None -> Err(NoNumerica)
    Some(n) ->
      if n < 0 or n > 130 { Err(FueraDeRango) }
      else { Ok(n) }
  }
```

`Result[ErrorEdad, Int]` se lee como "un `Int` o un error de tipo `ErrorEdad`". El error está en el primer parámetro y el valor exitoso en el segundo, al revés de la convención que usan algunos lenguajes; kaikai sigue la tradición de Haskell en este punto.

Tres patrones que vas a ver mucho:

- **Encadenar una falla con `!`.** Si tienes una expresión que devuelve `Result[E, A]` y quieres "si falla, propaga el error; si no, sigue con el valor", escribes `expr!`. Esto lo viste en §2.3.
- **Funciones de orden superior**: `option.map`, `option.and_then`, `result.map_err`. Las cubrimos en el capítulo 6.
- **Convertir entre los dos**: `option.ok_or(error)` toma un `Option[a]` y un error de tipo `e` y devuelve un `Result[e, a]`. Útil cuando la pérdida de información del `None` ya no te alcanza.

`Option` y `Result` son tipos suma como cualquier otro. Los hemos separado del capítulo 5 porque su rol en el diseño cotidiano es central: los vas a usar antes de empezar a declarar tus propios tipos suma.

4.6 Tuplas

Una tupla es un agregado **posicional**: como un record, pero sin nombres de campo. La sintaxis es paréntesis con elementos:

```
let punto2d = (3, 4)
let trio    = ("Ada", 30, true)
```

Su tipo se escribe igual: `(Int, Int)`, `(String, Int, Bool)`.

kaikai admite tuplas de **arity 2 a 4**. No hay tuplas de un elemento: un paréntesis solo, `(e)`, es agrupación, no tupla. Y `(a, b, c, d, e)` es un error de parseo.

¿Por qué la cota? Porque las tuplas largas son ilegibles. Si estás llegando a 5 elementos, casi siempre lo que querías era un record con campos nombrados.

Para descomponer una tupla, destructuring:

```
fn divmod(a: Int, b: Int) : (Int, Int) = (a / b, a % b)

fn main() {
  let (cociente, resto) = divmod(17, 5)
  println("17/5 = #{cociente}, resto #{resto}")
}
```

Por dentro, las tuplas son azúcar sobre tres records del stdlib: `Pair[A, B]` para arity 2, `Triple[A, B, C]` para 3, `Quad[A, B, C, D]` para 4. La declaración

```
let p = (1, 2)
```

es exactamente equivalente a

```
let p = Pair { fst: 1, snd: 2 }
```

Esto es útil saberlo cuando ves un tipo `Pair[Int, String]` en una firma de stdlib: ahora sabes que es lo mismo que `(Int, String)`, y que puedes destructurarlo con `let (a, b) = p`.

¿Tupla o record?

Cuando dudes, **usa un record**. Las tuplas son cómodas para casos donde los nombres no aportan: el resultado de `divmod`, donde "el primer valor es el cociente y el segundo el resto" es información que el lector ya tiene del nombre de la función. O cuando el agregado vive un solo paso:

```
xs |> map((e) => (e.nombre, e.edad))
```

Pero apenas el mismo agregado aparece más de una vez, o cruza una frontera de módulo, o su forma es algo que el lector no tiene cómo deducir, conviene un record. Un `Empleado` es mucho más fácil de leer que un `(String, Int, Bool, String, Int)`.

Ejercicios

4.1. Define un record `Libro` con campos `titulo`, `autor` y `paginas`. Escribe una función `fn corto(b: Libro) : Bool` que devuelva `true` si el libro tiene menos de 200 páginas. Construye dos libros y prueba la función.

4.2. Escribe una función `fn maximo(xs: [Int]) : Option[Int]` que devuelva el mayor elemento de la lista, o `None` si la lista está vacía. Hazlo con recursión y `match`.

4.3. Reescribe `fn parsear_edad` de §4.5 para que distinga entre **tres** errores: no numérico, edad negativa, edad > 130. Usa un tipo suma con tres constructores y un `Result`.

4.4. Define una función `fn separar(xs: [Int]) : ([Int], [Int])` que devuelva una tupla con los pares y los impares de `xs`, en su orden original. ¿Por qué tupla y no dos parámetros separados? ¿Y por qué tupla y no un record?

4.5. Dado un `[Punto]` (con `Punto = { x: Int, y: Int }`), escribe `fn centro(ps: [Punto]) : Option[Punto]` que devuelva el promedio de coordenadas, o `None` si la lista está vacía. Pista: vas a necesitar dos pasadas o un acumulador, y `int_to_real` para promediar, pero nota que el resultado final tiene que volver a ser `Punto` con campos `Int`, así que también necesitas `real_to_int` (o conformarte con la parte entera).

4.6. En un papel, sin escribir código, dibuja qué pasa en memoria cuando ejecutas estas tres líneas:

```
let xs = [1, 2, 3]
let ys = [0, ...xs]
let zs = [99, ...xs]
```

¿Cuántas listas se crearon? ¿Cuántos elementos se copiaron? ¿Hay alguna celda que comparten `xs`, `ys`, `zs`? La respuesta te ayuda a entender por qué la inmutabilidad no es cara cuando las estructuras son enlazadas.

Capítulo 5 · Sum types, uniones y `match`

Este es el capítulo donde, si vienes de un lenguaje imperativo, cambias la forma de modelar datos. Los tipos suma y el `match` que los acompaña no son una construcción exótica: son la herramienta que reemplaza a la mitad de las jerarquías de clases, los `enum` con flags, los `instanceof` y los visitor patterns que probablemente has venido escribiendo. Una vez que los manejas, no quieres soltarlos.

El capítulo va de menos a más. Empezamos por los sum types básicos, pasamos por la recursión en tipos, vemos `match` con todas sus formas, llegamos a las uniones de tipos existentes (una idea poco común en lenguajes mainstream) y cerramos con un evaluador completo de expresiones aritméticas con errores tipados.

5.1 Tipos suma con `|`

Un **tipo suma** declara que un valor puede ser uno de varios constructores distintos. La sintaxis es directa:

```
type Color = Rojo | Verde | Azul
```

`Color` es un tipo. `Rojo`, `Verde`, `Azul` son sus tres **constructores**. Un valor de tipo `Color` es exactamente uno de los tres. No hay un cuarto valor escondido, no hay `null`, no hay "color desconocido". El tipo lo dice todo.

Si vienes de un lenguaje con `enum`, esto se le parece, pero con dos diferencias: los constructores **pueden cargar datos**, y el compilador **verifica que los uses todos** cuando decides según el constructor.

Empecemos por la primera. Un constructor puede recibir parámetros posicionales, igual que un record con campos numerados:

```
type Forma
  = Circulo(Real)
  | Rectangulo(Real, Real)
  | Triangulo(Real, Real, Real)
```

`Circulo(2.0)` es un valor de tipo `Forma` con un dato real adentro. `Rectangulo(3.0, 4.0)` es otro valor de `Forma` con dos datos. `Triangulo(3.0, 4.0, 5.0)` es un tercero. Los tres caen bajo el mismo tipo

`Forma`, pero el compilador sabe cuál es cuál y nos obliga a manejarlos por separado cuando los consumimos.

```
fn area(f: Forma) : Real =
  match f {
    Circulo(r)        -> 3.14159 * r * r
    Rectangulo(w, h)  -> w * h
    Triangulo(a, b, c) -> {
      let s = (a + b + c) / 2.0
      s
    }
  }
```

`match` es una expresión que decide según el constructor. Cada rama es un patrón seguido de `->` y la expresión que esa rama produce. Los patrones desempacan los datos al mismo tiempo: en `Circulo(r)`, la `r` es la variable que ata el `Real` que venía adentro del constructor. No hay un `cast` aparte ni un acceso por índice; el patrón hace el trabajo.

Tres detalles que vas a usar siempre:

- **El compilador verifica exhaustividad.** Si quitas la rama `Triangulo(...)` y el compilador sabe que `f : Forma` puede ser un triángulo, no compila. Esto te salva de bugs sutiles cuando agregas un constructor nuevo: todos los `match` que no lo cubren se vuelven errores que apuntan exactamente a dónde falta atención.
- **Los constructores son nombres como cualquier otro.** Cuando declaras `type Color = Rojo | Verde | Azul`, kaikai crea cuatro símbolos: `Color` es el tipo, y `Rojo`, `Verde`, `Azul` son tres tipos a la vez (cada uno con un solo habitante, equivalente al de un valor). En el capítulo 9 vamos a ver cómo este detalle te deja extender un sum type con protocolos definidos para sus constructores individualmente.
- **El operador `|` siempre significa unión de tipos.** Lo vamos a ver más en §5.5: `Color = Rojo | Verde | Azul`, `ErrorEval = ErrorAritmetico | ErrorAmbiente` y `Resultado = Ok(Int) | Err(String)` son la misma construcción. La diferencia entre "declarar tipos nuevos" y "componer tipos existentes" la decide el compilador inspeccionando si los nombres del lado derecho ya están declarados.

5.2 Constructores con y sin payload

Lo viste arriba pero conviene fijarlo. Un constructor puede:

- **No cargar datos** (`Rojo`, `Verde`, `Azul`). Es un valor único del tipo, sin parámetros.
- **Cargar uno o más datos posicionales** (`Circulo(Real)`, `Rectangulo(Real, Real)`). El constructor se aplica como una función a los datos para producir el valor.

No hay límite en cuántos datos puede cargar un constructor: la gramática los acepta todos, pero por ergonomía, si pasas de tres o cuatro datos posicionales conviene declarar un record y ponerlo en el payload. La diferencia entre

```
type Evento
  = Login(String, String, Int, Bool)
  | Logout(String, Int)
```

y

```
type DatosLogin = { usuario: String, ip: String, timestamp: Int, exitoso: Bool }
type DatosLogout = { usuario: String, timestamp: Int }
type Evento = Login(DatosLogin) | Logout(DatosLogout)
```

es que el segundo se lee mejor en cualquier lugar donde construyas o destructures un evento. La regla práctica: si los datos se nombran solos por su posición (un punto es `(Real, Real)`), quedan posicionales. Si los datos requieren que el lector recuerde el orden, conviene un record.

5.3 Recursión en tipos

Acá empieza lo poderoso. Un constructor de un tipo suma puede mencionar al tipo en sus campos. Eso te da árboles, listas, grafos pequeños y la representación de cualquier estructura con anidamiento.

Un árbol binario:

```
type Arbol
  = Hoja
  | Nodo(Int, Arbol, Arbol)
```

`Arbol` es `Hoja` (sin datos) o `Nodo` con un entero y dos subárboles. Los subárboles son del mismo tipo `Arbol`, así que pueden ser a su vez `Hoja` o `Nodo`, recursivamente, hasta la profundidad que quieras.

Una función sobre un árbol también es recursiva: el caso base es `Hoja`, el caso recursivo desciende por los hijos.

```
fn altura(t: Arbol) : Int =
  match t {
    Hoja             -> 0
    Nodo(_, izq, der) -> 1 + max_int(altura(izq), altura(der))
  }
```

El patrón `Nodo(_, izq, der)` ignora el dato del nodo (no nos interesa para calcular altura) y ata los dos subárboles a las variables `izq` y `der`. Después se recurre sobre cada uno y se toma el máximo más uno.

Lo mismo aplica a un AST de expresiones aritméticas:

```
type Expr
  = Lit(Int)
  | Suma(Expr, Expr)
  | Mul(Expr, Expr)
```

```
  | Neg(Expr)

fn eval(e: Expr) : Int =
  match e {
    Lit(n)      -> n
    Suma(a, b)  -> eval(a) + eval(b)
    Mul(a, b)   -> eval(a) * eval(b)
    Neg(x)      -> -eval(x)
  }
```

Construir una expresión es transparente: `Mul(Neg(Suma(Lit(2), Lit(3))), Lit(4))` es la representación literal de `-(2 + 3) * 4`, que evaluada da `-20`. El árbol y el código que lo recorre se escriben prácticamente solos, una vez que sabes mirarlo así.

Esto reemplaza a las jerarquías de clases con visitor patterns en lenguajes OO. La diferencia clave: en kaikai, agregar un nodo nuevo a `Expr` se hace en una línea, y todos los `match` sobre `Expr` en el código se vuelven errores de compilación que apuntan a los lugares que necesitan atención. En un visitor pattern, agregas un método nuevo en cada subclase y el compilador no te ayuda a no olvidar ninguno.

5.4 `match`: patrones, guardas, exhaustividad

Ya viste `match` en acción. Acá lo formalizamos.

Un `match` toma una **expresión escrutinio** (lo que se está inspeccionando) y una serie de **arms**, cada uno con un **patrón** seguido de `->` y la expresión que esa rama produce. La rama cuyo patrón calza con el valor del escrutinio es la que se ejecuta; su valor es el valor del `match`.

Los patrones que kaikai acepta:

- **Literales**: `0`, `"hola"`, `true`. Calzan con el valor exacto.
- **Constructores**: `Some(x)`, `Lit(n)`, `Suma(a, b)`. Calzan con el constructor y atan los datos a las variables.
- **Records**: `Punto { x, y }`, `Punto { x: 0, y: _ }`.
- **Listas**: `[]`, `[h, ...t]`, `[unico]`, `[primero, segundo, ...]`.
- **Wildcard**: `_`. Calza con cualquier cosa, no ata nada.
- **Variable**: cualquier identificador no declarado. Calza con cualquier cosa y ata el valor a esa variable.

Los patrones se anidan: `Some(Punto { x, y })` calza con un `Some` que contiene un `Punto`, y desempaca `x` e `y` en una sola pasada.

Guardas

Un patrón puede ir seguido de `if` y una condición, una **guarda**, que se evalúa después del match estructural. Si la guarda es falsa, la rama no se considera y se sigue con la siguiente:

```
fn signo(n: Int) : String =
  match n {
    0          -> "cero"
    k if k > 0  -> "positivo"
    _          -> "negativo"
  }
```

El patrón `k if k > 0` calza con cualquier entero, lo ata a `k`, y entonces evalúa `k > 0`. Si es verdad, ejecuta la rama; si no, sigue. La rama final con `_` no tiene guarda y calza con todo lo restante: los enteros que no son cero ni positivos.

Las guardas son convenientes pero no participan en el verificación de exhaustividad: el compilador no puede saber que `k > 0` y `k < 0` se complementan, así que necesita un patrón final sin guarda que cubra el caso "todo lo demás". Si lo omites, no compila.

Exhaustividad

El compilador verifica que **todos los habitantes posibles** del tipo del escrutinio estén cubiertos. Si tu `Expr` tiene cuatro constructores y tu `match` cubre tres, no compila, y el mensaje no se queda en "falta algo": te dice qué falta, qué cubriste y cómo arreglarlo:

```
error: non-exhaustive match on Expr: missing Neg
 --> evaluador.kai:12:3
   |
12 |   match e {
   |   ^
 = note: missing variant: `Neg`
 = note: covered: Lit, Suma, Mul
 = help: add an arm `Neg -> ...` or a wildcard `_ -> ...`
```

El wildcard `_` cubre todo lo que las ramas anteriores no hayan cubierto, así que un `match` con un `_` final es trivialmente exhaustivo. Pero usar `_` como rama final en lugar de enumerar los casos es una forma de **silenciar** la ayuda del compilador: cuando agregues un constructor nuevo a `Expr`, los `match` con `_` final lo absorberán sin quejarse, y vas a perder el aviso.

La regla práctica: usa `_` solo cuando de verdad no te interesa distinguir el resto. Si los casos son tres y los tres te importan, escribe los tres.

5.5 Uniones de tipos existentes

Hasta ahora todos los `|` que hemos visto tenían **nombres nuevos** del lado derecho (`Rojo`, `Verde`, `Lit`, `Circulo`) que kaikai auto-declara como constructores. Pero el operador `|` no exige nombres nuevos. Si los nombres del lado derecho **ya están declarados como tipos**, kaikai construye una **unión** que puede llevar cualquier valor de esos tipos.

Esto es la herramienta clave para componer errores entre capas:

```
type ErrorIdentidad = CuentaNoExiste | KycVencido | Congelada
type ErrorAuth      = SaldoInsuficiente | LimiteDiario
```

```
type ErrorConsulta = ErrorIdentidad | ErrorAuth
```

`ErrorConsulta` es la unión de los dos tipos previos. Un valor de `ErrorConsulta` es **cualquier** valor de `ErrorIdentidad` o **cualquier** valor de `ErrorAuth`. No hay envoltorio nuevo: `CuentaNoExiste` ya era un valor válido, y ahora es también un valor válido de `ErrorConsulta`.

Ese paso se llama **upcast implícito**: una variable tipada `ErrorIdentidad` calza donde se espera `ErrorConsulta`, sin conversión:

```
let id_err : ErrorIdentidad = CuentaNoExiste
let qb_err : ErrorConsulta  = id_err     # OK, sin ceremonia
```

Esto es lo que en otros lenguajes te obliga a escribir constructores wrapper (`QBIdentity(IdentityError)`), implementaciones de `From` o conversiones manuales con `map_err`. En kaikai no hay ninguna de esas tres cosas: el compilador sabe que `ErrorIdentidad` es un componente de `ErrorConsulta` y reescribe el upcast por ti.

Pattern matching sobre uniones

Un `match` sobre un valor de unión tiene dos sabores que kaikai te deja **mezclar libremente**.

El primero, ya conocido, es enumerar todos los constructores individualmente:

```
fn describir(e: ErrorConsulta) : String =
  match e {
    CuentaNoExiste     -> "id: cuenta no existe"
    KycVencido         -> "id: KYC vencido"
    Congelada          -> "id: cuenta congelada"
    SaldoInsuficiente  -> "auth: saldo insuficiente"
    LimiteDiario       -> "auth: límite diario"
  }
```

Esto funciona, pero es tedioso para uniones grandes. Y peor, si un componente crece (agregas `RegulatoryHold` a `ErrorAuth`), el `match` se vuelve un error de compilación en cinco lugares en vez de uno.

El segundo sabor es el **patrón de narrowing** `bind : ComponentType`, que calza cuando el valor pertenece al componente nombrado y lo amarra bajo ese tipo más estrecho:

```
fn describir(e: ErrorConsulta) : String =
  match e {
    ie : ErrorIdentidad -> "id: " ++ id_str(ie)
    ae : ErrorAuth      -> "auth: " ++ auth_str(ae)
  }
```

Cada rama delega en una función que conoce ese componente específico. Si `ErrorAuth` crece, solo cambia `auth_str`. El `match` sobre `ErrorConsulta` ni se entera.

Las dos formas también se mezclan en un mismo `match` cuando quieres extraer un caso específico y delegar el resto:

```
match e {
  CuentaNoExiste      -> "id: específicamente, cuenta missing"
  ie : ErrorIdentidad -> "id: " ++ id_str(ie)       # KycVencido, Congelada
  ae : ErrorAuth      -> "auth: " ++ auth_str(ae)
}
```

El compilador trata las ramas de narrowing como si cubrieran todos los constructores de su componente, así que la exhaustividad cierra correctamente.

Una limitación deliberada: el upcast no encadena

Hay una regla de kaikai que es importante conocer. El upcast implícito **vale solo un paso**. Imagina tres tipos encadenados, cada uno conteniendo al anterior:

```
type ErrorIdentidad = CuentaNoExiste | KycVencido
type ErrorAuth      = SaldoInsuficiente | LimiteDiario
type ErrorConsulta  = ErrorIdentidad | ErrorAuth
type ErrorRuteo     = NoEncontrado | ServidorCaido
type ErrorApp       = ErrorConsulta | ErrorRuteo

fn manejar_app(e: ErrorApp) : String = "..."
```

Y un valor del tipo más interno:

```
let id : ErrorIdentidad = CuentaNoExiste
manejar_app(id)       # ERROR: ErrorIdentidad no es componente directo
                      # de ErrorApp.
```

Aunque lógicamente todo `ErrorIdentidad` es un `ErrorApp` (vía `ErrorConsulta`), el compilador no busca esa cadena. Para pasar de `ErrorIdentidad` a `ErrorApp` tienes que escribir el salto intermedio explícito:

```
let q : ErrorConsulta = id        # un paso: ErrorIdentidad ⟶ ErrorConsulta
manejar_app(q)                    # otro paso: ErrorConsulta ⟶ ErrorApp
```

Esto es deliberado. Subtipado encadenado vuelve la inferencia de tipos frágil y los mensajes de error confusos. La regla "un paso" mantiene a kaikai con un sistema de tipos predecible y con mensajes que apuntan al lugar correcto.

5.6 Errores como uniones, sin wrappers

Las uniones son tan útiles para errores que vale la pena detenerse en el patrón. Cuando una función puede fallar de varias maneras y esas maneras se descomponen en categorías claras, el patrón natural es:

1. Cada categoría es un sum type pequeño.
2. El error compuesto es la unión de las categorías.
3. Cada función devuelve `Result[ErrorCompuesto, T]`.
4. El operador `!` propaga el error de cualquier capa hasta el `Result` compuesto, vía el upcast implícito.

```
fn check_identidad(req: Req) : Result[ErrorIdentidad, Cuenta] = ...
fn check_auth(c: Cuenta) : Result[ErrorAuth, Aprobado] = ...

fn consultar_saldo(req: Req) : Result[ErrorConsulta, Saldo] = {
  let cuenta = check_identidad(req)!     # ErrorIdentidad <: ErrorConsulta
  let app    = check_auth(cuenta)!       # ErrorAuth      <: ErrorConsulta
  Ok(cargar_saldo(app))
}
```

Cada `!` desempaca el `Ok` y propaga el `Err` con el upcast correcto al `Result[ErrorConsulta, _]` que devuelve la función externa. Cero wrappers, cero `map_err`, cero `From`. La firma de `consultar_saldo` documenta exactamente qué errores puede emitir, y el compilador se asegura de que todos estén cubiertos cuando alguien la consume.

Lo que `!` hace por dentro

Vale la pena mirar el operador con un poco más de cuidado, porque la primera vez parece mágico. **`!` es un `return` temprano disfrazado de operador.** La línea

```
let cuenta = check_identidad(req)!
```

es exactamente equivalente a:

```
let cuenta = match check_identidad(req) {
  Ok(x)  -> x                 # desempaca y sigue
  Err(e) -> return Err(e)     # sale de consultar_saldo con ese error
}
```

Sobre `Option[A]` el desugar es análogo: `Some(x)` desempaca, `None` provoca `return None`.

Tres cosas que conviene fijar:

- **No es un `panic` ni una excepción.** El programa no aborta; la función actual termina normalmente devolviendo el `Err` o el `None` a quien la llamó. El control sale **un solo nivel**, no más allá.
- **Solo funciona si la función actual devuelve un tipo compatible.** Si tu función devuelve `Int`, no puedes usar `!` sobre un `Result` adentro: el `return Err(e)` no tendría dónde aterrizar. El compilador te lo dice claro.
- **El upcast sucede en el `return`.** Cuando `check_identidad(req)` devuelve `Result[ErrorIdentidad, _]` pero `consultar_saldo` declara `Result[ErrorConsulta, _]`, el `return Err(e)` aplica el upcast `ErrorIdentidad <: ErrorConsulta` al pasar. Por eso `!` y las uniones de errores se llevan tan bien: cada nivel de la cascada absorbe el error de la capa inferior sin escribir conversiones.

Si vienes de Rust, este es exactamente el operador `?`. Si vienes de Swift, es lo que `try` con `throws` hace. Si vienes de Haskell, es la `do`-notation sobre `Either` colapsada en un solo símbolo.

Compáralo con la versión imperativa típica:

```
# Python: nada en el tipo de retorno te dice qué puede fallar
def consultar_saldo(req):
```

```
    cuenta = check_identidad(req)   # puede tirar AccountNotFound, KycExpired...
    app    = check_auth(cuenta)     # puede tirar InsufficientBalance...
    return cargar_saldo(app)
```

El que llama a `consultar_saldo` en Python tiene que leer el código de las funciones internas (o la doc, si la hay) para saber qué excepciones esperar. En kaikai, la firma se lo dice.

5.7 Caso de estudio: evaluador con errores tipados

Cerramos el capítulo con un caso integrador. Vamos a construir un evaluador de expresiones aritméticas con tres categorías de error:

- **Aritméticos**: división por cero, raíz de un número negativo.
- **De ambiente**: variable no definida.

Los dos forman una unión, `ErrorEval`, y el evaluador devuelve `Result[ErrorEval, Real]`. El código completo está en `ejemplos/cap05/05_evaluador.kai`; acá vamos paso a paso por las partes interesantes.

El AST

```
type Expr
  = Lit(Real)
  | Var(String)
  | Suma(Expr, Expr)
  | Mul(Expr, Expr)
  | Div(Expr, Expr)
  | Raiz(Expr)
```

Cinco constructores, dos de ellos recursivos. `Var(String)` introduce una novedad respecto al evaluador del cap. 1: ahora las expresiones pueden referenciar variables, así que el evaluador necesita un **ambiente** que asocie nombres a valores.

Los errores

```
type ErrorAritmetico = DivCero | RaizNegativa(Real)
type ErrorAmbiente   = NoDefinida(String)
type ErrorEval       = ErrorAritmetico | ErrorAmbiente
```

Tres tipos: dos categorías y la unión. Cuatro constructores en total, pero distribuidos en categorías que tienen sentido por sí mismas. `RaizNegativa(Real)` carga el valor que se intentó pasar a la raíz: es información útil para el mensaje de error final. `NoDefinida(String)` carga el nombre de la variable que faltaba.

El ambiente

```
type Env = [(String, Real)]

fn lookup(env: Env, nombre: String) : Result[ErrorEval, Real] {
```

```
  case [], _                    -> Err(NoDefinida(nombre))
  case [(k, v), ...], n when k == n   -> Ok(v)
  case [_, ...resto], n             -> lookup(resto, n)
}
```

`Env` es un alias para una lista de pares: recorrido lineal, suficiente para un evaluador de juguete. `lookup` está escrito en la **forma multi-clause** del cap. 6: cada `case` lista un patrón por argumento separado por coma (acá `env` y `nombre`), con un `when` opcional para guardas. Tres casos:

- Lista vacía: la variable no estaba; devolvemos un error.
- Cabeza con la clave que buscamos: éxito.
- Cualquier otra cabeza: seguimos en la cola.

Fíjate que el `Err` es del tipo `ErrorEval`, no `ErrorAmbiente`, pero `NoDefinida(nombre)` se construye como `ErrorAmbiente` y el upcast implícito lo promueve a `ErrorEval` en el sitio del retorno, sin conversión explícita.

`eval` (a continuación) usa la forma con `match` envoltorio porque dispatcha sobre un tipo suma con muchos constructores y la forma `match e { … }` se lee mejor cuando el discrimen es sobre un solo argumento de tipo bien marcado. Las dos formas conviven en el mismo archivo sin tensión.

El evaluador

```
fn eval(env: Env, e: Expr) : Result[ErrorEval, Real] =
  match e {
    Lit(n)      -> Ok(n)
    Var(nombre) -> lookup(env, nombre)
    Suma(a, b)  -> {
      let va = eval(env, a)!
      let vb = eval(env, b)!
      Ok(va + vb)
    }
    Mul(a, b)   -> {
      let va = eval(env, a)!
      let vb = eval(env, b)!
      Ok(va * vb)
    }
    Div(a, b)   -> {
      let va = eval(env, a)!
      let vb = eval(env, b)!
      if vb == 0.0 { Err(DivCero) } else { Ok(va / vb) }
    }
    Raiz(x)     -> {
      let v = eval(env, x)!
      if v < 0.0 { Err(RaizNegativa(v)) } else { Ok(raiz(v)) }
    }
  }
```

Cada caso de `match` corresponde a un constructor de `Expr`. Los casos recursivos usan `!` para evaluar los subárboles y propagar cualquier error que aparezca; los casos que pueden

fallar localmente (división por cero, raíz negativa) construyen un `Err` del tipo apropiado y dejan que el upcast los promueva.

El operador `!` aparece varias veces y vale la pena leerlo con calma. `eval(env, a)!` significa: si `eval(env, a)` devuelve `Ok(v)`, ata `va` a `v`; si devuelve `Err(e)`, **sale inmediatamente** de la función actual devolviendo ese `Err`. En este caso particular, el `Err` que se propaga es `Result[ErrorEval, _]`, y como `eval` devuelve exactamente ese tipo, el upcast es trivial. Pero la misma forma funciona cuando los tipos de error de las llamadas internas son componentes distintos de la unión.

Imprimiendo el resultado

```
fn describir(e: ErrorEval) : String =
  match e {
    DivCero          -> "división por cero"
    RaizNegativa(v)   -> "raíz de número negativo (" ++ real_to_string(v) ++ ")"
    NoDefinida(nombre) -> "variable no definida: " ++ nombre
  }

fn imprimir(r: Result[ErrorEval, Real]) : Unit =
  match r {
    Ok(v)  -> println("ok: " ++ real_to_string(v))
    Err(e) -> println("error: " ++ describir(e))
  }
```

`describir` consume un `ErrorEval` enumerando sus tres constructores. Acá el `match` enumera porque queremos mensajes específicos por constructor; en otros casos, cuando la lógica para cada categoría difiere, usaríamos narrowing.

El programa principal arma el ambiente, evalúa varias expresiones y las imprime:

```
fn main() {
  let env : Env = [("x", 9.0), ("y", 4.0)]

  imprimir(eval(env, Suma(Var("x"), Var("y"))))    # ok: 13
  imprimir(eval(env, Raiz(Var("x"))))              # ok: 3
  imprimir(eval(env, Div(Lit(10.0), Lit(0.0))))    # error: división por cero
  imprimir(eval(env, Raiz(Lit(0.0 - 1.0))))        # error: raíz de negativo
  imprimir(eval(env, Var("z")))                    # error: variable no definida: z
}
```

Lo bonito de este caso: cada error está documentado en el tipo del evaluador. Si más adelante quieres agregar un **ErrorTipo** (por ejemplo, intentar sumar dos cosas que no son números), declaras la categoría nueva, la incluyes en `ErrorEval`, y el compilador te lleva a todos los lugares que necesitan adaptarse.

Ejercicios

5.1. Define `type Maybe[a] = Just(a) | Nothing` (que es otra forma de `Option`) y escribe `fn or_else[a](m: Maybe[a], default: a) : a` que devuelva el valor si es `Just`, o el default si es `Nothing`.

Nota: el nombre `Nothing` también es un tipo primitivo de kaikai (el bottom type del cap. 3); usa `Vacio` u otro nombre si te molesta el shadowing.

5.2. Extiende el evaluador del §5.7 para que admita restas: agrega `Resta(Expr, Expr)` al AST, agrégalo al `match` de `eval`, y verifica que `2 - 3` evalúe a `-1`. ¿Cuántas líneas tuviste que cambiar? ¿Cuántos lugares tocó el compilador?

5.3. Define un árbol de decisión binario:

```
type Decision[a]
  = Hoja(a)
  | Pregunta(String, Decision[a], Decision[a])
```

donde una `Pregunta` tiene un texto, una rama `sí` y una rama `no`. Escribe `fn aplicar[a](d: Decision[a], respuestas: [(String, Bool)]) : Option[a]` que recorra el árbol siguiendo las respuestas a cada pregunta, y devuelva `Some` con la decisión final si llega a una hoja, o `None` si en alguna pregunta no hay respuesta.

5.4. Tomas el ejemplo de uniones con `ErrorIdentidad` y `ErrorAuth`. Agrega un tercer componente `ErrorRedAdministrativa` con dos constructores. Reescribe el `match` con narrowing para que el código quede igual de corto. Luego agrega un constructor nuevo a `ErrorAuth` (digamos `Bloqueado`): ¿qué pasa con tu código? ¿En cuántos lugares interviene el compilador?

5.5. Crea un sum type para representar un mensaje de chat: `type Mensaje = Texto(String) | Imagen(String, Int, Int) | Audio(String, Int)` (ruta, dimensiones o duración). Escribe una función `fn descripcion(m: Mensaje) : String` que devuelva una descripción humana, y una función `fn pesado(m: Mensaje) : Bool` que devuelva `true` si el mensaje es de imagen o audio. La segunda función debería usar narrowing en el `match`, no enumerar.

5.6. En un papel, dibuja el árbol de evaluación que construye el evaluador del §5.7 cuando procesa la expresión `Mul(Suma(Var("x"), Lit(2.0)), Var("y"))` con el ambiente `[("x", 9.0), ("y", 4.0)]`. ¿Cuántas llamadas a `eval` hace? ¿Cuántas a `lookup`? ¿En qué orden? Esto te ayuda a entender por qué el operador `!` corta la ejecución apenas algo falla: tres llamadas en cascada, una sola línea por nivel.

Capítulo 6 · Funciones y pipelines

Hasta acá fuiste viendo funciones a medida que las necesitábamos: las primeras en el tour, las del cap. 3 con sus dos formas de cuerpo, las recursivas del cap. 5 que descomponen tipos suma. Este capítulo pone todo junto y agrega lo que faltaba: cómo declarar funciones con cuidado, cómo escribir lambdas, qué son las funciones de orden superior, y los cuatro operadores pipe que kaikai usa para encadenar transformaciones.

Vamos también a ver con detalle algo que kaikai promete y que muy pocos lenguajes garantizan en serio: la **eliminación de llamadas en posición de cola**. Esa garantía es lo que te deja escribir loops sin `for` ni `while` y dormir tranquilo.

6.1 Declaración

Una función se declara con `fn`, los parámetros tipados, el tipo de retorno, y un cuerpo separado por `=`:

```
fn doble(x: Int) : Int = x * 2
```

El cuerpo puede tomar **tres formas**. La primera es la que acabas de ver: una sola expresión.

La segunda es un bloque `{ … }` con `let`s intermedios y la expresión final implícita:

```
fn cuadrado_mas_uno(x: Int) : Int = {
  let cuadrado = x * x
  cuadrado + 1
}
```

La tercera son **arms con patrones**, donde la función decide qué hacer según la forma de sus argumentos. Cada arm empieza con `case`, va seguido de un patrón, una flecha `->` y la expresión que ese caso produce:

```
fn signo(n: Int) : String {
  case 0          -> "cero"
  case k when k > 0 -> "positivo"
  case _          -> "negativo"
}
```

Es exactamente equivalente a:

```
fn signo(n: Int) : String =
  match n {
    0          -> "cero"
    k if k > 0 -> "positivo"
    _          -> "negativo"
  }
```

solo que sin el `match` envoltorio. Cuando la función tiene **varios parámetros**, los patrones se listan separados por comas, uno por argumento:

```
fn divide(a: Real, b: Real) : Result[Error, Real] {
  case _, 0.0 -> Err(DivCero)
  case a, b   -> Ok(a / b)
}
```

Una regla del lenguaje: dentro del bloque `{ ... }` de una función, **o todo son `case` arms o todo son sentencias**. Mezclar es un error de parseo. Si necesitas setup antes de discriminar, envuelve un `match` en la forma corta o extrae un helper.

La regla práctica para elegir entre las tres:

- **Cuerpo corto con `=`** cuando la función es una expresión directa, sin pasos intermedios.
- **Bloque con `{ ... }`** cuando hay `let` s intermedios o varios pasos visualmente separados.
- **Multi-clause con `case`** cuando la función decide principalmente por la forma de sus argumentos. Es la forma natural para muchas funciones recursivas y para dispatchers sobre tipos suma.

Las tres son aceptadas por el compilador; la elección es para quien lee.

Algunas notas que conviene fijar:

- **Anotaciones de parámetros: obligatorias.** kaikai infiere tipos en bindings locales (`let x = 5`), pero no en firmas de función. Cada parámetro tiene que decir su tipo.
- **Tipo de retorno: obligatorio en funciones públicas y recomendado siempre.** El compilador puede inferirlo en funciones locales, pero la firma documenta el contrato y hace que los errores de tipo se reporten en el lugar correcto. Anota.
- **Funciones con efecto: el efecto va después del `/`.** Una función que escribe a stdout es `: Unit / Stdout`, y el cap. 12 entra en eso con calma. Por ahora basta con saber que si tu función llama a `println`, su firma lo declara.

```
fn anunciar(mensaje: String) : Unit / Stdout =
  println("[ANUNCIO] " ++ mensaje)
```

`main` es la única función donde el tipo de retorno es opcional. Si lo omites, kaikai asume `Unit`.

6.2 Lambdas

Una **lambda** es una función anónima, una expresión que se evalúa a un valor de tipo función. kaikai te da tres formas para escribirlas:

```
# Forma 1: flecha con un argumento
let cuadrado = (x) => x * x

# Forma 2: flecha con varios argumentos
let suma = (a, b) => a + b

# Forma 3: placeholder, lambda implícita unaria
xs |> list.filter(. > 0)
```

Las dos primeras son intercambiables; la elección es de estilo. La tercera es **azúcar** que solo aplica cuando el contexto **espera una función**: el segundo argumento de `list.filter`, por ejemplo, es de tipo `(Int) -> Bool`, y el compilador convierte `. > 0` en `(n) => n > 0`.

Las reglas del placeholder son cuatro:

- `.` solo funciona en posición donde se espera una función. Fuera de ahí es un error de compilación.
- Funciones unarias solamente. Para dos o más argumentos, flecha explícita.
- Múltiples ocurrencias del mismo `.` se refieren al **mismo valor**. Por ejemplo, `xs |> list.map(. * .)` eleva al cuadrado.
- El acceso a campo funciona: `personas |> list.map(.nombre)` proyecta el campo `nombre` de cada elemento.

¿Cuándo usar cuál? Mi sugerencia:

- **Placeholder** cuando la lambda es trivial y está dentro de un pipe. `xs |> list.filter(. > 0)` se lee de un vistazo.
- **Flecha** cuando la lambda tiene varios pasos, o cuando el argumento aparece en lugares no obvios. `(p) => p.edad + bonus * 2` no gana nada con placeholder.
- **Función con nombre** cuando la lambda se usa más de una vez, o cuando el nombre documenta la intención. Si la misma lambda aparece en tres lugares, dale un nombre.

Las lambdas son **valores de primera clase**: las atas a `let`, las pasas como argumento, las devuelves de funciones, las guardas en records. Eso es lo que hace que las funciones de orden superior sean naturales.

6.3 Funciones de orden superior

Una **función de orden superior** es una que recibe o devuelve otra función. Esa es toda la definición. Lo interesante no es el nombre: es lo que te deja hacer.

El caso más simple es una función que aplica otra dos veces:

```
fn dos_veces[a](f: (a) -> a, x: a) : a = f(f(x))
```

`f` es el primer parámetro y su tipo es `(a) -> a`: cualquier función que vaya de `a` a `a`. `dos_veces(mas_uno, 5)` calcula `mas_uno(mas_uno(5))` = `7`. La función es **genérica** sobre el tipo `a`:

funciona con `Int`, con `String`, con cualquier tipo, mientras `f` mantenga el mismo tipo de entrada que de salida. Las anotaciones `[a]` después del nombre declaran el parámetro de tipo.

Un caso más interesante es una función que **devuelve** otra función: un *closure*.

```
fn sumar(n: Int) : (Int) -> Int = (x) => x + n
```

`sumar(10)` devuelve una nueva función que suma 10 a su argumento. El truco es que la lambda **captura** `n` del contexto donde se creó. Una vez devuelta, esa función tiene una copia de `n` adentro:

```
let mas_diez = sumar(10)
mas_diez(7)       # 17
mas_diez(100)     # 110
```

Y la composición clásica:

```
fn componer[a, b, c](f: (b) -> c, g: (a) -> b) : (a) -> c =
  (x) => f(g(x))
```

`componer(f, g)` es la función que primero aplica `g` y después `f`. Tres parámetros de tipo (`a`, `b`, `c`) porque las funciones encadenadas tocan tres tipos distintos en general.

Las funciones de orden superior son la herramienta principal para abstraer **lo que hay que hacer**. En vez de escribir `para cada elemento, hacer X` y `para cada elemento, hacer Y`, escribes `para cada elemento, hacer F`, donde `F` es un parámetro. Así nacen `list.map`, `list.filter`, `list.fold`, todos los caballos de batalla de la programación funcional.

6.4 Pipes: `|>`, `|`, `||`

kaikai trae cuatro operadores para encadenar. Los cuatro son distintos y cada uno comunica una intención específica.

`|>`: apply

`xs |> f` es exactamente `f(xs)`. El operador toma el lado izquierdo y lo pone como **primer argumento** del llamado de la derecha. Si `f` tiene varios argumentos, los demás van en los paréntesis:

```
xs |> list.sum                 # ≡ list.sum(xs)
xs |> list.filter(es_par)      # ≡ list.filter(xs, es_par)
xs |> list.map((n) => n * 2)   # ≡ list.map(xs, (n) => n * 2)
```

Pero hay un detalle útil: a veces el valor del pipe **no quiere ir como primer argumento**. Por ejemplo, una función de división donde lo que estás canalizando es el divisor, no el dividendo. kaikai te deja indicar la posición exacta con un guión bajo `_`:

```
fn divide(a: Int, b: Int) : Int = a / b

100 |> divide(_, 4)       # ≡ divide(100, 4) = 25
100 |> divide(1000, _)    # ≡ divide(1000, 100) = 10
```

El `_` es el **hueco** donde aterriza el lado izquierdo. Sin guión bajo, el lado izquierdo va al primer argumento: es la forma corta de `f(_, a, b)`. Con guión bajo, va donde lo pongas. Esto te deja escribir pipelines naturales aun cuando las funciones del stdlib no estén diseñadas con el "argumento principal" en la primera posición.

```
fn entre(low: Int, x: Int, high: Int) : Bool =
  x >= low and x <= high

50 |> entre(0, _, 100)    # ≡ entre(0, 50, 100) = true
```

Es lo que viene de Elixir y F#: un pipe **general** con control de posición opcional.

`|`: map

`xs | f` es exactamente `list.map(xs, f)`. Es un operador **específico** para mapear listas:

```
let dobles = xs | (n) => n * 2    # ≡ list.map(xs, (n) => n * 2)
```

¿Por qué tener `|` si `|>` ya cubre el caso? Porque `xs | f` se lee como "xs procesado con f", que es exactamente lo que un map es. La forma más corta hace que los pipelines de transformación de listas se lean como secuencias declarativas:

```
let total = xs | doble | restar_uno |> list.sum
```

vs.

```
let total = xs |> list.map(doble) |> list.map(restar_uno) |> list.sum
```

Las dos son equivalentes. La primera tiene **menos ruido sintáctico** porque las dos transformaciones unitarias usan `|`. La regla práctica: `|` para mapear, `|>` cuando el lado derecho es un llamado más complejo (con argumentos extra, sumas, lookups).

`||`: flat-map

`xs || f` es `list.flat_map(xs, f)`. Cada elemento `x` produce una **lista** `f(x)`, y el resultado es la concatenación de todas las listas:

```
fn vecinos(n: Int) : [Int] = [n - 1, n, n + 1]

[10, 20, 30] || vecinos
# = [9, 10, 11, 19, 20, 21, 29, 30, 31]
```

`||` desazucara directamente a `list.flat_map(xs, f)`, igual que `|` desazucara a `list.map(xs, f)`. Las tres formas siguientes son equivalentes:

```
let extendido = [10, 20, 30] || vecinos              # azúcar
let extendido = list.flat_map([10, 20, 30], vecinos)    # llamada directa
let extendido = [10, 20, 30] | vecinos |> list.concat    # map + concat
```

Las tres producen `[9, 10, 11, 19, 20, 21, 29, 30, 31]`. La primera dice "expande cada elemento". La segunda es la llamada con su nombre real. La tercera muestra cómo flat-map se define: mapear y aplanar. Usa `||` cuando quieras hacer evidente la operación dentro de un pipeline, y la llamada directa cuando no estés en pipeline.

`|?` : filter

`xs |? p` es exactamente `list.filter(xs, p)`. El operador **filtra** la lista quedándose con los elementos para los que el predicado es verdadero:

```
fn es_par(n: Int) : Bool = n % 2 == 0

[1, 2, 3, 4, 5, 6] |? es_par      # ≡ list.filter(xs, es_par) = [2, 4, 6]
```

El predicado es cualquier expresión que el contexto admita como `(a) -> Bool`: un nombre de función, una flecha, una lambda en bloque:

```
xs |? es_par                  # nombre
xs |? (n) => n > 3              # flecha
xs |? { n -> n % 3 == 0 }         # bloque
```

`|?` cierra la familia de pipes específicos para listas: `|` es map, `||` es flat-map, `|?` es filter. Las tres operaciones canónicas de transformación de secuencias tienen su propio operador.

Todo junto

Los cuatro operadores se mezclan libremente:

```
let total =
  pedidos
  |? esta_pendiente
  | aplicar_descuento
  | monto_de
  |> list.sum
```

Cada paso del pipeline hace una sola cosa. La firma del resultado se entiende de izquierda a derecha. No hay variables temporales, no hay anidamiento de paréntesis. Esto es la principal razón de que kaikai tenga cuatro operadores y no uno.

6.5 Trailing lambdas y otros azúcares

kaikai trae varios azúcares sintácticos que vas a ver en código real, sobre todo cuando uses funciones de orden superior. Los principales:

Trailing lambdas

Cuando una función toma una lambda como **último** argumento, puedes sacarla de los paréntesis y ponerla en llaves al final, con la sintaxis `{ param -> body }`:

```
list.map(xs) { n -> n * 2 }
list.filter(xs) { n -> n > 0 }
```

Es equivalente a `list.map(xs, (n) => n * 2)`. Las dos formas son aceptadas; la trailing es más amigable cuando el cuerpo de la lambda es largo.

Cuando estás dentro de un pipeline con `|`, el bloque-lambda es aún más compacto. `|` ya espera una función como segundo argumento, así que puedes escribir el cuerpo directamente:

```
xs | { n -> n * 2 }          # equivalente a xs | ((n) => n * 2)
xs | { n -> n * n + 1 }
```

Esto se lee casi como prosa: "los xs, cada uno transformado a n por dos". La forma vale tanto para `|` como para `||` (que también esperan una función), y se mezcla con el resto del pipeline sin ruido.

Doble trailing lambda

Si los **dos** últimos argumentos son lambdas, los dos van en llaves:

```
fn while(cond: () -> Bool, body: () -> Unit) : Unit / e = ...

while { @i < 10 } { i := @i + 1 }
```

Esto le da a kaikai control de flujo definible por el usuario. `while` es una función ordinaria del stdlib; solo parece un keyword.

Block como lambda

`{ x -> body }` es una forma alternativa para una lambda que encaja mejor cuando el cuerpo abarca varias líneas:

```
xs | { n ->
  let cuadrado = n * n
  cuadrado + 1
}
```

Es lo mismo que `(n) => { let cuadrado = n * n; cuadrado + 1 }` pero menos cargado visualmente.

Ninguno de estos azúcares introduce semántica nueva. Son formas alternativas de escribir lambdas que el compilador desazucara antes de la inferencia de tipos. Úsalos cuando la lectura mejore, ignóralos cuando no.

6.6 Recursión y TCO obligatoria

Una promesa importante de kaikai: **toda llamada recursiva en posición de cola se compila a un loop**. No consume stack. No hay riesgo de stack overflow por una recursión larga.

Veamos qué significa "posición de cola". Una llamada está en **posición de cola** si es lo último que la función va a hacer antes de devolver. Compara estas dos versiones de `suma`:

```
# NO en posición de cola: la llamada deja pendiente `h + ...`
fn suma_naive(xs: [Int]) : Int {
  case []         -> 0
  case [h, ...t]  -> h + suma_naive(t)
}
```

Acá, después de que `suma_naive(t)` devuelve, todavía hay que sumar `h`. La llamada **no** es lo último; queda una operación pendiente. Cada llamada consume un frame del stack.

```
# En posición de cola: la última cosa que hace cada rama es
# **solo** la llamada, sin operación pendiente.
fn suma_tco_loop(xs: [Int], acc: Int) : Int {
  case [], a         -> a
  case [h, ...t], a  -> suma_tco_loop(t, a + h)
}

fn suma(xs: [Int]) : Int = suma_tco_loop(xs, 0)
```

Acá, en cada rama recursiva, `suma_tco_loop(t, acc + h)` es **lo último**. La suma `acc + h` se evalúa primero, se pasa como argumento, y entonces la llamada ocurre. Cuando la llamada devuelve, la función actual también devuelve inmediatamente: no queda nada por hacer. El compilador detecta este patrón y lo compila a un loop, sin frame nuevo.

```
# Esto funciona sin reventar el stack:
let muchos = [1..100_000]
suma(muchos)        # 5_000_050_000
```

La técnica del **acumulador** que ves en `suma_tco_loop` es la forma estándar de convertir una recursión naive en una recursiva en cola: agregas un parámetro extra que va llevando el resultado parcial, y al terminar lo devuelves.

¿Por qué importa esto en serio?

- **Sin TCO obligatoria, no podrías reemplazar `for` y `while`.** Con stack limitado, una iteración sobre un millón de elementos te dejaría sin stack. Solo con TCO garantizada puedes dar el paso a programar con recursión.
- **Es una garantía del lenguaje, no una optimización oportunista.** Algunos lenguajes optimizan TCO cuando se acuerdan; kaikai te lo promete. Si una llamada recursiva está en cola, el compilador la convierte. Punto.
- **El compilador te avisa si crees que escribiste TCO pero no.** Hay un flag para verificar esto, así no te enteras por sorpresa cuando tu programa muere en producción.

En la práctica, la mayoría de las funciones recursivas que escribas para procesar listas o árboles van a ser de la forma `match xs { [] -> base; [h, ...t] -> recursión }`. La versión naive (con operación pendiente) es la primera que escribes; la versión con acumulador es la que dejas. Para operaciones más complejas, las funciones de orden superior (`list.fold`, `list.reduce`) ya están escritas con TCO y son casi siempre lo que querías.

6.7 Caso de estudio: pipeline de transformación

Cerramos con un caso integrador. Tienes una lista de pedidos de una tienda, con un id, un monto y un estado. Quieres calcular el total a cobrar **considerando solo los pedidos pendientes** y **aplicando un 10% de descuento a los montos altos**:

```
type Estado = Pendiente | Pagado | Cancelado

type Pedido = {
  id: Int,
  monto: Int,
  estado: Estado,
}

fn esta_pendiente(p: Pedido) : Bool =
  match p.estado {
    Pendiente -> true
    Pagado    -> false
    Cancelado -> false
  }

fn aplicar_descuento(p: Pedido) : Pedido =
  if p.monto >= 1000 {
    Pedido { ...p, monto: p.monto - p.monto / 10 }
  } else {
    p
  }

fn monto_de(p: Pedido) : Int = p.monto
```

Cada función chica hace **una sola cosa**: una decide si un pedido está pendiente, otra le aplica descuento si corresponde, otra extrae el monto. Ninguna sabe del pipeline; todas son útiles por separado.

Y el pipeline las compone:

```
let total =
  pedidos
  |? esta_pendiente
  | aplicar_descuento
  | monto_de
  |> list.sum
```

Cinco líneas, cuatro pasos. Filtra los pendientes, aplica descuento a cada uno, extrae el monto de cada uno, suma todo. Si el cliente pide después "agreguemos un cargo fijo de 100 a los pedidos > 5000", agregas una función chica más y la metes en el pipeline:

```
fn cargo_si_grande(p: Pedido) : Pedido =
  if p.monto > 5000 {
    Pedido { ...p, monto: p.monto + 100 }
  } else {
    p
  }

let total =
  pedidos
  |? esta_pendiente
  | aplicar_descuento
  | cargo_si_grande
  | monto_de
  |> list.sum
```

Una línea más, cero acoplamiento. Ese es el punto del estilo funcional: **funciones chicas que se componen en pipelines de lectura lineal**. La complejidad vive en cada función individual; la composición es trivial.

Compáralo con la versión imperativa típica:

```
total = 0
for p in pedidos:
    if p.estado != "pendiente":
        continue
    monto = p.monto
    if monto >= 1000:
        monto -= monto // 10
    if monto > 5000:
        monto += 100
    total += monto
```

Funciona, pero la lógica del cálculo está enredada con la mecánica del bucle. Cambiar el orden de las transformaciones, agregar un paso, sacar un paso: todas operaciones que en el pipeline son una línea, en la versión imperativa son un refactor.

Si vienes de Java, JavaScript, C# o Kotlin, has visto algo parecido con sus respectivos stream/iterator APIs. La diferencia es que en kaikai el pipeline es una construcción sintáctica del lenguaje, no una API encima del lenguaje. No necesitas envolver la lista en un objeto especial, no hay métodos `.collect()` ni terminadores; los pipes son tan ordinarios como `+` o `*`.

Ejercicios

6.1. Escribe una función `fn aplicar_n(f: (Int) -> Int, x: Int, n: Int) : Int` que aplique `f` a `x` exactamente `n` veces. Por ejemplo, `aplicar_n(mas_uno, 5, 3)` debería ser `8`. Hazlo con recursión en posición de cola.

6.2. Define `fn componer3[a, b, c, d](f: (c) -> d, g: (b) -> c, h: (a) -> b) : (a) -> d` que componga tres funciones. Comprueba con `componer3(cuadrado, mas_uno, doble)(2)`.

6.3. Reescribe el pipeline del §6.7 usando solo `|>` (sin `|` ni `||`). ¿Cuántos caracteres más tiene? ¿Cómo cambia la legibilidad?

6.4. Define un sum type `type Operacion = Sumar(Int) | Multiplicar(Int) | Negar`. Escribe una función `fn aplicar(op: Operacion, x: Int) : Int` que ejecute la operación sobre `x`, y una función `fn ejecutar_todas(ops: [Operacion], x: Int) : Int` que ejecute una secuencia de operaciones empezando con `x`. Pista: `list.fold` con `aplicar` invertido es un buen camino.

6.5. Tienes una lista de strings con números: `["10", "abc", "20", "", "30"]`. Quieres la suma de los que **sí son números válidos**, ignorando los demás. Construye un pipeline usando `list.flat_map` (o `||`) y la función `string_to_int : String -> Option[Int]`. Pista: una función `Option[a] -> [a]` te ayuda: si es `Some(x)` devuelve `[x]`, si es `None` devuelve `[]`. Ese paso es el flat-map natural.

Capítulo 7 · Pruebas, propiedades y benchmarks

Hasta acá estuviste escribiendo funciones y mirando la salida. Es un ciclo razonable mientras tu programa cabe en la cabeza, pero no escala. Apenas tu código pasa de unas decenas de líneas (apenas hay más de tres funciones que se llaman entre sí), dejas de poder verificar a ojo que cada cambio mantiene el comportamiento.

Para eso están las pruebas. kaikai trae **tres construcciones top-level** dedicadas: `test`, `check` y `bench`. Las tres viven en el mismo archivo del código que prueban, las tres se ejecutan vía el driver `kai`, y las tres se ignoran cuando construyes un binario para producción.

Este capítulo recorre las tres, explica cuándo usar cuál, y cierra con un caso de estudio: un mini-evaluador con tests contractuales, propiedades verificadas, y benchmarks que miden cuánto cuesta cada operación.

7.1 `test "..." { ... }` y `assert`

La forma más simple es un test con un nombre y un cuerpo:

```
fn cuadrado(n: Int) : Int = n * n

test "cuadrado de cero" {
  assert cuadrado(0) == 0
}

test "casos pequeños" {
  assert cuadrado(3) == 9
  assert cuadrado(7) == 49
}
```

`test` es un **bloque top-level**: convive con `fn` en el mismo archivo, no se anida en otra función. Su nombre es un literal de string que el runner reporta tal cual. Adentro vas escribiendo aserciones con `assert`: una expresión `Bool` que debe ser `true`. Si todas las aserciones del bloque pasan, el test pasa. Si **una** falla, el test falla y el runner sigue con los siguientes.

El nombre del test debería decir **qué se está probando**, no cómo. "cuadrado de cero" es bueno; "test 1" no.

`assert` también acepta un mensaje opcional con coma:

```
test "rangos válidos" {
  let n = clasificar(42)
  assert n > 0, "se esperaba positivo, no #{n}"
}
```

El mensaje aparece cuando la aserción falla. Es útil cuando la expresión que evaluaste no transmite por sí sola qué fue lo inesperado.

Lo que el runner imprime

```
$ kai test ejemplos/cap07/01_test_basico.kai
  ok   caso base
  ok   casos pequeños
  ok   caso significativo

3/3 tests passed
```

Si un test falla, la salida cambia para mostrarlo:

```
$ kai test ejemplos/cap07/02_assert_falla.kai
  ok   doble preserva positivos
  FAIL test roto: el assert va a fallar : assertion failed
  ok   este test sigue corriendo

2/3 tests passed
```

Tres detalles que vale recordar:

- **Los tests se ejecutan en orden de declaración.** Tu archivo los lista de arriba a abajo y el runner los corre en ese orden. No hay paralelismo dentro de un mismo archivo.
- **Un test que falla no detiene a los demás.** El runner sigue con los tests siguientes y reporta el conteo final.
- **Los bloques `test` no terminan en el binario de producción.** `kai run` y `kai build` los descartan. Solo se compilan y se ejecutan bajo `kai test`.

7.2 `kai test` y el ciclo corto de retroalimentación

El comando es directo:

```
$ kai test mi_archivo.kai
```

Compila el archivo en modo `--test` (que activa los bloques `test`), produce un binario, lo ejecuta, y reporta. El ciclo edición → prueba toma uno o dos segundos en archivos chicos.

Si llamas a `kai test` sin nombre de archivo, no hay un descubrimiento automático estilo `pytest` o `cargo test`. Eso es deliberado (kaikai todavía no tiene un layout de proyecto estándar), pero conviene saber el flujo: tú apuntas al archivo, el runner corre todo lo que ese archivo y sus imports declaren con `test`.

Hay tres recomendaciones prácticas que vas a internalizar a las pocas semanas:

- **Tests al lado del código, en el mismo archivo.** No los separes en `tests/` o en archivos paralelos. Cuando tocas una función, los tests de esa función están al ojo.
- **Un test por aspecto, no por línea.** Si tu función tiene un caso base, casos pequeños y un caso límite, escribe tres `test`. Si dentro de "casos pequeños" hay tres ejemplos, agrégalos como tres `assert` en el mismo bloque.
- **Nombres descriptivos.** El nombre va a aparecer en la salida del runner cada vez que corras los tests. `"validar email rechaza espacios"` se entiende; `"test_3"` no.

7.3 `check "..."`: propiedades

Los tests que viste hasta acá comprueban **casos fijos**: "para esta entrada, espero esta salida". Es lo que en otros lenguajes se conoce como "example-based testing". Es lo más común, pero tiene un límite obvio: solo prueba lo que escribes.

Las **propiedades** invierten la cosa. En vez de "para `cuadrado(7)` espero `49`", escribes "para todo `n` entero, `cuadrado(n)` debe ser `>= 0`". El runner genera valores de `n` al azar y verifica la propiedad sobre cada uno. Si encuentra un contraejemplo, te lo muestra; si pasa cien iteraciones sin falla, considera la propiedad probada.

```
fn doble(n: Int) : Int = n * 2

check "doble es par" with n: Int {
  doble(n) % 2 == 0
}

check "suma conmutativa" with a: Int, b: Int {
  a + b == b + a
}

check "reverse de reverse" with xs: [Int] {
  list.reverse(list.reverse(xs)) == xs
}
```

`check "..." with name: Type { body }` declara una propiedad. La cláusula `with` lista los parámetros que el runner va a generar al azar; el `body` es una expresión `Bool` que tiene que ser `true`.

```
$ kai check ejemplos/cap07/03_check_propiedades.kai
  doble es par: 100 iter, OK
  suma conmutativa: 100 iter, OK
  suma asociativa: 100 iter, OK
  reverse de reverse: 100 iter, OK

4/4 checks passed
```

Cien iteraciones por propiedad es lo predeterminado; cada iteración genera valores nuevos. Para `Int`, el rango por defecto es `[-50, 50]`. Para `[Int]`, listas pequeñas. Para records y sum types, el generador estructura recursivamente sus componentes.

Cuando una propiedad falla

```
check "todos los Int son positivos" with n: Int {
  n > 0
}
```

```
$ kai check propiedad_falsa.kai
  todos los Int son positivos: counterexample at iter 1: n=-32

0/1 checks passed
```

El runner te entrega el **contraejemplo exacto** (`n = -32`) en la primera iteración que falló. Eso te dice tres cosas:

- La propiedad es falsa para algún valor de `n`.
- El valor concreto.
- En qué iteración falló (útil para reproducir con la misma semilla si quieres depurar).

A diferencia de un `test` con un caso fijo, donde el nombre del test es lo que diagnostica el fallo, un `check` te entrega el caso de prueba **junto con** el reporte. No tienes que escribirlo: lo tienes.

Cuándo escribir un `check`

Las propiedades son útiles cuando puedes **enunciar una verdad universal** sobre tu código. Algunos ejemplos comunes:

- **Inversas**: `decode(encode(x)) == x`, `parse(format(x)) == x`.
- **Idempotencia**: `normalize(normalize(s)) == normalize(s)`.
- **Invariantes algebraicas**: conmutatividad, asociatividad, identidad.
- **Conservación**: el `length` de la salida es el mismo que el de la entrada, la suma se mantiene, etc.
- **Monotonía**: si `a < b`, entonces `f(a) < f(b)`.

Si lo que quieres comprobar es "para la entrada 7, sale 14", eso es un `test`. Si es "para cualquier entrada, lo que sale duplica el valor", es un `check`.

7.4 `bench "..." { ... }`: medir, no adivinar

La tercera construcción es para **performance**. `bench` toma un bloque y mide cuánto tarda en ejecutarse, repetido muchas veces para sacar promedio:

```
fn fib(n: Int) : Int =
  if n < 2 { n } else { fib(n - 1) + fib(n - 2) }

bench "aritmética: 2 + 3 * 4" {
```

```
  2 + 3 * 4
}

bench "fib(10)" {
  fib(10)
}

bench "fib(15)" {
  fib(15)
}
```

```
$ kai bench ejemplos/cap07/04_bench_basico.kai
  aritmética: 2 + 3 * 4: 1000 iter / 7 ns/iter
  fib(10): recursión sin memo: 1000 iter / 92 ns/iter
  fib(15): el costo crece exponencial: 1000 iter / 1063 ns/iter
  list.sum [1..100]: 1000 iter / 4305 ns/iter

4 benches
```

Cada bench corre 1000 iteraciones (configurable con `KAI_BENCH_ITERS`) y reporta nanosegundos por iteración.

Lo importante de los benchmarks no es el número absoluto (depende de la máquina y de qué más esté corriendo), sino la **comparación**. Cuando refactorizas una función crítica, corres el bench antes y después. Cuando tu pipeline empieza a sentirse lento, comparas las versiones de funciones candidatas. La regla:

> ***Optimizar sin medir es adivinar.***

Si tu código no anda lento, no lo benchees. Si anda lento, no lo optimices sin medirlo primero. Los `bench` son la herramienta que cierra ese ciclo.

Tres consejos prácticos:

- **El cuerpo del `bench` es lo que se mide.** Si tu setup es caro y no quieres incluirlo, hazlo afuera del bloque y deja solo la operación a medir adentro.
- **kaikai no descarta llamadas "puras" sin efecto observable.** `bench "fib(10)" { fib(10) }` realmente computa `fib(10)` cada iteración. En otros lenguajes con optimizaciones más agresivas hay que usar trucos para evitar dead-code elimination; aquí no.
- **El número absoluto es indicativo, no autoridad.** Para comparar, mide siempre en la misma máquina, en la misma sesión, sin otras cargas pesadas corriendo en paralelo.

7.5 Cuándo usar cuál

Ya tienes las tres herramientas. La decisión se reduce a una pregunta simple:

Pregunta	Herramienta
¿Para esta entrada concreta, sale lo que espero?	`test`
¿Para **toda** entrada, vale esta invariante?	`check`
¿Cuánto cuesta esta operación?	`bench`

Las tres se complementan. Un proyecto serio va a tener las tres en el mismo archivo: tests para los casos contractuales (los del cliente, los de borde, los que históricamente fallaron), checks para las invariantes algebraicas que el código preserva, y benchmarks para las pocas funciones críticas donde la performance importa.

Una nota sobre el orden de escritura. La secuencia natural suele ser:

1. **Empieza con un `test`**: el caso concreto del feature que estás desarrollando. Es la prueba más fácil de escribir y la más fácil de mirar cuando algo falla.
2. **Agrega `test`s** para casos límite a medida que aparecen.
3. **Pasa a `check`s** cuando ves un patrón en los tests: "todos estos casos están comprobando la misma invariante, pero con datos distintos". Eso es señal de que una propiedad debería capturar la regla general.
4. **Agrega un `bench`** cuando empieces a notar lentitud, o antes de un refactor de optimización para tener una línea base.

No al revés. Empezar con un `check` cuando todavía no sabes qué propiedades vas a preservar te lleva a propiedades vagas que pasan por accidente. Empezar con un `bench` antes de que la performance importe es optimización prematura. Tests primero.

7.6 Caso de estudio: pruebas para un mini-evaluador

Cerramos con un ejemplo integrador: un evaluador chico de expresiones aritméticas con manejo de errores, probado con las tres herramientas. El código completo está en `ejemplos/cap07/05_evaluador_pruebas.kai`; acá lo recorremos por partes.

El AST y el evaluador

```
type Expr
  = Lit(Int)
  | Suma(Expr, Expr)
  | Mul(Expr, Expr)
  | Div(Expr, Expr)

type ErrorEval = DivCero(Int)

fn eval(e: Expr) : Result[ErrorEval, Int] =
  match e {
    Lit(n)     -> Ok(n)
    Suma(a, b) -> {
      let va = eval(a)!
      let vb = eval(b)!
      Ok(va + vb)
    }
    Mul(a, b)  -> { ... }
```

```
    Div(a, b)  -> {
      let va = eval(a)!
      let vb = eval(b)!
      if vb == 0 { Err(DivCero(va)) } else { Ok(va / vb) }
    }
  }
```

Es un primo más chico del evaluador del cap. 5: cuatro constructores, una sola categoría de error (división por cero). Suficiente para mostrar el flujo.

Tests para los casos contractuales

```
test "literal" {
  assert debe_dar(Lit(42), 42)
}

test "expresión combinada: 2 + 3 * 4 = 14" {
  assert debe_dar(Suma(Lit(2), Mul(Lit(3), Lit(4))), 14)
}

test "división por cero da error" {
  assert debe_fallar(Div(Lit(10), Lit(0)))
}
```

Tres tests que documentan tres comportamientos. `debe_dar` y `debe_fallar` son helpers que envuelven el `match` sobre `Result` y devuelven `Bool`, para que el `assert` se mantenga legible.

```
4/4 tests passed
```

Checks para las invariantes

```
check "Lit(n) evalúa a n" with n: Int {
  debe_dar(Lit(n), n)
}

check "Suma(a, b) == Suma(b, a)" with a: Int, b: Int {
  debe_dar(Suma(Lit(a), Lit(b)), a + b) and
    debe_dar(Suma(Lit(b), Lit(a)), a + b)
}
```

Dos propiedades. La primera dice que un literal evalúa a sí mismo: una invariante trivial pero importante. Si fallara, algo está muy mal en el evaluador. La segunda comprueba que la suma es conmutativa **a través del evaluador**, no solo a nivel de aritmética entera.

```
2/2 checks passed
```

Cien iteraciones por cada una con valores generados al azar. Ninguna falló. Si en el futuro alguien rompe la conmutatividad (por ejemplo, agregando un efecto secundario al evaluar `Suma` que dependa del orden), los `check`s detectan el contraejemplo de inmediato.

Benchmarks para las decisiones de performance

```
bench "literal" {
  eval(Lit(42))
}

bench "expresión profunda (3 niveles)" {
  eval(Suma(Mul(Lit(2), Lit(3)), Suma(Lit(4), Mul(Lit(5), Lit(6)))))
}
```

Dos benchmarks: el caso barato (un literal) y un caso más complejo (tres niveles de anidamiento). En mi máquina:

```
literal:                        1000 iter / 15  ns/iter
expresión profunda (3 niveles): 1000 iter / 134 ns/iter
```

El segundo es ~9 veces más caro que el primero. Esa es la información que necesitas si más adelante decides que el evaluador es un cuello de botella: sabes contra qué línea base estás midiendo.

Lo que el archivo no muestra

El mecanismo es directo. El archivo declara funciones, declara tests, declara checks, declara benches, todo en el mismo lugar. Tres comandos lo procesan:

```
$ kai test  ejemplos/cap07/05_evaluador_pruebas.kai
$ kai check ejemplos/cap07/05_evaluador_pruebas.kai
$ kai bench ejemplos/cap07/05_evaluador_pruebas.kai
```

Y `kai run` y `kai build` ignoran las tres construcciones. El binario que despliegas no carga ni los tests ni los checks ni los benches: solo el código de producción.

Esa unificación es lo que hace al modelo cómodo. No hay proyecto de tests aparte, no hay frameworks que importar, no hay decisiones sobre dónde poner cada cosa. La pregunta "¿dónde están las pruebas de esta función?" tiene una sola respuesta posible: al lado de la función.

Ejercicios

7.1. Toma una función simple que ya hayas escrito (puede ser de los capítulos anteriores o algo nuevo) y escribe tres tests para ella: uno con un caso típico, uno con un caso límite, y uno con una entrada inválida (si el tipo lo admite). Corre `kai test` y verifica que pasen los tres.

7.2. Escribe `fn esta_ordenada(xs: [Int]) : Bool` que devuelva `true` si la lista está ordenada de menor a mayor. Después escribe un `check` que verifique `esta_ordenada(list.sort(xs))` para todo `xs : [Int]`. ¿Qué pasa si tu `esta_ordenada` tiene un bug (por ejemplo, acepta listas que tienen un elemento "saltado")? El runner te debería entregar un contraejemplo.

7.3. Vuelve al evaluador del §7.6. Agrega un constructor nuevo `Resta(Expr, Expr)` al `Expr` y la rama correspondiente en `eval`. ¿Qué tests rompen? ¿Cuáles tests deberías agregar para

cubrir el nuevo caso? ¿Hay alguna **propiedad nueva** que valga la pena escribir como `check` (por ejemplo, `Resta(Lit(a), Lit(0)) == Lit(a)`)?

7.4. Para una operación de tu elección, escribe dos implementaciones: una "naive" y una "optimizada". Escribe un `bench` para cada una. ¿Cuántas veces más rápida es la optimizada? ¿La diferencia justifica la complejidad agregada?

7.5. Observa atentamente este `check` aparentemente inocente:

```
check "concatenar listas preserva el largo" with xs: [Int], ys: [Int] {
  list.length(list.concat([xs, ys])) == list.length(xs) + list.length(ys)
}
```

¿Qué propiedad expresa? ¿Por qué es trivial pero útil? ¿Qué debería pasar si alguien (tú, en seis meses, con prisa) "optimiza" `list.concat` y rompe la propiedad? Escribe el check, córrelo, y luego prueba romper `list.concat` mentalmente: ¿en qué iteración crees que el contraejemplo aparecería?

Capítulo 8 · Módulos, imports, organización del código

Hasta ahora todo tu código ha vivido en un archivo. Es la forma correcta de partir, pero apenas el programa crece, un archivo único se vuelve un problema: los lectores se pierden, los cambios chocan, las búsquedas confunden módulos lógicos que están físicamente revueltos. Hay que partir el código en pedazos con nombres.

kaikai resuelve esto con un sistema deliberadamente simple. **Un archivo es un módulo.** No hay declaraciones `module Foo`. No hay archivos especiales que reabran un módulo desde otro lado. El nombre del módulo se deriva de la ruta del archivo y eso es todo. Por encima de los módulos viene una segunda escala: un **proyecto**, descrito por un `kai.toml`, que organiza sus módulos y sus dependencias externas. Y por encima del proyecto, un **package manager** que resuelve dependencias entre proyectos.

Este capítulo recorre las tres escalas, de menos a más.

8.1 Un archivo, un módulo

Crea un archivo `aritmetica.kai` con algunas funciones:

```
pub fn duplicar(x: Int) : Int = x * 2

fn helper(x: Int) : Int = x + 1

pub fn doble_mas_uno(x: Int) : Int = helper(duplicar(x))
```

Tres decisiones de diseño aparecen en estas tres líneas:

- `pub` **marca lo que el módulo exporta.** Por defecto, una declaración es privada al archivo. Solo las cosas marcadas con `pub` son visibles desde otro módulo que importe este. Es exactamente al revés de Java o C++, donde lo público es el default y hay que recordar marcar lo privado.
- **Los nombres del módulo no necesitan declararse.** El archivo `aritmetica.kai` es el módulo `aritmetica`. Si lo pones bajo un subdirectorio como `util/aritmetica.kai`, pasa a ser el módulo `util.aritmetica`. El nombre se deriva de la ruta relativa a la raíz del proyecto.

- **Cualquier `fn`, `type`, `effect` o `let` puede llevar `pub`.** No hay diferencia en visibilidad entre tipos y funciones: una construcción es visible afuera si y solo si está marcada `pub`.

Las funciones privadas son útiles para descomponer una pública sin contaminar el namespace del consumidor. En `aritmetica`, `helper` solo existe dentro del archivo; quien importa el módulo no la ve.

8.2 `import` y nombres calificados

Para usar `aritmetica` desde otro archivo:

```
import aritmetica

fn main() {
  println("doble_mas_uno(5) = #{aritmetica.doble_mas_uno(5)}")
}
```

`import aritmetica` deja accesibles las cosas `pub` de ese módulo bajo el prefijo `aritmetica.`. La función `doble_mas_uno` se llama como `aritmetica.doble_mas_uno`.

El prefijo es deliberado. Cuando un proyecto crece a quince o veinte módulos, ver `aritmetica.doble_mas_uno` en una expresión te dice de inmediato de dónde sale. La alternativa, importar todos los nombres sueltos al namespace del consumidor, ahorra teclas pero pierde esa pista.

El mismo prefijo se usa para los tipos que el módulo exporta y para construirlos:

```
import geometria

fn main() {
  let a : geometria.Punto = geometria.Punto { x: 0.0, y: 0.0 }
  let b : geometria.Punto = geometria.Punto { x: 3.0, y: 4.0 }
  println("distancia = #{geometria.distancia(a, b)}")
}
```

Tres usos del prefijo, los tres coherentes: en la anotación de tipo (`: geometria.Punto`), en la construcción del record (`geometria.Punto { ... }`), y en la llamada a la función (`geometria.distancia(a, b)`). Un único patrón mental.

Si el módulo está bajo un subdirectorio, el import usa el mismo nombre punteado que el módulo:

```
import util.matematica

fn main() {
  println("3^4 = #{matematica.pow(3, 4)}")
}
```

Fíjate en dos detalles: el archivo está en `util/matematica.kai`, el módulo se llama `util.matematica`, pero al usarlo, el prefijo es solo `matematica` (el último segmento). Eso evita prefijos largos sin perder el origen estructural.

Tres formas de importar

kaikai admite tres formas que cubren el espacio:

```
import math.vector                # use qualified: vector.dot(a, b)
import math.vector as V           # alias: V.dot(a, b)
import math.vector.{dot, cross}   # nombres específicos en alcance
```

La primera es la más común y la que debería ser tu opción por defecto: el prefijo `vector.` deja claro de dónde sale cada nombre. La segunda es útil cuando el nombre del módulo es largo o aparece muchas veces y un alias corto te gana legibilidad sin perder la pista. La tercera es la **importación selectiva**: una lista explícita de nombres que se traen al namespace local. Te conviene solo cuando los nombres son los protagonistas del archivo y el prefijo se vuelve ruido (un `Punto` que aparece cuarenta veces, por ejemplo). El precio es que el lector ya no sabe de dónde salió `Punto` sin mirar arriba.

No hay wildcard import. Es decir, no existe `import math.vector.*` o equivalente que traiga "todo lo que el módulo exporta" al namespace local. Es una decisión deliberada: el wildcard parece cómodo al escribir pero hace que el lector futuro no sepa de dónde vino un nombre.

8.3 Visibilidad: el contrato del módulo

`pub` es el contrato que tu módulo le hace al mundo. Todo lo no marcado es privado, y al marcar algo `pub` estás diciendo "voy a sostener este nombre, esta firma, este tipo".

Una regla que el lenguaje no impone pero que conviene aplicar: **lo público debe ser estrecho.** Cada nombre `pub` es un contrato que vas a tener que mantener. Si exportas una función auxiliar útil hoy pero específica de tu implementación, mañana cuando refactorices vas a tener que elegir entre romper a quien la usa o cargar con código que ya no quieres. Lo público se mide en lo que prometes; lo privado, en lo que puedes cambiar sin avisar.

En la práctica, esto se traduce a:

- **Tipos exportados:** sí, suelen ser parte del contrato.
- **Funciones helper:** rara vez. Si las necesitas exportar, pregúntate si está en el lugar correcto.
- **Constantes:** sí, si forman parte del API. Si son detalles de implementación, no.

La mayoría de los lenguajes con visibilidad pública por defecto acaban con módulos cuyo "API real" se mezcla con todo el resto. kaikai invierte eso: lo público es lo que nombraste explícitamente.

Campos privados dentro de un tipo público

Hay una asimetría útil. Cuando declaras `pub type T = { ... }`, los campos del record son **públicos por defecto**: el tipo está en el contrato, sus campos también. Eso es lo que quieres la mayoría de las veces.

Pero a veces el tipo en sí merece ser parte del contrato y algún campo es detalle de implementación. La palabra `priv` antes del nombre del campo lo marca como invisible para otros módulos:

```
pub type Cuenta = {
  nombre:    String,
  priv saldo: Real,
}
```

Desde fuera del módulo que declara `Cuenta`, `c.saldo` no compila y un literal `Cuenta { nombre: "x", saldo: 100.0 }` tampoco. Solo el módulo declarante puede leer el campo y mencionarlo al construir. Los §4.1.1 mostró el detalle del patrón; lo importante para el cap. 8 es que `priv` opera a **granularidad de campo**, complementa el `pub` que controla visibilidad a nivel de declaración, y juntos cubren un caso muy común: "el tipo se exporta, su interior no".

El ejemplo `cap08/06_priv/` es un proyecto de dos archivos que demuestra exactamente este rechazo desde el otro lado del límite del módulo.

8.4 El stdlib que ya tienes

Hay un módulo especial que **no necesitas importar**: el stdlib core. Funciones como `println`, `assert`, `string_concat`, `real_sqrt` viven en el namespace global y están disponibles en todo archivo .kai sin trámite. Es lo que en otros lenguajes se conoce como **prelude**.

Lo que vive en core es deliberadamente pequeño: los tipos primitivos, las operaciones aritméticas, los `Option` y `Result`, algunas funciones de lista, IO básica con `println`. El resto del stdlib (encoding, redes, archivos, criptografía) vive en módulos separados que se importan como cualquier otro:

```
import list            # operaciones de lista no-prelude
import string          # operaciones de string
import encoding.json   # parseo de JSON
```

El criterio para core es severo: solo entra ahí lo que casi todo programa usa.

8.5 Proyectos: `kai.toml`

Hasta acá hablamos de **módulos** dentro de un mismo árbol de archivos. Cuando esos módulos se convierten en una unidad que quieres versionar, distribuir, o que depende de otras unidades similares, pasas a una segunda escala: el **proyecto**.

Un proyecto kaikai se describe con un archivo `kai.toml` en su raíz. La forma mínima es esta:

```
name = "miapp"
version = "0.1.0"

[dependencies]
```

`name` es el nombre del proyecto. Tiene que ajustarse a una gramática simple: minúsculas, dígitos, guion bajo y guion, sin empezar con dígito. Es la misma forma que usan Cargo, Go y Hex.pm, y evita problemas de path traversal, colisiones con flags y otros sustos.

`version` es la versión del proyecto. Antes de 1.0, `0.MINOR.PATCH` con cambios incompatibles subiendo MINOR (convención cz). Después de 1.0, semver estándar.

`[dependencies]` es la tabla donde declaras qué otros proyectos necesita el tuyo. Vacío si tu proyecto no depende de nadie más fuera del stdlib.

`edition`

Hay un campo opcional más que conviene fijar en cuanto un proyecto deja de ser un experimento:

```
name = "miapp"
version = "0.1.0"
edition = "hanga-roa"

[dependencies]
```

`edition` ata el código fuente a una versión del **contrato del lenguaje**: sintaxis, semántica del sistema de tipos, firma del stdlib, API del binario `kai`. Mientras una edición esté activa, kaikai garantiza que tu proyecto seguirá compilando al subir el compilador, aunque internamente cambien cosas. Cuando una edición se cierra y entra una nueva, kaikai sigue aceptando la anterior — solo que tienes que decir cuál usas.

Si omites el campo, el compilador asume la edición default de la instalación. Eso está bien para borradores; pero en un proyecto que va a vivir, fijar la edición es lo que evita que un upgrade silencioso te cambie el suelo. El cap. 16 cuenta el resto: cómo se eligen las ediciones, cómo migrar entre ellas, y qué pasa con APIs marcadas como `#[unstable]`.

Crear un proyecto nuevo

```
$ mkdir miapp && cd miapp
$ kai init miapp
kai-pkg: wrote kai.toml for package 'miapp'
```

`kai init` escribe el `kai.toml` esqueleto. Después agregas archivos `.kai` y los importas como vimos en §8.2.

8.6 Dependencias: git, path, lock

Cuando declaras una dependencia, tienes tres formas:

```
[dependencies]
manutara = { source = "github.com/kaikailang-org/manutara", ref = "v0.1.0" }
kohau = "github.com/kaikailang-org/kohau@v0.2.0"
local = { path = "../mi-otra-lib" }
```

La primera (tabla con `source` y `ref`) es la forma canónica para dependencias de git. `source` es la URL del repo. `ref` es lo que git entiende como referencia: una tag (`v0.1.0`), una branch (`main`), o un commit (`abc1234`). Las tags son la convención; branches y commits son escapes para casos puntuales.

La segunda (string `"<source>@<ref>"`) es la misma cosa pero abreviada. El `@` separa source de ref.

La tercera (`{ path = "..." }`) es la **dependencia local**. Apunta a otro proyecto en tu disco, normalmente porque lo estás desarrollando en paralelo. Edita el código de la dependencia, vuelve a correr tu app, los cambios se ven al instante sin republicar.

Agregar una dependencia

```
$ kai add github.com/kaikailang-org/manutara@v0.1.0
```

`kai add` hace dos cosas atómicamente: clona la dependencia y escribe la entrada en `kai.toml`. Si el clone falla (URL mala, ref que no existe, error de red), ni el manifest ni el lockfile cambian. Esto es importante: el árbol de trabajo nunca queda en un estado inconsistente.

El lockfile

Cuando resuelves dependencias por primera vez, `kai` clona los repos, captura el commit exacto (la SHA), y escribe un archivo `kai.lock`:

```
# kai.lock — generated by kai-pkg. Do not edit by hand.

[[package]]
name = "manutara"
source = "github.com/kaikailang-org/manutara"
ref = "v0.1.0"
sha = "abc123def456..."
```

El lockfile cierra el contrato. Si dos programadores corren `kai install` con el mismo `kai.toml` y el mismo `kai.lock`, **bajan exactamente la misma SHA**, exactamente el mismo árbol de archivos, exactamente el mismo binario al final. La reproducibilidad es la promesa central del lockfile.

Por eso `kai.lock` se commitea junto al código: es parte del contrato del proyecto. `kai.toml` declara qué quieres; `kai.lock` declara qué obtuviste.

Cuándo se actualiza el lock

- `kai install` lo crea si no existe; si existe, lo respeta.
- `kai update` lo regenera con la última versión de cada dependencia que cumpla la ref declarada.
- `kai add` lo refresca cuando agregas una dependencia nueva.
- `kai run` y `kai build` lo refrescan automáticamente si detectan deps en `kai.toml` que no están en el lock (la primera vez después de un `git clone`, por ejemplo).

8.7 Selección de versiones: MVS

Cuando un proyecto depende transitivamente del mismo otro proyecto por dos rutas distintas con versiones distintas declaradas, ¿qué versión gana?

kaikai resuelve esto con **minimum-version selection (MVS)**, el algoritmo que usa Go modules. La regla es brutal pero clara: del conjunto de versiones declaradas por la cadena transitiva entera, gana la **máxima**.

Si tu app declara `manutara@v0.1.0` y una dependencia tuya declara `manutara@v0.2.0`, todo el proyecto usa `v0.2.0`. La suposición subyacente es que las versiones son *backwards-compatible* dentro de un major: si todo el mundo respeta semver, subir al mayor número no debería romper a nadie.

MVS contrasta con el algoritmo de Cargo o npm, que resuelve restricciones complejas y a veces baja a una versión menor para satisfacer a alguien. MVS es **predecible**: dado el mismo árbol, el resultado es el mismo. No hay "resolver fallido", no hay diamond--dependency hell.

El precio es que la responsabilidad de mantener compatibilidad queda en los autores de bibliotecas. Si subes a una versión que rompe, todo el mundo que dependa transitivamente de ti se rompe. Eso te obliga a tomar el versionado en serio.

8.8 Cache y `kai install`

Cuando bajas una dependencia, no se queda en tu proyecto: se queda en un **cache compartido entre proyectos**, por defecto en `~/Library/Caches/kai/pkg` (macOS) o `~/.cache/kai/pkg` (Linux).

La estructura del cache es:

```
~/Library/Caches/kai/pkg/
  github.com/kaikailang-org/manutara/
    abc123def456.../          # contenido fijado a esa SHA
    789abc012def.../          # otra SHA del mismo repo
```

Cada entrada se identifica por su SHA, no por la ref que el usuario ve. Esto significa que tres proyectos distintos que piden `manutara@v0.1.0` comparten el mismo árbol en disco. Y si en algún momento `v0.1.0` se actualiza upstream (movimiento de tag, que no debería pasar pero pasa), el cache fijado al SHA original sigue intacto.

`kai install` se puede correr explícitamente, pero **`kai run` y `kai build` lo invocan automáticamente** cuando detectan deps no resueltas. En la práctica, después de `git clone` de un proyecto, basta con `kai run main.kai`: el driver baja lo que falta y corre.

8.9 Caso de estudio: refactor de un proyecto monolítico

Imagina un proyecto que partió como un solo archivo `main.kai` de 800 líneas. Adentro hay tres responsabilidades mezcladas: parseo de un formato de configuración, lógica de negocio, e IO con archivos. La velocidad de cambio cayó: cualquier modificación obliga a leer demasiado.

El refactor procede en cuatro pasos.

Paso 1: separar en módulos del mismo proyecto

Identificas las tres responsabilidades y las separas en archivos:

```
miapp/
├── kai.toml
├── main.kai       # solo el flujo principal
├── config.kai     # parseo del formato
└── negocio.kai    # lógica de dominio
```

En `main.kai`:

```
import config
import negocio

fn main() {
  let cfg = config.cargar("./config.toml")
  let resultado = negocio.procesar(cfg)
  println("resultado: #{resultado}")
}
```

`config.kai` y `negocio.kai` exportan solo las funciones que `main` necesita. Todo lo demás (helpers, tipos internos) queda privado.

Paso 2: revisar `pub`

Repasa cada `pub`. Pregunta para cada uno: ¿es de verdad parte del contrato del módulo, o es un helper que dejé exportado por inercia? Cada `pub` extra es un compromiso futuro.

Paso 3: extraer una dependencia local

`config.kai` resulta útil más allá de este proyecto. La extraes a su propio repositorio:

```
config-lib/
├── kai.toml       # name = "config_lib"
└── config.kai

miapp/
├── kai.toml       # ahora depende de config_lib
├── main.kai
└── negocio.kai
```

En `miapp/kai.toml`:

```
name = "miapp"
version = "0.1.0"

[dependencies]
config_lib = { path = "../config-lib" }
```

Mientras desarrollas, `config_lib` queda como dependencia local. `main.kai` cambia su import a `import config_lib.config` (o un alias, si el nombre largo molesta).

Paso 4: publicar

Cuando `config_lib` está estable, le pones una tag:

```
$ cd ../config-lib
$ git tag v0.1.0
$ git push origin v0.1.0
```

Y en `miapp` cambias el path a git:

```
[dependencies]
config_lib = { source = "github.com/tuusuario/config-lib", ref = "v0.1.0" }
```

`kai install` baja la versión fijada por el lockfile. El código de `main.kai` y `negocio.kai` no cambia: los imports siguen siendo los mismos.

Esa es la trayectoria completa: de archivo único a módulos, de módulos a proyectos, de proyecto local a dependencia distribuida. Cada paso es reversible. Cada paso es opcional.

8.10 Filosofía: simple y previsible

Las decisiones de este capítulo no son las más expresivas que existen. Hay lenguajes con sistemas de módulos más potentes: re-exports automáticos, alias dinámicos, módulos parametrizados por valores. kaikai elige una variante austera y se queda ahí.

La razón es la misma que en el resto del lenguaje: lo que ahorra dolores de cabeza vale más que lo que agrega potencia. Un sistema de módulos donde nada se importa sin que aparezca en una lista visible, donde el nombre de un módulo se deriva de su ruta, y donde las dependencias se fijan a un SHA exacto es un sistema que se entiende leyéndolo, sin tener que ejecutarlo mentalmente.

Es el mismo principio que en pattern matching exhaustivo: si el compilador te puede llevar al lugar exacto donde tienes que hacer algo, no necesitas otra herramienta más sofisticada.

Ejercicios

8.1. Toma un programa que hayas escrito en los capítulos anteriores y pártelo en dos archivos. Identifica qué declaraciones necesitan ser `pub` y cuáles pueden quedarse privadas. ¿Sentiste la tentación de marcar algo `pub` "por si acaso"?

8.2. Crea un proyecto con `kai init`. Agrégale un archivo auxiliar bajo un subdirectorio (por ejemplo, `util/strings.kai`). Importa una función desde `main.kai`. ¿Cómo se llama el módulo desde el punto de vista del import?

8.3. Escribe dos proyectos hermanos en disco: una biblioteca `mi_lib` con una función pública, y una app `mi_app` que la usa con `{ path = "../mi_lib" }`. Edita `mi_lib`, vuelve a correr `mi_app`. ¿Cuánto demora el ciclo edit-run?

8.4. Mira un proyecto en `github.com/kaikailang-org/kaikai` (el compilador mismo) y abre su `kai.toml` si lo tiene. ¿Qué dependencias declara? ¿Reconoces el patrón de versionado?

8.5. Si `mi_app` declara `manutara@v0.1.0` y una dependencia transitiva declara `manutara@v0.2.0`, ¿qué versión usa el proyecto? ¿Por qué ese diseño descarta el "diamond dependency hell"? Da un escenario donde la regla de MVS te puede sorprender.

8.6. Una biblioteca tuya tenía una función `pub fn parse_legacy(s: String)` que ya nadie usa. ¿Bajo qué condiciones podrías removerla sin liberar una versión `1.0`? ¿Qué tipo de comunicación con tus usuarios necesitas antes?

Capítulo 9 · Protocolos

Hasta acá viste cómo agrupar datos (records, tipos suma) y cómo definir funciones que operan sobre esos datos. Lo que falta es responder una pregunta concreta: **¿cómo le agregas operaciones a un tipo desde fuera de su declaración?**

Por ejemplo: `Punto` es un record con dos campos. Quieres que se imprima como `(3, 4)` cuando aparece en una interpolación. ¿Modificas el tipo? ¿Pasas una función al `println`? ¿Defines una variante de `println` solo para `Punto`? Ninguna escala.

La respuesta de kaikai son los **protocolos**: un contrato con nombre y un puñado de operaciones, que cualquier tipo puede satisfacer. Conceptualmente, los protocolos hacen lo que las **interfaces** de Go, los **traits** de Rust, los **protocols** de Clojure y Elixir, y la parte fácil de las **typeclasses** de Haskell. Pero kaikai elige un punto preciso del espacio de diseño: **single-dispatch explícito, sin propagación de constraints, sin tipos de orden superior**. Eso le saca complejidad al sistema de tipos a cambio de algunas cosas que no se pueden expresar, y este capítulo cubre las dos caras.

9.1 Por qué hay protocolos

Tres dolores concretos que los protocolos resuelven.

Imprimir tus tipos sin escribir cada vez. Cuando declaras un record, querer imprimirlo en logs, en respuestas, en errores, no debería costarte una función `usuario_a_string` por cada tipo. Con `Show` implementado una vez, la interpolación `"#{usuario}"` lo usa automáticamente.

Igualdad estructural sin esfuerzo. Cualquier tipo nuevo necesita "esto es igual a aquello" en algún momento. Sin protocolos, escribes `eq_punto`, `eq_cuenta`, `eq_factura`, y vives el resto del proyecto recordando cuál nombre usaste en cuál archivo. Con `Eq` implementado, la operación se llama `eq` para todos.

Comparación, hashing, serialización. El mismo argumento del párrafo anterior, multiplicado por las tres operaciones estándar que casi todo tipo necesita en algún momento.

Sin protocolos, cada uno de los tres se resuelve con convenciones de nombres caso por caso, o con `match` enormes que enumeran cada tipo posible. Con un solo mecanismo, los tres y los que vengan después se resuelven en una línea por tipo.

9.2 Declarar un `protocol` y `impl`

La sintaxis es directa. Un protocolo declara un nombre y una o más operaciones:

```
protocol Show {
  show(x: Self) : String
}
```

`Self` es un nombre reservado que se refiere al tipo que más adelante implemente el protocolo. Cada operación menciona `Self` al menos una vez: por eso se llama **single--dispatch**, la operación se decide a partir de un único tipo, el de `Self`.

Para implementarlo, escribes un `impl ... for ...`:

```
type Punto = { x: Int, y: Int }

impl Show for Punto {
  fn show(p: Punto) : String =
    "(" ++ int_to_string(p.x) ++ ", " ++ int_to_string(p.y) ++ ")"
}
```

Y a partir de ese momento, `show(p)` llama a tu implementación cuando `p : Punto`:

```
fn main() {
  let p = Punto { x: 3, y: 4 }
  println(show(p))            # imprime "(3, 4)"
  println("p está en #{p}")       # interpolación: usa Show
}
```

Tres detalles que conviene fijar:

- **El cuerpo del `impl` lista las funciones del protocolo, una por una**, con la sintaxis estándar de `fn`. Cada `fn` en el bloque tiene que coincidir con la firma declarada en el `protocol`, sustituyendo `Self` por el tipo concreto.
- **Una sola implementación por par `(protocolo, tipo)`**. Si hay dos `impl Show for Punto` en la misma compilación, el compilador rechaza con "duplicate impl". No hay sobreescritura ni resolución contextual.
- **La regla orphan**: solo puedes implementar un protocolo `P` para un tipo `T` si `P` se declara en tu módulo **o** `T` se declara en tu módulo. Esto evita que dos paquetes externos definan implementaciones conflictivas para tipos que ambos importan. Es una limitación práctica, no del sistema de tipos.

9.3 Los cinco protocolos del stdlib

kaikai trae cinco protocolos en `stdlib/protocols.kai` que vas a usar todo el tiempo. La tabla siguiente los lista con sus operaciones:

Protocolo	Operaciones	Para qué
`Show`	`show(x: Self) : String`	Convertir a string para imprimir
`Eq`	`eq(a: Self, b: Self) : Bool`	Igualdad
`Ord`	`cmp(a: Self, b: Self) : Int`, `min`, `max`	Orden total
`Hash`	`hash(x: Self) : Int`	Para tablas hash y conjuntos
`Serialize`	`to_string`, `from_string`	Conversión texto ↔ valor

Los tipos primitivos (`Int`, `Real`, `Bool`, `String`, `Char`) ya tienen implementaciones para los cinco protocolos. Cuando declaras un tipo nuevo, eliges qué protocolos vale la pena implementar para él.

`Ord` merece una nota: tiene **tres operaciones**, no una. `cmp(a, b) : Int` devuelve un entero negativo si `a < b`, cero si son iguales, positivo si `a > b`. Las otras dos, `min(a, b)` y `max(a, b)`, devuelven uno de los dos argumentos según el orden. Las tres se implementan juntas en el mismo bloque:

```
impl Ord for Cuenta {
  fn cmp(a: Cuenta, b: Cuenta) : Int =
    if a.saldo < b.saldo { 0 - 1 }
    else if a.saldo > b.saldo { 1 }
    else { 0 }

  fn min(a: Cuenta, b: Cuenta) : Cuenta =
    if a.saldo < b.saldo { a } else { b }

  fn max(a: Cuenta, b: Cuenta) : Cuenta =
    if a.saldo > b.saldo { a } else { b }
}
```

Si `Ord` te da pereza implementar tres operaciones, espera a §9.4: en muchos casos, el compilador las deriva por ti.

9.4 `#[derive(...)]` y cuándo usarlo

Para records donde la implementación obvia "delega en cada campo" basta, kaikai te ofrece un atajo: la directiva `#[derive(...)]` antes de la declaración de tipo.

```
#[derive(Show)]
type Persona = {
  nombre: String,
  edad: Int,
}

#[derive(Show, Eq)]
type Punto = {
  x: Int,
  y: Int,
}
```

`#[derive(Show)]` le dice al compilador "genérame un `Show` para este record, recorriendo los campos y delegando en el `Show` de cada tipo de campo". El resultado para `Persona`:

```
$ kai run ejemplo.kai
Persona { nombre: Ada, edad: 30 }
```

El formato canónico, `TipoNombre { campo: valor, ... }`, es lo que el `#[derive(Show)]` produce, y es razonable para debugging y logs. Si quieres otro formato (por ejemplo, el clásico `(3, 4)` para un punto), escribes el `impl` a mano, como en §9.1.

`#derive` funciona para los cinco protocolos del stdlib, siempre que **cada campo del record también implemente** el protocolo que estás derivando. Si tu record tiene un campo cuyo tipo no tiene `Show`, `#[derive(Show)]` falla en compilación con un mensaje que apunta al campo problemático.

La regla práctica:

- **Empieza con `#derive`**: es la forma más rápida y casi siempre correcta para records.
- **Cambia a `impl` manual** cuando la implementación derivada no te sirve: formato distinto, igualdad solo por algunos campos, comparación según un campo específico (no el orden natural).

Ya viste `#[derive(Show)]` sobre `Punto` en el tour (§1.6); acá mostramos también la implementación manual y cuándo conviene una sobre la otra.

9.5 Protocolos propios

Los cinco del stdlib son los más comunes, pero nada te impide declarar los tuyos. Es exactamente la misma sintaxis que usa el stdlib, pero en tu código:

```
protocol Drawable {
  dibujar(x: Self) : String
}

type Circulo = { radio: Int }
type Cuadrado = { lado: Int }
type Triangulo = { base: Int, altura: Int }

impl Drawable for Circulo {
  fn dibujar(c: Circulo) : String =
    "Círculo de radio " ++ int_to_string(c.radio)
}

impl Drawable for Cuadrado {
  fn dibujar(c: Cuadrado) : String =
    "Cuadrado de lado " ++ int_to_string(c.lado)
}

impl Drawable for Triangulo {
  fn dibujar(t: Triangulo) : String =
    "Triángulo " ++ int_to_string(t.base) ++ "x" ++ int_to_string(t.altura)
}
```

A partir de ese momento, tres tipos distintos comparten la operación `dibujar`. La función polimórfica `dibujar` se resuelve estáticamente: el compilador sabe en cada llamada qué `impl` usar a partir del tipo del argumento.

`Drawable` es un ejemplo de juguete; los casos reales que verás en código kaikai incluyen `Encodable` para distintos formatos, `Loggable` para que el sistema de logging sepa representar tu tipo, `Validable` para reglas de validación, etc. Cualquier "comportamiento que comparten varios tipos" es candidato.

9.6 Por qué no hay typeclasses al estilo Haskell

Los protocolos de kaikai pueden parecerse a las typeclasses de Haskell (y se inspiran en ellas), pero son **deliberadamente más simples**. Tres cosas que kaikai no hace y que Haskell sí:

Sin constraints en firmas de funciones. Esta es la diferencia más visible. En Haskell, una función que ordena una lista declara que el tipo del elemento tiene que tener `Ord`:

```
sort :: Ord a => [a] -> [a]
```

El `Ord a =>` es la **constraint**. Cuando llamas a `sort xs`, el compilador busca por su cuenta el `Ord` para el tipo de `xs` y se lo "inyecta" a la función sin que tú escribas nada. La constraint viaja escondida.

En kaikai eso no existe. No puedes escribir:

```
fn ordenar[T : Ord](xs: [T]) : [T] = ...    # ERROR: kaikai no admite constraints
```

¿Cómo se ordena entonces una lista? La función recibe el **comparador como un argumento explícito**:

```
fn sort_by[T](xs: [T], cmp: (T, T) -> Int) : [T] = ...
```

Y el call site **nombra** el comparador. Si `Transaccion` implementa `Ord`, su `cmp` está disponible como una función ordinaria, y la pasas:

```
list.sort_by(transacciones, cmp)   # cmp viene de impl Ord for Transaccion
```

La diferencia es chica de escribir pero grande conceptualmente: en Haskell el `Ord` está implícito, en kaikai está explícito. La función `sort_by` no "exige" que `T` tenga `Ord`: solo exige que **alguien le pase una función de comparación**. Que esa función venga de un `impl Ord for T` es decisión del que llama, no de la firma de `sort_by`.

Sin tipos de orden superior (HKT). `protocol Functor[F[_]]` no parsea. Los parámetros de tipo son siempre de primer orden. Esto descarta una familia de abstracciones (Functor, Monad, Applicative, etc.) que en Haskell son centrales y que en kaikai se resuelven con efectos algebraicos (cap. 12) y combinadores explícitos.

Sin propagación de constraints. Una función polimórfica no "lleva" el `Ord` consigo a las funciones que llama. Si llamas a algo que requiere `Ord`, le pasas el comparador.

¿Qué se gana con estas restricciones? Tres cosas:

- **Compilación rápida**. La inferencia de tipos sigue siendo Hindley-Milner extendido con efectos, sin el costo del resolver de constraints de Haskell.
- **Errores claros**. Si una función necesita `Ord` y no lo recibe, el error apunta al sitio donde te falta pasar el comparador. No hay cadenas de "no instance for `Ord (Maybe a)` because of `Ord a`".
- **El call site dice qué hace**. Cuando ves `list.sort_by(xs, cmp)`, sabes que se ordena con `cmp`. Cuando ves `sort xs` en Haskell, tienes que mirar la firma de `sort` para saber qué `Ord` se está usando.

¿Qué se pierde? Algunas abstracciones que en Haskell son elegantes, particularmente todo lo que vive sobre Functor y amigos. Ese trade-off es deliberado: las abstracciones que kaikai prioriza viven en el sistema de efectos (cap. 12), no en el sistema de tipos.

9.7 Operadores: `+`, `==`, `<` como protocolos

Una nota práctica: los operadores estándar **son** protocolos. `==` es `Eq.eq`, `<` es comparación basada en `Ord.cmp`, `+` es `Add.add`. Cuando declaras `impl Eq for Cuenta`, automáticamente `c1 == c2` (con `c1, c2 : Cuenta`) llama a tu implementación.

```
#[derive(Eq)]
type Punto = { x: Int, y: Int }

fn main() {
  let p = Punto { x: 3, y: 4 }
  let q = Punto { x: 3, y: 4 }
  if p == q { println("iguales") }   # usa Eq.eq derivado
}
```

Esto unifica la sintaxis: `+` para `Int`, `Real`, vectores, matrices, monedas, todos los que tengan `impl Add for ...`. `==` para todo lo que tenga `Eq`. La uniformidad no es casualidad; es el primer beneficio de tener un mecanismo único para "operaciones dispatched por tipo".

Los operadores que kaikai trata como protocolos:

Operador	Protocolo	Op
`==`, `!=`	`Eq`	`eq`
`<`, `<=`, `>`, `>=`	`Ord`	`cmp`
`+`, `-`, `*`, `/`	`Add`, `Sub`, `Mul`, `Div`	`add`, `sub`, `mul`, `div`
`++`	`Concat`	`concat`

Los tipos primitivos los implementan todos. Los tipos tuyos los implementan cuando los declaras. Y cuando algún operador no tiene sentido para un tipo, simplemente no lo implementas, y el compilador rechaza esa expresión.

Ejercicios

9.1. Define `type Distancia = { metros: Int }` y dale un `impl Show` para que `show(d)` produzca `"42 m"`. Luego prueba `println("la distancia es #{d}")`. Cuidado: la interpolación funciona, pero recuerda el workaround del cap. 9 si llamas a otro protocolo dentro.

9.2. Define `type Carta = { palo: String, valor: Int }` y dale `impl Ord` que ordene por `valor`. Verifica con tres cartas que `cmp(c1, c2)` devuelve los números esperados.

9.3. Toma cualquier record que hayas escrito en capítulos anteriores y agrégale `#[derive(Show, Eq)]` arriba de la declaración. Verifica que `show` y `eq` funcionan sin que hayas escrito nada más.

9.4. Declara un protocolo propio `Validable` con una operación `validar(x: Self) : Result[String, Self]` que devuelva `Ok(x)` si el valor es válido o `Err("razón")` si no. Implementa `Validable` para `type Edad = { años: Int }` de manera que rechace edades negativas o mayores a 130.

9.5. Lee el código del stdlib en `stdlib/protocols.kai`. ¿Cuántas operaciones tiene `Hash`? ¿Cómo se usaría para implementar una tabla de hash de `Cuenta`? Diseña la firma de la función que harías para "buscar una cuenta por id" en una tabla de hash hipotética: ¿qué tendría que estar implementado en `Cuenta` para que funcionara?

Capítulo 10 · Unidades de medida y branded types

En 1999, la NASA perdió la sonda Mars Climate Orbiter (327 millones de dólares de proyecto) porque dos módulos de software se intercambiaban valores numéricos sin acuerdo sobre las unidades. Uno producía empuje en libras-fuerza por segundo, otro lo leía como newtons por segundo. Nadie había escrito las unidades en la interfaz. La sonda se desintegró contra Marte.

Es un ejemplo extremo, pero la familia es enorme: dos componentes que comparten el mismo tipo numérico pero no la misma interpretación. Pasar segundos donde se esperan milisegundos. Sumar saldos en monedas distintas. Mezclar un `UserId` con un `OrderId` porque ambos son enteros. En todos los casos, el sistema de tipos del lenguaje ve dos números o dos strings y no tiene cómo distinguirlos.

kaikai resuelve esta familia entera con una sola construcción: las **unidades de medida**. La idea, heredada de F#, es que puedes anotar un número con una unidad (`Real<USD>`, `Int<Seconds>`) y el compilador rechaza cualquier operación que mezcle unidades incompatibles. La unidad **vive en el tipo**, **se borra en runtime**, y queda documentada en cada firma que la toca.

Este capítulo cubre las unidades clásicas (física, finanzas, tiempo) y su uso menos obvio pero más frecuente en código del día a día: los **branded types**, donde la unidad es solo una etiqueta que distingue tipos que el lenguaje, sin ella, trataría como iguales.

10.1 `unit` y literales anotados

Una unidad se declara con la palabra clave `unit`:

```
unit USD
unit EUR
unit m
unit sec
unit kg
```

Eso es todo. `unit USD` introduce un símbolo `USD` que el sistema de tipos puede usar como anotación. No hay implementación, no hay valor en runtime: `unit` solo declara la existencia del símbolo.

Para anotar un número con una unidad, usas paréntesis angulares en el literal:

```
let precio : Real<USD> = 1.50<USD>
let velocidad : Real<m / sec> = 9.81<m / sec>
let timeout : Int<sec> = 30<sec>
```

Tres cosas que vale fijar:

- **Las unidades se escriben con** `<...>`, no con `[...]`. Los corchetes son para parámetros de tipo (`List[Int]`, `Option[String]`); los paréntesis angulares son para unidades.
- **El literal y el tipo se anotan los dos.** `1.50<USD>` es un literal `Real` con unidad `USD`. `Real<USD>` es el tipo. Los dos coinciden, pero conviene entender que cada uno cumple un rol distinto.
- **El compilador permite cualquier identificador como unidad.** Por convención, las unidades SI van en minúscula (`m`, `s`, `kg`), las nombradas por personas en titlecase (`Newton`, `Pascal`), y las monedas en mayúsculas según ISO 4217 (`USD`, `EUR`, `CLP`). El compilador no fuerza ninguna de estas convenciones: es estilo de la comunidad.

10.2 Aritmética con unidades

La aritmética entre valores con la misma unidad funciona como esperarías. Sumar dos `Real<USD>` da otro `Real<USD>`:

```
let precio : Real<USD> = 1.50<USD>
let propina : Real<USD> = 0.30<USD>
let total : Real<USD> = precio + propina    # 1.80 USD
```

Pero **mezclar unidades incompatibles es un error de compilación**. Si intentas:

```
let mezcla = precio + 1.20<EUR>    # error de tipo
```

el compilador rechaza con un error que dice "esperaba `USD`, encontré `EUR`". El programa nunca corre con valores mezclados: el bug se detecta antes de que exista.

La regla más amplia es:

- `+`, `-` entre dos valores de la misma unidad: legal, preserva la unidad.
- `*`, `/` entre valores con unidades: legal, **componen las unidades**. Lo veremos en §10.3.
- `+`, `-` entre unidades distintas: error de tipo.
- **Comparaciones** (`<`, `==`) entre unidades distintas: error de tipo.

La aritmética con un valor sin unidad y uno con unidad es asimétrica. Multiplicar por un escalar (`2.0 * precio`) es legal: el escalar se trata como adimensional. Sumar un escalar (`precio + 1.0`) es error: no sabemos en qué unidad está el `1.0`.

10.3 Álgebra de unidades: producto, cociente, potencia

Multiplicar dos unidades produce una unidad compuesta. El ejemplo canónico es la fuerza:

```
let masa : Real<kg> = 70.0<kg>
let aceleracion : Real<m / sec^2> = 9.81<m / sec^2>
let fuerza : Real<kg * m / sec^2> = masa * aceleracion
```

`kg * m / sec^2` es la unidad compuesta de fuerza. El compilador la deduce automáticamente de los operandos: `kg * (m / sec^2)` se simplifica a `kg * m / sec^2`. Si después divides la fuerza por el área para obtener presión, la unidad final es `kg / (m * sec^2)`, todo derivado mecánicamente:

```
let area : Real<m^2> = 4.0<m^2>
let presion : Real<kg / (m * sec^2)> = fuerza / area
```

El sistema de tipos hace **álgebra de unidades**. `m * m` se simplifica a `m^2`, `sec / sec` se cancela a sin-unidad, `(m / sec) * (sec)` se cancela a `m`. Es álgebra abeliana sobre los símbolos de unidad, y cualquier expresión legal en matemática elemental sobre unidades es legal en kaikai.

Alias para unidades derivadas

Cuando una composición aparece a menudo, puedes darle un nombre con un `unit` con cuerpo:

```
unit Newton = kg * m / sec^2
unit Pascal = Newton / m^2
unit Hertz  = 1 / sec
```

Y a partir de ahí, `Real<Newton>` es exactamente lo mismo que `Real<kg * m / sec^2>`: el compilador los acepta intercambiables. La diferencia es para quien lee: `Real<Newton>` comunica intención; `Real<kg * m / sec^2>` comunica derivación.

10.4 Unidades genéricas

Hasta acá, cada función opera sobre una unidad concreta. Pero muchas operaciones son **agnósticas a la unidad**: promediar, sumar, ordenar, encontrar el máximo. Esas funciones se escriben **genéricas sobre la unidad**, igual que una función puede ser genérica sobre el tipo de los elementos de una lista.

```
fn promedio[u: Measure](a: Real<u>, b: Real<u>) : Real<u> =
  (a + b) / 2.0
```

`u : Measure` declara `u` como un parámetro de tipo en el **kind** `Measure`. Lo único que `u` admite es ser una unidad. La función `promedio` acepta dos `Real<u>` y devuelve un `Real<u>`: el "para cualquier u" es lo que permite usarla con `USD`, `kg`, `m/sec`, lo que sea, **siempre y cuando los dos argumentos tengan la misma unidad**.

```
let pp : Real<USD> = promedio(10.0<USD>, 20.0<USD>)  # 15 USD
let pm : Real<kg>  = promedio(70.0<kg>, 80.0<kg>)     # 75 kg
```

Pero esto es error de tipo:

```
let mezcla = promedio(10.0<USD>, 70.0<kg>)  # u no puede ser USD y kg
```

El compilador instancia `u = USD` para el primer argumento, y exige que el segundo también use `u = USD`. `kg` no calza, y el programa no compila.

Las unidades genéricas son lo que hace al sistema **escalable**. Las funciones del stdlib (`list.sum`, `list.max`, `list.min`) son polimórficas sobre la unidad: si pasas una `[Real<USD>]`, te devuelven `Real<USD>`. Si pasas una `[Real<kg>]`, te devuelven `Real<kg>`. La unidad se preserva sin que tú la nombres.

10.5 Conversiones explícitas

¿Qué pasa cuando **sí** quieres mezclar dos unidades distintas? Por ejemplo, sumar un saldo en USD con uno en EUR. El compilador no te lo permite por accidente, pero te lo permite con una **conversión explícita**.

La técnica es un factor de conversión que lleva la unidad cociente `destino/origen`. Multiplicar por él cancela la unidad origen y deja la unidad destino:

```
let monto_eur : Real<EUR> = 80.0<EUR>
let tasa : Real<USD / EUR> = 1.10<USD / EUR>
let monto_usd : Real<USD> = monto_eur * tasa   # 88 USD
```

La aritmética se sigue: `EUR * (USD / EUR)` cancela `EUR` y deja `USD`. El compilador verifica que la cancelación sea correcta: si pones la tasa al revés (`Real<EUR / USD>`), la multiplicación produce `Real<EUR^2 / USD>`, que no calza con el tipo `Real<USD>` esperado en el `let`.

La regla mental: **la conversión es una multiplicación por un factor cuyas unidades cancelan a la salida**. Es exactamente lo que se hace en física a mano cuando uno escribe "$10 \text{ km} \times 1000 \text{ m/km} = 10000 \text{ m}$". kaikai obliga a escribir esa multiplicación explícita.

Esto es deliberado y a la vez liberador. Deliberado porque forzar una conversión visible hace que el programador piense qué tasa está usando, en qué momento se aplicó, y de dónde salió. Liberador porque, una vez escrita la conversión, el compilador garantiza que no la olvidaste en ningún lugar.

10.6 Branded types

Las unidades son útiles cuando el valor "tiene una dimensión física": metros, kilogramos, dólares. Pero la misma maquinaria se aplica a un caso más cotidiano y más frecuente: **distinguir valores que tienen el mismo tipo subyacente pero significan cosas distintas**.

El ejemplo clásico son los identificadores. Un `UserId` es un entero. Un `OrderId` también. Sin unidades, el compilador no puede distinguirlos:

```
fn cancelar_orden(id: Int) : Unit / Stdout = ...
fn enviar_email(uid: Int) : Unit / Stdout = ...
```

```
let user_id = 42
let order_id = 99
cancelar_orden(user_id)  # bug: pasamos un id de usuario
                         # a una función que espera un id
                         # de orden, pero compila igual.
```

Con unidades, los dos identificadores son tipos distintos:

```
unit UserId
unit OrderId

fn cancelar_orden(id: Int<OrderId>) : Unit / Stdout = ...
fn enviar_email(uid: Int<UserId>) : Unit / Stdout = ...

let user_id : Int<UserId>  = 42<UserId>
let order_id : Int<OrderId> = 99<OrderId>

cancelar_orden(user_id)  # ERROR de tipo: UserId ≠ OrderId
```

El compilador detecta el bug antes de que exista. La técnica de usar una unidad como **etiqueta** sobre un tipo numérico se llama **branded type**, y es uno de los usos que más rinde en código del día a día. Casos típicos:

- `Int<UserId>` vs `Int<OrderId>`: identificadores que comparten tipo subyacente.
- `Int<Cents>` vs `Int<Quantity>`: un dinero en centavos vs una cantidad de unidades.
- `Int<Seconds>` vs `Int<Milliseconds>`: los timeouts mal expresados son responsable de un porcentaje no menor de bugs intermitentes.
- `String<Email>` vs `String<Username>`: strings que pasaron por validaciones distintas.
- `String<RawHtml>` vs `String<Sanitized>`: input sin escapar vs ya escapado para inyección segura.

Los dos primeros casos sobre `Int` funcionan en cualquier versión actual de kaikai. Los últimos dos sobre `String` están implementados parcialmente; el manejo completo de branded types sobre `String` y records arbitrarios es una extensión que la doc del lenguaje lista como próximo hito.

Costo cero en runtime

Las unidades se borran después de la verificación de tipos. El binario que produce `kai build` opera con `Int` plano, `Real` plano, `String` plano. La unidad **no existe** en runtime: no hay tag, no hay verificación dinámica, no hay overhead. La promesa es la misma de los efectos algebraicos y de los contratos: información en el tipo, costo cero en runtime.

10.7 Caso de estudio: cartera multi-moneda

Cerramos con un ejemplo integrador. Una cartera contiene saldos en distintas monedas. Sumar saldos de la misma moneda es trivial; combinarlos en un total exige convertir.

```
unit USD
unit EUR
```

```
unit CLP

fn sumar[c: Measure](a: Real<c>, b: Real<c>) : Real<c> = a + b

fn convertir[origen: Measure, destino: Measure](monto: Real<origen>, tasa: Real<destino / origen>) :
Real<destino> = monto * tasa
```

Tres declaraciones. `sumar` es genérica sobre la moneda y preserva la unidad de los argumentos: la misma técnica del §10.4. `convertir` recibe un monto y una tasa, y devuelve el monto en la unidad destino, gracias a la cancelación de unidades.

Y el cálculo principal:

```
fn main() {
  let saldo_usd_1 : Real<USD> = 100.0<USD>
  let saldo_usd_2 : Real<USD> = 50.0<USD>
  let total_usd : Real<USD> = sumar(saldo_usd_1, saldo_usd_2)    # 150

  let saldo_eur : Real<EUR> = 80.0<EUR>
  let tasa_eur_usd : Real<USD / EUR> = 1.10<USD / EUR>
  let eur_en_usd : Real<USD> = convertir(saldo_eur, tasa_eur_usd)
  let total_global : Real<USD> = sumar(total_usd, eur_en_usd)      # 238

  let saldo_clp : Real<CLP> = 100000.0<CLP>
  let tasa_clp_usd : Real<USD / CLP> = 0.0011<USD / CLP>
  let clp_en_usd : Real<USD> = convertir(saldo_clp, tasa_clp_usd)
  let total_final : Real<USD> = sumar(total_global, clp_en_usd)   # 348
}
```

Tres conversiones, tres aplicaciones de `sumar`. Cada una verificada por el compilador: si en algún paso hubieras intentado sumar `Real<EUR>` con `Real<USD>` sin convertir, no compilaría. Si la tasa estuviera invertida (`<EUR / USD>` en vez de `<USD / EUR>`), tampoco. La cartera completa es **un mini-sistema de tipos sobre dinero** que el compilador sostiene.

¿Qué pasa el día que un colega llega y agrega una nueva moneda? Declara un `unit JPY`, agrega su tasa, y el resto del código sigue compilando o no según corresponda. Ningún cálculo viejo se rompe (los tipos son ortogonales), y los cálculos que necesitan considerar JPY tienen que decirlo explícito. Es el mismo principio que el cap. 5 mostró con las uniones de errores: agregar un componente nuevo es seguro porque el compilador te lleva al lugar exacto donde necesitas hacer algo.

Compáralo con la versión sin unidades:

```
fn sumar(a: Real, b: Real) : Real = a + b

let total = sumar(saldo_usd, saldo_eur)   # compila, suma valores
                                           # en monedas distintas
                                           # sin convertir.
```

Funciona, devuelve un número, y produce un total que **no significa nada**. En kaikai con unidades, ese mismo programa no compila. El bug que en otros lenguajes se descubre en producción (o nunca, según la suerte) aquí no existe.

Ejercicios

10.1. Define `unit Celsius` y `unit Fahrenheit`. Escribe una función `fn celsius_a_fahrenheit(c: Real<Celsius>) : Real<Fahrenheit>` que aplique la fórmula correcta. ¿Cuál es el factor de conversión? ¿Tiene unidad?

10.2. Define `unit Cents` (centavos) y escribe `fn pagar(monto: Int<Cents>) : Unit / Stdout`. Construye un ejemplo donde el lenguaje detecte un bug de "pasar un monto en pesos donde se esperaban centavos".

10.3. Toma el caso de estudio del §10.7 y agrega una nueva moneda `JPY` con su tasa. ¿Cuántas líneas tienes que cambiar? ¿Hay alguna línea del código existente que se rompa solo porque agregaste la unidad?

10.4. Escribe una función genérica `fn rango[u: Measure](xs: [Real<u>]) : Option[Real<u>]` que devuelva la diferencia entre el máximo y el mínimo de una lista, o `None` si la lista está vacía. ¿Qué unidad tiene el resultado?

10.5. En tu trabajo o en un proyecto personal, identifica **dos** lugares donde dos enteros distintos significan cosas distintas (identificadores, índices, contadores, timeouts). Escribe en pseudo-kaikai cómo serían las firmas de las funciones afectadas si usaras branded types. ¿Cuántos bugs históricos podrían haberse evitado?

Capítulo 11 · Programación por contrato y refinement types

Este capítulo cierra la Parte II y completa el hilo "información en el tipo, costo cero en runtime" que arrancan los efectos algebraicos (cap. 12), continúan las unidades de medida (cap. 10) y rematan acá con dos mecanismos que provienen de Eiffel (1986) y Ada 2012: los **contratos** (precondiciones y postcondiciones que viven en la firma de una función) y los **refinement types** (restricciones sobre los valores que un tipo admite).

Los dos mecanismos comparten la misma idea: **enunciar restricciones en el tipo, hacer que el compilador las verifique cuando puede, y diferirlas a runtime cuando no puede**. Donde difieren es en el alcance: los contratos hablan de **operaciones** (qué espera y qué garantiza una función); los refinements hablan de **valores** (qué números o strings son válidos). Las dos cubren áreas distintas y se complementan.

11.1 Por qué contratos y refinements van juntos

Imagina que quieres modelar una cuenta bancaria. La regla natural es "el saldo nunca puede ser negativo". Hay dos formas de expresar esa regla:

- **En el valor**: declarar `type SaldoValido = Int where self >= 0`. Cualquier valor de tipo `SaldoValido` cumple la regla por construcción. El compilador rechaza cualquier intento de meter un negativo.
- **En la operación**: declarar `fn retirar(c: Cuenta, monto: Int) : Cuenta requires c.saldo >= monto ensures result.saldo >= 0`. La función exige una condición sobre los argumentos y promete una sobre el resultado.

Las dos formas dicen lo mismo y la una sin la otra es incompleta. Los refinements describen **qué valores son legales**; los contratos describen **qué hacen las operaciones con esos valores**. Una cuenta bancaria robusta usa los dos: saldo refinado a no negativo, operaciones con precondiciones que aseguran que la regla se mantiene.

kaikai trata las dos como un solo proyecto. La doc del lenguaje (`refinements-and-contracts.md`) dice:

> *Together they form a single coherent mechanism — types describe what values are valid, contracts describe what operations guarantee — and the two share most of the implementation machinery.*

Es por eso que viven en el mismo capítulo.

11.2 `requires` y `ensures` en una firma

Un contrato se escribe como anotaciones en la firma de una función, antes del `=` que abre el cuerpo:

```
fn divide(a: Int, b: Int) : Int
  requires b != 0
  ensures  result * b + (a % b) == a
= a / b
```

Tres componentes:

- `requires <expr>`: una **precondición**. La expresión tiene que ser `true` al **entrar** a la función. Quien llama es responsable. Si se viola, el bug es de quien llama.
- `ensures <expr>`: una **postcondición**. La expresión tiene que ser `true` al **salir** de la función. Tu cuerpo es responsable. Si se viola, el bug es interno.
- `result`: un nombre reservado dentro del `ensures` que se refiere al valor de retorno.

Una función puede tener múltiples `requires` y múltiples `ensures`. El compilador los acumula:

```
fn retirar(c: Cuenta, monto: Int) : Cuenta
  requires monto > 0
  requires c.saldo >= monto
  ensures  result.saldo == c.saldo - monto
=
  Cuenta { ...c, saldo: c.saldo - monto }
```

Dos precondiciones (el monto positivo, y que haya saldo suficiente), una postcondición (la cuenta resultante tiene exactamente `c.saldo - monto`). Las precondiciones se verifican en orden al entrar; la postcondición al salir.

Tres detalles que vale fijar:

- **`requires` y `ensures` no son comentarios**. Son código que el compilador emite como verificaciones reales. Si una precondición se viola en runtime, el programa aborta con un mensaje claro:

  ```
  panic: requires violated in `divide`
  required: b != 0
  declared at line 9, col 14
  ```

- **El compilador prueba estáticamente lo que puede**. Si llamas a `divide(10, 0)` con literales, el compilador ve que `b == 0` y rechaza la llamada en compile time: no esperas a runtime. Si llamas con valores dinámicos, inserta el assert.

- **Los contratos no se ejecutan en builds de release**, según un flag de compilación. En ese modo, los `requires` y `ensures` desaparecen del binario; el costo es cero. Para desarrollo y para tests, los contratos se evalúan.

11.3 `result` y los nombres en alcance dentro del `ensures`

Dentro de una postcondición tienes a disposición:

- `result`: el valor de retorno.
- **Los nombres de los parámetros**: sus valores de entrada.
- **Cualquier función pura** que el módulo provea.

Esto te deja escribir relaciones entre la entrada y la salida:

```
fn duplicar(n: Int) : Int
  ensures result == n * 2
= n + n

fn ordenar(xs: [Int]) : [Int]
  ensures list.length(result) == list.length(xs)
= ...
```

La primera dice "la salida es el doble de la entrada". La segunda dice "la lista resultante tiene el mismo largo que la entrada": un invariante razonable de cualquier ordenamiento. Fíjate que **no** estamos diciendo que `result` está ordenado, solo que conserva el largo. Las postcondiciones documentan lo que vale la pena documentar; no tienen que ser exhaustivas.

A diferencia de Eiffel, **no hay** `old`. En Eiffel necesitas `ensures balance = old balance + amount` porque los records son mutables y el `balance` que ves en el `ensures` ya cambió. En kaikai los records son inmutables: el `c` que entra y el `result` que sale son **valores distintos**, los dos en alcance, sin necesidad de un mecanismo para "el valor antes de la llamada".

11.4 Refinement types

Mientras que los contratos hablan de operaciones, los **refinement types** hablan de valores. Un refinement type es un tipo base más un predicado:

```
type NoNeg = Int where self >= 0
type Probabilidad = Real where self >= 0.0 and self <= 1.0
type Edad = Int where self >= 0 and self <= 130
type Puerto = Int where self >= 1 and self <= 65535
```

El predicado se refiere a `self`, el valor que estamos restringiendo. Cualquier valor de tipo `NoNeg` cumple `self

> *= 0* `por construcción; cualquier` *Probabilidad* `cae en el intervalo unitario; cualquier` *Puerto` está dentro del rango TCP.*

Construir un valor del tipo refinado requiere que el predicado se cumpla. Si lo cumples con un literal, el compilador lo verifica en compile time:

```
let x : NoNeg = 16       # OK: 16 >= 0
let y : NoNeg = 0 - 5    # ERROR: -5 no satisface self >= 0
```

Si lo cumples con un valor dinámico, el compilador inserta un verificación en runtime: igual que con un `requires` cuyo argumento no se puede deducir estáticamente.

Las funciones que aceptan tipos refinados **se benefician de la garantía sin verificarla**. Si tu firma dice `n: NoNeg`, adentro puedes asumir `n >= 0` sin escribir un `if`. Eso es exactamente lo que un contrato `requires n >= 0` hace, pero codificado en el tipo en vez de en la firma.

¿Cuándo usas uno y cuándo el otro? La regla simple:

- **Refinement type** cuando la restricción **define qué es un valor válido** del dominio. `Edad`, `Probabilidad`, `Puerto` son ejemplos clásicos: el tipo no tiene sentido fuera de la restricción.
- **Contrato** cuando la restricción es **sobre la operación**, no sobre el valor. "no dividir por cero" es del operando de división; "el monto a retirar tiene que ser positivo" es de la operación retirar; "la lista resultante mantiene el largo" es del comportamiento de la función.

A veces los dos aplican y eliges según legibilidad. La cuenta bancaria del §11.7 usa contratos porque "el saldo no se puede dejar negativo" es una propiedad del **comportamiento de las operaciones**, no del valor del saldo.

11.5 Cuándo se prueba estáticamente, cuándo en runtime

kaikai trata los contratos y los refinements como un **continuo entre estático y dinámico**, decidido por lo que el compilador puede demostrar. Tres niveles:

Compile time, completamente probado. Cuando los argumentos son literales o el compilador conoce sus rangos:

```
divide(10, 0)            # ERROR de compilación: 0 != 0 es falso
let x : NoNeg = 0 - 5    # ERROR de compilación: -5 < 0
```

El programa no llega a generar binario. Es la garantía más fuerte que existe.

Compile time, parcialmente probado. Cuando los rangos se pueden inferir de un análisis acotado, kaikai prueba lo prudente sin invocar un solver SMT pesado. El alcance es limitado: comparadores con literales sobre `Int`, `Bool`. Si el predicado pasa por aritmética compleja o por funciones que el compilador no puede inspeccionar, se difiere.

Runtime. Cuando lo anterior no es decidible, el compilador emite un assert en el código generado. El programa compila; la verificación se hace al ejecutar la función. Si falla, aborta con `panic: requires violated`.

Lo que kaikai **no hace**, deliberadamente, es **invocar un solver SMT como Z3 o CVC5** para probar contratos arbitrariamente complejos. Esa es la frontera con SPARK (el subset verificable de Ada). kaikai prefiere un evaluador de intervalos pequeño, decidible, lineal (y

diferir el resto a runtime), sobre tener compilación impredecible y dependencias externas pesadas.

La regla mental: **lo que el compilador puede probar barato, lo prueba; lo demás se prueba al ejecutarse**. El binario lleva ambos casos sin que tengas que distinguir cuál de los dos aplicó.

11.6 Tres formas de garantía

Llevamos tres mecanismos para "garantizar que el código hace lo correcto" repartidos en tres capítulos:

Mecanismo	Cap.	Qué garantiza	Cuándo se verifica
`test`	7	Para una entrada específica, la salida es esta	Bajo `kai test`
`check`	7	Para **toda** entrada, vale esta invariante	Bajo `kai check`, con valores generados
Tipos suma + `match`	5	Cubrir todos los casos posibles del tipo	Compile time
Contratos + refinements	11	Restricciones sobre entrada/salida y valores válidos	Compile time cuando se puede, runtime cuando no

Las tres son herramientas distintas con áreas que se solapan. Una función bien escrita probablemente las use todas:

- **Tipos** que son lo más estrechos posibles para el dominio (sum types, refinements, units of measure).
- **Contratos** que documentan lo que la función exige y garantiza, en términos relacionales.
- **Tests** que cubren los casos contractuales puntuales que el cliente pide.
- **Checks** que verifican las invariantes algebraicas ("invertir dos veces es identidad", "ordenar preserva el largo").

No son redundantes: cada uno atrapa una clase distinta de bugs y los tres juntos forman una red de seguridad mucho más densa que cualquiera por separado. Y los cuatro tienen costo cero en runtime cuando no fallan: un `test` que pasa no se ejecuta en producción; un contrato que se puede probar estáticamente no genera assert; un refinement legal no genera verificación dinámica.

11.7 La familia Design by Contract

Los contratos no son invención de kaikai. Son una idea con historia, y vale la pena ubicar a kaikai en esa historia.

Eiffel (Bertrand Meyer, 1986) introdujo el término "Design by Contract" y la mayoría de las ideas: `require` / `ensure` sobre métodos, invariantes de clase, herencia con debilitamiento

de precondiciones y fortalecimiento de postcondiciones. La sintaxis canónica vive dentro del cuerpo:

```
deposit (amount: REAL)
  require
    positive_amount: amount > 0
  do
    balance := balance + amount
  ensure
    balance_increased: balance = old balance + amount
end
```

kaikai toma la idea pero pone los contratos en la **firma**, no en el cuerpo, y elimina `old` porque los records son inmutables.

Ada 2012 agregó "aspect specifications" a Ada con la sintaxis `with Pre =>`, `with Post =>`. Esto es lo más parecido a kaikai estilísticamente:

```
function Divide (A, B : Integer) return Integer
  with Pre  => B /= 0,
       Post => Divide'Result * B = A;
```

El `with Pre => / Post =>` de Ada y el `requires / ensures` de kaikai son la misma idea anotacional. Ada también introdujo **subtypes con predicados** (`subtype Positive is Integer range 1 .. Integer'Last`), que son los antepasados directos de los refinement types de kaikai.

SPARK es el subset verificable de Ada, que usa un solver SMT para probar contratos arbitrarios estáticamente. Kaikai **no adopta esto a propósito**: SPARK requiere instalar Z3 o similar, y los tiempos de compilación se vuelven impredecibles. La doc del lenguaje (`refinements-and-contracts.md`) lo dice explícito: el evaluador de intervalos es de unas pocas centenas de líneas, decidible, lineal; lo que no entra ahí se posterga a runtime sin remordimiento.

D tiene contratos similares a kaikai con la sintaxis `in { ... } / out (result) { ... }`. **Cobra**, **Kotlin** (con `require/check`), **Clojure** (con `:pre/:post`) son otras versiones más livianas de la misma idea.

Lo que **distingue** a kaikai en esta familia:

1. **Sin SMT**. La línea de SPARK queda fuera. La consecuencia es que los contratos complejos se verifican en runtime; el beneficio es compilación rápida y sin dependencias externas.

2. **Pureza por defecto**. Eiffel y Ada manejan mutabilidad rampante; los contratos tienen que lidiar con `old` y aliasing. kaikai parte de inmutabilidad: la postcondición habla de la entrada y de la salida como dos valores que coexisten, sin máquina extra.

3. **Continuidad con el resto del sistema de tipos**. Contratos y refinements son la **tercera pata** de "información en el tipo, costo cero en runtime", junto a efectos algebraicos y unidades de medida. Las tres usan el mismo patrón: declarar en la firma o en el tipo, verificar estáticamente cuando se puede, runtime cuando hace falta. Eiffel y Ada no tienen efectos algebraicos ni UoM, así que sus contratos viven más solos.

11.8 Lo que kaikai no hace, y por qué

Vale enumerar lo que kaikai **deliberadamente** no implementa:

- **No SMT solving**. Si tu contrato es `ensures result.entries == sort(c.entries)`, kaikai no va a probar estáticamente que tu cuerpo en efecto ordena la lista. Lo va a verificar en runtime.
- **No refinements arbitrarios sobre estructuras complejas**. `Real where 0.0 <= self <= 1.0` es legal. `[Int] where list.length(self) > 0` no está implementado en la versión inicial. La restricción se puede expresar con un sum type (`type ListaNoVacia = ...`) o un wrapper, pero no con un refinement directo.
- **No herencia de contratos** al estilo Eiffel. kaikai no tiene clases ni herencia; los contratos viven en la firma de cada función individual.

¿Por qué estas restricciones? El argumento es el mismo de todo el lenguaje: **simplicidad y predecibilidad**. Un sistema de tipos que invoca un solver es opaco: el programador no sabe por qué su programa compila o no, y los mensajes de error se vuelven incomprensibles. Un sistema acotado, que prueba lo obvio y difiere lo demás, da garantías más débiles pero **comprensibles**, y deja al runtime del programa saludable como red de seguridad.

11.9 Caso de estudio: cuenta bancaria

Cerramos con una cuenta bancaria mínima, donde los contratos documentan y aseguran el comportamiento.

```
type Cuenta = {
  titular: String,
  saldo: Int,
}

fn abrir(titular: String, deposito_inicial: Int) : Cuenta
  requires deposito_inicial >= 0
  ensures  result.saldo == deposito_inicial
=
  Cuenta { titular: titular, saldo: deposito_inicial }

fn depositar(c: Cuenta, monto: Int) : Cuenta
  requires monto > 0
  ensures  result.saldo == c.saldo + monto
=
  Cuenta { ...c, saldo: c.saldo + monto }

fn retirar(c: Cuenta, monto: Int) : Cuenta
  requires monto > 0
  requires c.saldo >= monto
  ensures  result.saldo == c.saldo - monto
=
  Cuenta { ...c, saldo: c.saldo - monto }
```

Tres operaciones, cinco precondiciones, tres postcondiciones. Lectura humana:

- `abrir` abre una cuenta con saldo inicial no negativo. La postcondición confirma que el saldo de la cuenta nueva es exactamente lo depositado.
- `depositar` exige que el monto sea positivo (no acepta depósitos de cero o negativos), y promete que el saldo final es el inicial más el monto.
- `retirar` exige monto positivo y suficiente saldo, y promete que el saldo final es el inicial menos el monto.

¿Qué pasa si alguien (tú, en seis meses, con prisa) escribe `retirar(cuenta, 0 - 50)` (pasando un negativo)? El contrato `requires monto > 0` se viola y el programa aborta con un mensaje que apunta a la línea exacta del `requires`. No silencio, no comportamiento extraño, no saldo inconsistente: aborto inmediato y diagnóstico.

¿Y si el cuerpo de `retirar` tuviera un bug (alguien cambia `c.saldo - monto` por `c.saldo + monto` accidentalmente)? El `ensures result.saldo == c.saldo - monto` se viola al salir y aborta también. La postcondición es tu seguro contra bugs internos, así como el `requires` es tu seguro contra abusos de quien llama.

Con cuatro líneas de contratos, esta cuenta bancaria mínima documenta sus reglas, las verifica al ejecutarse, y deja un diagnóstico claro cuando se rompen. Compáralo con la versión sin contratos:

```
fn retirar(c: Cuenta, monto: Int) : Cuenta =
  Cuenta { ...c, saldo: c.saldo - monto }
```

Funciona en el caso feliz. Pero alguien que llama con un monto negativo acaba con una cuenta cuyo saldo creció (porque `c.saldo - (-50) == c.saldo + 50`), y nadie se entera. Alguien que llama con más monto que saldo termina con una cuenta de saldo negativo, y nadie se entera. Los bugs que en kaikai con contratos son abortos inmediatos, en kaikai sin contratos son saldos incorrectos sin que nadie se entere en producción.

Ejercicios

11.1. Define `type Edad = Int where self >= 0 and self <= 130`. Escribe `fn promedio_edades(a: Edad, b: Edad) : Edad`. ¿En qué línea va a aparecer la verificación cuando construyes una `Edad` desde un valor dinámico?

11.2. Toma la función `divide` del §11.2. Agrega una postcondición que asegure que cuando `b > 0` y `a > 0`, el resultado es no negativo. ¿Cómo se ve la postcondición? ¿Qué pasa si tu cuerpo tuviera un bug que devolviera un negativo en algún caso?

11.3. Reescribe la cuenta bancaria del §11.9 usando un **refinement type** para el saldo (`type SaldoValido = Int where self >= 0`), y `Cuenta = { titular: String, saldo: SaldoValido }`. ¿Qué precondiciones y postcondiciones se vuelven redundantes con este cambio? ¿Qué información pierde la firma?

11.4. Imagina una función `fn percentil(p: Probabilidad, xs: [Real]) : Real`. ¿Qué precondiciones agregarías sobre `xs`? ¿Qué postcondiciones documentan el comportamiento correcto? ¿Cuál de las dos formas (refinement type o contrato) usarías para cada restricción?

11.5. Lee `docs/refinements-and-contracts.md` del repo de kaikai. Identifica una restricción que la doc menciona como "post-MVP". Discute con un colega o con un agente IA qué consecuencias

tendría implementarla: qué clase de errores nuevos atraparía, qué clase de programas se volverían más cargados de verificaciones.

Capítulo 12 · Efectos algebraicos

Este es el capítulo donde kaikai paga su novedad. Hasta ahora hemos visto tipos, pattern matching, protocolos, unidades de medida, contratos. Todas son piezas elegantes pero no únicas: las encontrarías parecidas en Haskell, en Rust, en F#. Los **efectos algebraicos** son lo que distingue a kaikai de casi cualquier lenguaje de uso real hoy.

La idea es simple de enunciar y rara al principio: **una función declara en su firma qué efectos usa, pero no cómo se realizan**. Imprimir a pantalla, leer un archivo, fallar con un error, suspender la ejecución, generar un número aleatorio: todos son efectos. La función dice "yo necesito esta capacidad"; otro código más arriba decide qué significa "esta capacidad" en este contexto. La separación entre **qué** y **cómo** es el corazón del sistema.

Si esto suena a *dependency injection*, mantén la idea cerca. Si suena a excepciones, también. Si suena a generadores, igualmente. La razón por la que un solo mecanismo se parece a tantas cosas es que en la teoría detrás, todas esas cosas son lo mismo. Los efectos algebraicos son la generalización.

Vamos despacio. Este capítulo se lee mejor con paciencia que con prisa: la primera lectura es para que cada idea aparezca; la intuición sólida llega cuando vuelvas un mes después y todo se sienta obvio.

12.1 La fricción que los efectos resuelven

Antes de mostrar la sintaxis, veamos los problemas concretos que los efectos resuelven. Si reconoces el patrón en tu trabajo, sabrás por qué pagar el costo de aprender una herramienta nueva.

Excepciones invisibles

En Java o Python, cualquier llamada a una función puede lanzar una excepción y nada en la firma te lo dice. Lees el código y no sabes qué puede fallar. Lo descubres en producción.

```
def cargar_usuario(id):
    return db.get(id)  # ¿puede fallar? ¿con qué excepciones?
```

Lenguajes con excepciones verificadas (Java) intentaron arreglarlo forzándote a declarar `throws`. El resultado fue que la gente escribió `throws Exception` para callarlo, y volvimos al inicio. Lenguajes funcionales modernos (Rust, OCaml, kaikai con el cap. 5) empujan a usar `Result` o `Option`: el costo de fallar aparece en el tipo, y quien llama decide qué hacer.

Bien para fallas locales, pero pesado cuando hay muchas funciones que fallan: la firma se llena de wrapping y unwrapping.

`async` / `await` que infecta

En JavaScript, Python, C#, Rust, marcar una función como `async` te obliga a marcar como `async` también a todas las que la llaman. Un cambio aparentemente local contagia el árbol entero de llamadas.

```
async function leer(path) { ... }
async function procesar(path) {
  const data = await leer(path);   // procesar también tuvo que ser async
  return transformar(data);
}
```

Bob Nystrom le puso nombre a esto en 2015 en un ensayo famoso: **"What Color is Your Function?"**. La idea es que `async` divide las funciones en dos colores. Las rojas (`async`) y las azules (no-async). Una roja puede llamar a una azul, pero una azul no puede llamar a una roja. Si tienes una función azul y necesitas usar una roja adentro, tienes que repintar la azul. Y la que la llamaba. Y así hasta arriba. Es una infección que no se queda local.

El punto del ensayo no es que `async` sea malo. Es que esta clase de marcadores en la firma, cuando son específicos a un solo tipo de efecto, crean dos sistemas paralelos que no componen. `async` no compone con generadores: necesitas `async function*`. No compone con excepciones de la misma manera (las excepciones en funciones async se vuelven rejected promises). Cada combinación nueva requiere su propia sintaxis.

Los efectos algebraicos resuelven el problema al nivel de raíz: **no hay colores especiales**, hay una sola dimensión que es la fila de efectos. `async` no necesita ser una propiedad sintáctica de la función; es simplemente un efecto entre otros. Y la fila se extiende sin sintaxis nueva: `/ Async + Fail` no es más raro que `/ Async`.

Inyección de dependencias

Para hacer una función testeable, le pasas como parámetro las "cosas externas" que usa: el reloj, el logger, el acceso a base de datos. Le pasas mocks en los tests, las cosas reales en producción.

```
class Procesador {
  public Procesador(Clock clock, Logger logger, DbClient db) { ... }
  public void run() { ... }
}
```

Funciona, pero ensucia las firmas y obliga a wiring manual. Y cuando una función nueva quiere usar uno de los servicios, hay que agregarlo a constructores arriba en la jerarquía.

El patrón común

Las tres frustraciones tienen la misma forma:

- Una función necesita una **capacidad** (la capacidad de fallar, la capacidad de suspenderse, la capacidad de loggear).

- La capacidad debe ser **explícita** (visible en el tipo) para no sorprender.
- La capacidad debe ser **provista por el contexto** (quien llama, test, framework) sin que la función se entere.
- Múltiples capacidades deben **componerse limpio**.

Los efectos algebraicos son **una sola construcción** que cubre los cuatro puntos. Lo que las excepciones, `async` / `await` y la inyección de dependencias hacen por separado y con frecuencia mal, los efectos lo hacen con un solo mecanismo.

12.2 Declarar un `effect`

Un efecto es una **interfaz**. Declara qué operaciones existen, con qué firma, pero no cómo se implementan.

```
effect Log {
  log(msg: String) : Unit
}
```

Esto introduce un efecto llamado `Log` con una operación `log` que recibe un string y no devuelve nada útil. Cualquier función que llame a `Log.log(...)` está usando el efecto `Log`.

Lo que **no** introduce: implementación. Aquí no hay cuerpo, no hay `if`, no hay print. Solo la firma. Esa es la diferencia fundamental con un protocolo o una interfaz tradicional: el efecto no decide nada, solo declara lo que se puede pedir.

Las operaciones de un efecto se declaran sin la palabra `fn` y sin cuerpo: solo nombre, parámetros y tipo de retorno. Un mismo efecto puede tener varias operaciones:

```
effect Io {
  print(s: String) : Unit
  read_line()      : String
}
```

12.3 Llamar a una operación: la firma cambia

Para usar una operación, llamas al método del efecto:

```
fn greet(name: String) : Unit / Log {
  Log.log("hola, " ++ name)
}
```

Dos cosas nuevas:

- `Log.log(...)` es la sintaxis de invocación. El efecto es el nombre del namespace; la operación es el método.
- `: Unit / Log` en la firma. La barra introduce la **fila de efectos** (effect row). Es la lista de efectos que esta función necesita para correr. Sin ese `/ Log`, la función no compila: está usando una capacidad que no declaró.

El sistema de tipos garantiza que **toda función que usa un efecto lo declara**. Si llamas `Log.log(...)` desde una función sin `/ Log`, el compilador rechaza con un mensaje claro. No hay escape: los efectos son visibles en la firma, siempre.

Y esto vale recursivamente. Si `greet` usa `Log` y `main` llama a `greet`, entonces `main` también necesita `Log` en su firma, a menos que **maneje** el efecto antes (eso es §12.4).

```
fn main() : Unit / Log {      # propaga el efecto
  greet("kaikai")
}
```

Es el mismo principio del contagio que vimos en `async`/`await`, pero sin el problema de los colores: aquí no hay sintaxis especial para "pagar el efecto" en la llamada. `Log.log(...)` es una llamada como cualquier otra. La fila en la firma es donde vive la disciplina, y agregar un efecto nuevo no introduce un color nuevo incompatible con el resto: solo extiende la fila.

Varios efectos: la fila

Si una función usa más de un efecto, los enumera con `+`:

```
fn procesar() : Int / Log + Fail {
  Log.log("inicio")
  if condicion_mala() {
    Fail.fail("no se puede")
  }
  42
}
```

`Log + Fail` es la **fila** de efectos. El orden no importa: la fila es un conjunto, no una secuencia. `Log + Fail` y `Fail + Log` son la misma fila para el compilador. El operador `+` es solo sintaxis para construirla.

12.4 Manejar un efecto con `handle ... with`

La parte interesante: **decidir qué significa un efecto** en un punto del programa. Eso es lo que hace `handle ... with`.

```
fn main() {
  handle {
    greet("kaikai")
    greet("Eduardo")
  } with Log {
    log(msg, resume) -> {
      println("[INFO] " ++ msg)
      resume(())
    }
  }
}
```

Lectura literal: "ejecuta este bloque, y para cuando alguien dentro invoque `Log.log`, hazlo así". El handler intercepta cada llamada a `log`, decide qué pasa, y reanuda con `resume(...)`.

Tres detalles que vale fijar:

- `handle { body } with Effect { clauses }` es una construcción de control, en la misma familia que `if` y `match`. No es una llamada a función. `handle` y `with` son palabras reservadas.
- **`body` es donde `Effect` queda manejado.** Adentro, llamar a `Log.log(...)` es legal sin tener `Log` en la fila de la función que envuelve, porque el handler la provee. Afuera del `with`, el efecto vuelve a ser exigido por el sistema de tipos.
- **`resume(())` continúa la ejecución del body** desde el punto donde se llamó `log`, con el valor que pasaste como argumento.

Lo notable: **el `main` no necesita declarar `Log` en su firma**. El `handle` lo "consume": el bloque de adentro lo usa, el `with` lo provee, y la fila de efectos del `main` queda sin `Log`. El sistema de tipos sigue siendo estricto, pero el handler es la puerta por la que un efecto sale del scope.

El mismo body con dos handlers

Lo poderoso aparece cuando notas que `greet` no cambia entre ejecuciones. Con un handler obtienes logs verbosos; con otro, silencio:

```
# Handler verboso
handle {
  greet("modo verboso")
} with Log {
  log(msg, resume) -> {
    println("[INFO] " ++ msg)
    resume(())
  }
}

# Handler silencioso
handle {
  greet("modo silencioso")
} with Log {
  log(msg, resume) -> resume(())   # ignora el mensaje
}
```

`greet` no se entera de la diferencia. Lo mismo vale para un handler que escribe a archivo, uno que acumula los mensajes en una lista, uno que los manda por red. La función `greet` es **agnóstica al handler**.

Esto es lo que reemplaza la inyección de dependencias. No hay constructor que pasar, no hay servicio que mockear: el handler ES la implementación.

12.5 `resume`: el handler decide qué pasa después

`resume` es la pieza que más confunde al inicio y la que más rinde cuando se entiende. Es el **continuation** del body desde el punto de la operación.

Cuando el body invoca `Log.log("hola")`, el control de la ejecución salta al handler. El handler recibe dos cosas:

1. El argumento que se pasó a la operación: `"hola"`.
2. Una función `resume` que, si se llama, continúa el body desde donde se quedó.

```
log(msg, resume) -> {
  println("[INFO] " ++ msg)   # decide qué hacer con el efecto
  resume(())                  # devuelve el control al body
}
```

`resume(())` pasa `()` como valor de retorno de la operación `log` (que devuelve `Unit`). El body continúa después del `Log.log(...)` con ese valor.

Operaciones que devuelven valores

`Log.log` no devuelve nada útil, pero otras operaciones sí. Un efecto puede **proveer** un valor:

```
effect Ask {
  name() : String
}

fn saludar() : String / Ask {
  "hola, " ++ Ask.name()
}

fn main() : Unit / Stdout {
  let mensaje = handle {
    saludar()
  } with Ask {
    name(resume) -> resume("mundo")
  }
  println(mensaje)   # imprime "hola, mundo"
}
```

`Ask.name()` suspende el body. El handler recibe `resume` y decide qué `String` darle a quien llamó: aquí, `"mundo"`. `resume("mundo")` continúa el body con ese valor, y el `++` lo concatena.

Esto reemplaza muchos usos de la inyección de dependencias: en vez de pasar `nombre` como parámetro hasta el fondo del árbol de llamadas, lo "preguntas" via efecto, y el handler en `main` decide qué responder. En tests, otro handler responde algo distinto.

Handlers que NO llaman a `resume`

Si el handler **no** llama a `resume`, el body se descarta y el valor del handler se vuelve el valor de todo el `handle`. Esto es como las excepciones se construyen:

```
effect Fail {
  fail(motivo: String) : Nothing
}

fn dividir(a: Int, b: Int) : Int / Fail {
  if b == 0 { Fail.fail("división por cero") }
  else      { a / b }
}
```

```
fn main() : Unit / Stdout {
  let r = handle {
    let x = dividir(10, 2)
    let y = dividir(20, 0)    # acá se invoca Fail.fail
    x + y                     # nunca llega
  } with Fail {
    fail(motivo, resume) -> {
      println("falló: " ++ motivo)
      0                       # valor de reemplazo
    }
  }
  println("resultado: #{r}")  # imprime: resultado: 0
}
```

La firma `fail(motivo: String) : Nothing` declara que la operación **nunca regresa**. `Nothing` es el tipo vacío de kaikai (el bottom type del cap. 3): no tiene habitantes, no se puede construir un valor de `Nothing`. Por construcción, no hay nada que pasar a `resume`, así que el sistema de tipos garantiza que no se puede continuar el body después de `Fail.fail`. El programa no se rompe si lo intentas, el compilador no te deja escribir el código.

Esa es la clave: las "excepciones" de kaikai son un caso particular del mecanismo general. No hay sintaxis especial para `try/catch`; hay `handle` y un efecto cuya operación devuelve `Nothing`.

12.6 Handlers con estado: el patrón `State`

Hasta acá los handlers fueron sin estado: solo decidían qué hacer y reanudaban. Pero un handler puede llevar **su propio estado** sin que el body se entere. Esto reemplaza la mutación global.

```
effect State[T] {
  get() : T
  set(v: T) : Unit
}

fn suma(xs: [Int]) : Int {
  handle {
    list.foreach(xs, (x) => State.set(State.get() + x))
    State.get()
  } with State[Int](0) {
    get(resume)    -> resume(state)        # devuelve estado
    set(v, resume) -> resume((), v)        # cambia estado
    return(x)      -> x                    # descarta estado al final
  }
}
```

Tres cosas nuevas:

- `State[T]` **es paramétrico.** El tipo `T` es lo que el estado guarda. Aquí va a ser `Int`.

- `with State[Int](0)` instala el handler con estado inicial `0`. El argumento entre paréntesis es el valor inicial.
- `state` es un identificador especial disponible dentro de las cláusulas del handler. Se refiere al valor actual del estado.
- `resume(value, new_state)` tiene dos argumentos cuando el handler tiene estado: el valor de retorno de la operación, y el estado nuevo. `resume(value)` sin segundo argumento deja el estado igual.
- `return(x) -> x` se ejecuta cuando el body termina normalmente. `x` es el resultado del body. Aquí descartamos el estado final y devolvemos solo `x`; si quisieras ambos, harías `return(x) -> (x, state)`.

Para el body, no hay mutación: solo invocaciones a operaciones puras. La mutación vive completa dentro del handler. Desde afuera del `handle`, no se ve nada.

Este patrón es genérico: con la misma forma se construyen `Reader` (entorno de lectura), `Writer` (acumulación de salida), contadores, caches, sesiones. Todos sin tocar variables globales y sin propagar parámetros.

12.7 `var`, `Ref[T]` y `Array[T]`: dos mecanismos distintos

`State[T]` es la herramienta general para llevar un valor que cambia, pero escribir un `handle ... with State[Int](0)` cada vez que quieres un contador local sería tedioso. Por eso kaikai trae azúcar sintáctica y, separadamente, un efecto del stdlib para casos donde la memoria sobrevive al bloque. Son dos construcciones distintas que vale la pena no confundir.

`var`: azúcar sobre `State[T]`

La forma corta de una celda local:

```
fn contar_pares(xs: [Int]) : Int {
  var n = 0
  list.foreach(xs, (x) => {
    if x % 2 == 0 {
      n := @n + 1
    }
  })
  @n
}
```

Tres formas nuevas:

- `var n = 0` declara la celda con su valor inicial.
- `@n` lee el valor actual.
- `n := v` escribe `v`.

¿Cómo funciona por dentro? **`var` es azúcar sintáctico sobre `State[T]`.** El compilador reescribe

```
var n = 0
... resto del bloque ...
```

a

```
handle {
  ... resto del bloque ...
} with State[Int](0) as n {
  get(resume)    -> resume(state)
  set(v, resume) -> resume((), v)
  return(x)      -> x
}
```

Como el `handle` que se inserta queda **dentro del mismo bloque** donde el `var` aparece, el efecto `State[Int]` se cierra ahí mismo y no escapa a la firma de la función. Para quien la llama, `contar_pares` es `: Int`. Sin efectos.

El efecto no se enmascara: el `handle` está literalmente al lado del `var`. La fila se cierra en el lugar exacto donde la celda se declara.

Y no es caro: el compilador detecta el patrón "celda local con `resume` de un disparo" y lo especializa a una posición de stack frame, equivalente a una variable mutable de C. Costo cero comparado con el código imperativo equivalente.

`Mutable`: el efecto detrás de `Ref[T]` y `Array[T]`

`var` cubre celdas locales. Pero hay casos donde la memoria tiene que **sobrevivir al bloque**: un array que vas a devolver, una celda que pasas entre funciones, una estructura que comparten varias rutinas. Para esos casos kaikai trae dos tipos del stdlib, `Ref[T]` y `Array[T]`, y ambos viven bajo el efecto **`Mutable`**.

```
fn rellenar(n: Int) : Array[Int] {
  let a = Mutable.array_make(n, 0)
  var i = 0
  list.foreach([0..n], (_) => {
    a[i] := @i * 2
    i := @i + 1
  })
  a
}
```

- **`Mutable.array_make(n, init)`** crea un array de tamaño `n` con valor inicial `init`.
- **`a[i]`** lee la posición `i`. Azúcar para `Mutable.array_get(a, i)`.
- **`a[i] := v`** escribe la posición `i`. Azúcar para `Mutable.array_set(a, i, v)`.

`Ref[T]` es la versión de una sola celda: `Mutable.ref_make(v)`, `Mutable.ref_get(r)`, `Mutable.ref_set(r, v)`. No tiene azúcar de indexación, pero el resto es paralelo a `Array[T]`.

Mira la firma de `rellenar`: dice `: Array[Int]`, **sin** **`Mutable`**. ¿Por qué, si la función claramente muta?

Porque `Mutable` sigue la disciplina de **efectos observables**: el efecto aparece en la firma solo cuando la mutación es **visible para quien llama**. Y aquí no lo es. El array se crea adentro, se llena adentro, y se devuelve cuando ya está listo. Quien recibe el array obtiene un valor ya armado, no observa ninguna mutación.

Cuándo `Mutable` se vuelve visible

Si la mutación es **observable**, el efecto aparece en la firma:

```
fn rellenar_en_sitio(a: Array[Int]) : Unit / Mutable {
  let n = Mutable.array_length(a)
  var i = 0
  list.foreach([0..n], (_) => {
    a[i] := @i * 2
    i := @i + 1
  })
}
```

Aquí `a` llega de afuera. Quien llama tiene una referencia al mismo array que estamos modificando. La mutación es visible para el caller, y la firma debe declararlo.

La regla del cap. 12 §12.3 se aplica igual que con cualquier otro efecto: el sistema de tipos garantiza que toda función que produce un efecto observable lo declara. Las "asignaciones secretas" no existen.

`Mutable` versus `State[T]`

Ambos representan estado mutable. ¿Cuándo conviene cada uno?

- **`var` (que es `State[T]`)**: la celda vive dentro del bloque. No tiene que sobrevivir a la función, no se pasa a otras rutinas, solo es un acumulador o contador local. La firma queda limpia.
- **`Mutable` con `Ref[T]` o `Array[T]`**: la memoria sobrevive al bloque o se comparte entre funciones. Aparece en la firma cuando la mutación es observable para quien llama.

Si pasas un `Ref[T]` o un `Array[T]` como argumento, o lo devuelves después de mutarlo, estás en territorio de `Mutable`. Si solo necesitas un contador local, es `var`.

12.8 Componer efectos: handlers anidados

Las funciones reales usan más de un efecto. Una que logguea y acumula puede declarar `/ Log + State[Int]`, y en `main` la manejas con dos `handle`s anidados:

```
fn acumular(xs: [Int]) : Int / Log + State[Int] {
  list.foreach(xs, (x) => {
    Log.log("sumando #{x}")
    State.set(State.get() + x)
  })
  State.get()
}

fn main() : Unit / Stdout {
  let total = handle {
    handle {
      acumular([10, 20, 30])
    } with State[Int](0) {
      get(resume)    -> resume(state)
      set(v, resume) -> resume((), v)
```

```
      return(x)      -> x
    }
  } with Log {
    log(msg, resume) -> {
      println("[LOG] " ++ msg)
      resume(())
    }
  }
  println("total = #{total}")
}
```

El orden de anidación importa **cuando los handlers interactúan**. Aquí no interactúan: `State` solo lee y escribe su estado, `Log` solo imprime. Cualquiera de los dos órdenes funciona. Pero si tuvieras `Fail` adentro de `State`, el orden decide si el estado sobrevive a una falla (Fail externo) o se descarta (Fail interno). Cada combinación tiene una semántica explícita y verificable.

Esa es una diferencia profunda con `try/catch + variables globales`: ahí el orden es implícito y depende del runtime. Acá es explícito y lo decides en la firma de los `handle`.

12.9 Alias de filas de efectos

Cuando una combinación aparece muchas veces, le das un nombre con `type`:

```
effect Log   { log(msg: String) : Unit }
effect Audit { audit(usuario: String, accion: String) : Unit }

type Tracing = Log + Audit
```

A partir de ahí, escribir `: Unit / Tracing` es lo mismo que escribir `: Unit / Log + Audit`. El alias es **transparente**: no introduce un efecto nuevo, solo abrevia la fila.

```
fn realizar_compra(usuario: String, monto: Int) : Unit / Tracing {
  Log.log("inicio compra")
  Audit.audit(usuario, "compra($#{monto})")
  Log.log("fin compra")
}
```

Una restricción: los alias deben ser **cerrados**. No puedes escribir `type WithIo[e] = Io + e` (con variable de fila). Esa restricción evita complicaciones en la unificación que el compilador no necesita pagar.

12.10 Default handlers: el efecto trae el suyo

Hasta aquí cada `handle` que vimos lo escribimos a mano. Pero hay efectos donde una de las implementaciones es tan obvia que pedirle al usuario que la escriba cada vez es ceremonia pura: si tu efecto es `Log` y la implementación "razonable" imprime con timestamp, querrías que esa implementación venga con el efecto.

Kaikai permite declarar un **bloque** `default {}` dentro de la declaración del efecto. Es exactamente lo mismo que un `handle ... with`, pero vive al lado de las operaciones y el compilador lo instala alrededor de `main` cuando nadie maneja el efecto a mano.

```
effect Log {
  info(msg: String) : Unit
  warn(msg: String) : Unit

  default {
    info(msg, resume) -> $extern_handler("kai_default_log_info")
    warn(msg, resume) -> $extern_handler("kai_default_log_warn")
  }
}
```

Las cláusulas del bloque `default` tienen la **misma forma** que las de un `handle`: nombre de operación, parámetros, `resume`, flecha, cuerpo. Lo que cambia es dónde viven y quién las dispara. Si `main()` declara `: Unit / Log` y no hay ningún `handle ... with Log` envolviéndolo, el compilador genera código equivalente a:

```
handle {
  main_original()
} with Log {
  info(msg, resume) -> ...   # las cláusulas del default
  warn(msg, resume) -> ...
}
```

El usuario no escribe ese wrapping. El compilador lo deriva del bloque `default` y lo emite al entrar al programa.

`$extern_handler`: el sigil y el puente a C

El cuerpo de cada cláusula del ejemplo anterior es `$extern_handler("kai_default_log_info")`. Eso necesita explicación.

`$` es un **sigil**: un carácter que marca una forma sintáctica especial. En kaikai introduce un **compiler intrinsic**, una construcción que el compilador resuelve directamente en vez de buscar una función definida en código kaikai. La forma general es `$nombre(args)`. Hoy hay uno solo: `$extern_handler`. Mañana puede haber más; el sigil queda reservado para esa categoría.

`$extern_handler("kai_default_log_info")` significa: "el cuerpo de esta cláusula es una llamada al símbolo C `kai_default_log_info`". Cuando el efecto se dispara, el compilador no busca una función kaikai que se llame así; emite directo una llamada al runtime de C que está enlazado al programa.

Esto es el puente entre los efectos algebraicos de alto nivel y el mundo concreto: archivos, sockets, syscalls. Las primitivas del sistema operativo viven en C; los efectos viven en kaikai; `$extern_handler` los conecta.

Cuándo dispara el default y cuándo no

La regla de búsqueda es la misma de §12.4 pero con un escalón extra al final:

1. El `handle ... with Eff` más cercano que cubre la operación gana.
2. Si no hay handle envolvente y la operación está en la fila de `main`, el compilador instala el default del efecto.
3. Si ni siquiera el default cubre la operación, el compilador rechaza el programa al typecheckar `main`.

Mira el detalle: el default **solo** se instala cuando la operación escaparía a `main`. Si tu función está dentro de un `handle`, gana el handle, no el default. No hay ambigüedad ni precedencia sorpresiva: lo más cercano siempre gana.

Volver al ejemplo desde otro ángulo

Esto explica por qué `println` compila sin que cada firma cargue `/ Stdout`:

```
fn hola() {
  println("hola")
}

fn main() {
  hola()
}
```

`Stdout` viene del stdlib con un bloque `default` cuyas cláusulas llaman a `$extern_handler("kai_default_stdout_print")` y similares. El símbolo C escribe al `stdout` real del proceso. Como `main` no maneja `Stdout`, el compilador instala ese default y el programa imprime.

Cuando quieras controlar la salida (silenciar en tests, redirigir a archivo, capturar para asertar contra ella), pones tu propio `handle ... with Stdout` y dentro del bloque el default del runtime no participa. **Lo más cercano gana**: el `handle` que escribiste está más cerca que el wrapping implícito al entrar a `main`.

Tu propio default: el ejemplo completo

Si declaras un efecto y lo equipas con un default, los programas que solo usan `main` se ven igual de simples:

```
effect MyLog {
  info(msg: String) : Unit
  default {
    info(msg, resume) -> $extern_handler("kai_default_log_info")
  }
}

fn greet(name: String) : Unit / MyLog {
  MyLog.info("hola, " ++ name)
}

fn main() : Unit / Stdout {
  let r = handle {
    MyLog.info("hello from extern_handler")
    7
  } with MyLog {
```

```
    info(msg, resume) -> resume(())   # silencia adentro del handle
  }
  print("result: #{int_to_string(r)}")
}
```

Adentro del `handle ... with MyLog`, las cláusulas explícitas ganan: el `info` queda silenciado. Si en otro `main` no hubiera ese `handle`, el default dispararía e imprimiría vía el runtime de C. La función `greet` no se entera: para ella, `MyLog` es lo que sea que el contexto haya decidido.

Cuándo no hay default — el efecto sin red

No todos los efectos traen `default`. `Fail` es el contraejemplo claro: si una operación puede abortar el programa, no quieres que "olvidarse de manejarla" sea legal. La declaración pública de `Fail` (apéndice D) no tiene bloque `default`, y por eso una función que produce `/ Fail` obliga a ser manejada en algún lado antes de `main`, o el compilador rechaza con un mensaje claro.

Lo mismo vale para `State[T]`, `Reader[T]`, `Writer[W]`: efectos genéricos en los que **no existe** una implementación razonable sin contexto, así que pedirle al usuario que la escriba no es ceremonia, es disciplina.

La regla mental: un efecto trae `default` cuando hay **una sola** implementación obvia (escribir a `stdout`, leer del reloj del sistema, generar números pseudo-aleatorios). Si "razonable" depende del programa, no hay default y el usuario lo provee.

Función envoltorio: la alternativa cuando no hay default

Cuando un efecto no trae `default`, o cuando el default existe pero tu programa siempre quiere otro, el patrón idiomático es una **función envoltorio**:

```
fn with_test_log[A](body: () -> A / MyLog) : A {
  handle {
    body()
  } with MyLog {
    info(msg, resume) -> resume(())   # silencia en tests
  }
}

fn main() {
  with_test_log { ->
    greet("kaikai")
    greet("ada")
  }
}
```

Es lo que el stdlib usa para construcciones como `try { body }` y `with_state(0) { body }`. La diferencia con un default es que la envoltura es **explícita en el código**: quien lee `main` ve la línea, abre la función, sabe qué hace. Un default vive en la declaración del efecto.

Si tu efecto tiene un default razonable para producción y uno distinto para tests, expón los dos como funciones envoltorio (`with_test_log`, `with_quiet_log`) y deja que `main` use el default. Quien escribe tests llama a la envoltorio.

12.11 Los handlers del stdlib son código kaikai

Cuando un programa `println("hola")` simplemente funciona, es fácil imaginar que el compilador trae un caso especial para `Stdout`. No es así. Los handlers que el runtime instala alrededor de `main` para `Stdout`, `Stdin`, `Random`, `Clock`, `File`, `Env`, `NetTcp`, y los demás están escritos en **stdlib kaikai normal**: cada uno es un `effect ... { ops; default { ... } }` que usa el mismo sigil `$extern_handler` que tú usarías para conectar tu efecto con C.

```
# stdlib/io/console.kai (forma esquemática)
effect Stdout {
  print(s: String) : Unit
  default {
    print(s, resume) -> $extern_handler("kai_default_stdout_print")
  }
}
```

El compilador no conoce a `Stdout` por su nombre. Conoce **bloques `default` y `$extern_handler`**. Para `Stdout`, instala el default igual que para tu `MyLog`: caminando el AST de la declaración del efecto, no leyendo una tabla hardcoded.

Esto no fue siempre así. Hasta la versión 0.55, el compilador traía tablas internas con los nombres `default_stdout_setup`, `default_random_shims`, etc., una entrada por cada efecto del stdlib. La trilogía #533 (PRs #551, #559, #561 en `kaikai-org/kaikai`) migró los diecisiete efectos builtin a `default {}` blocks declarados en stdlib y borró las tablas. El motivo no es estético: es que el AST se vuelve la única fuente de verdad, y los efectos de usuario obtienen exactamente las mismas garantías que los del stdlib. Si tu efecto declara `default {}` con `$extern_handler`, el compilador lo instala como builtin.

La consecuencia práctica: **puedes leer cómo está implementado `Stdout`**. Está en `stdlib/io/console.kai` (o el archivo análogo de la versión que estés usando). Es código kaikai como el tuyo. Si te aparece una duda sobre semántica del default — "¿qué pasa si el pipe está cerrado?", "¿quién captura `EPIPE`?" — la respuesta vive en la cláusula `print(s, resume) -> ...` o en el símbolo C al que puentea. No hay un comportamiento secreto del runtime separado del código que puedes leer.

Vale repetir la regla para amarrar el modelo: el compilador resuelve un efecto buscando, en orden, (1) el `handle ... with` más cercano, (2) el `default {}` block del efecto si la operación escapa a `main`, (3) error de compilación. Los handlers del stdlib no son una cuarta categoría; son instancias de (2).

Por qué el sigil tiene un nombre raro

`$extern_handler` puede sonar largo. La razón es que el sigil es un sistema, no una sola operación. La trilogía #533 introdujo `$` como prefijo para una **familia** de intrinsics; `$extern_handler` es el primero. Si más adelante kaikai necesita exponer otros puentes al runtime — pedir el `errno` actual, llamar a un símbolo de plataforma específica — vivirán bajo el mismo sigil con nombres descriptivos: `$os_name`, `$panic_with_trace`, lo que sea. Reservar `$<ident>(args)` deja la puerta abierta sin reabrir el debate sintáctico cada vez.

Para tu día a día: si nunca conectas un efecto a C, nunca vas a escribir `$extern_handler`. Pero cuando lo veas en stdlib sabes qué es: una cláusula que cede el cuerpo a un símbolo del runtime, declarada con la misma sintaxis que cualquier otro handler.

12.12 Caso de estudio: procesador de configuración

Cerramos con un ejemplo que mezcla los tres patrones que vimos: logueo, estado, fallo. El programa procesa una lista de líneas con formato `clave=valor`, las parsea, registra cada paso, cuenta cuántas pasaron, y aborta si alguna línea no tiene el formato esperado.

```
effect Log {
  log(msg: String) : Unit
}

effect State[T] {
  get() : T
  set(v: T) : Unit
}

effect Fail {
  fail(motivo: String) : Nothing
}

type Entrada = { clave: String, valor: String }

fn parsear(linea: String) : Entrada / Fail {
  match string.split(linea, "=") {
    [c, v] -> Entrada { clave: c, valor: v }
    _      -> Fail.fail("línea inválida: '#{linea}'")
  }
}

fn procesar(lineas: [String]) : Int / Log + State[Int] + Fail {
  list.foreach(lineas, (l) => {
    let e = parsear(l)
    Log.log("#{e.clave} = #{e.valor}")
    State.set(State.get() + 1)
  })
  State.get()
}
```

`procesar` declara los tres efectos en su firma y los usa libremente: parsea (puede fallar), registra (loggea), acumula (estado). Pero no decide nada sobre el contexto en que corre.

En `main`, los tres handlers anidados deciden:

```
fn main() : Unit / Stdout {
  let n = handle {
    handle {
      handle {
        procesar(["nombre=ada", "edad=42", "rol=admin"])
      } with State[Int](0) {
```

```
        get(resume)    -> resume(state)
        set(v, resume) -> resume((), v)
        return(x)      -> x
      }
    } with Log {
      log(msg, resume) -> {
        println("[LOG] " ++ msg)
        resume(())
      }
    }
  } with Fail {
    fail(motivo, resume) -> {
      println("error: " ++ motivo)
      0 - 1
    }
  }
  println("entradas procesadas: #{n}")
}
```

Salida:

```
$ kai run ejemplos/cap12/08_parser_config.kai
[LOG] nombre = ada
[LOG] edad = 42
[LOG] rol = admin
entradas procesadas: 3
```

Y si una línea es inválida, el `Fail` exterior la atrapa, imprime el motivo, y `n` queda en `-1`. El `Log` y el `State` interiores ya emitieron lo que alcanzaron antes del fallo.

¿Por qué este es un buen ejemplo para terminar el capítulo? Porque muestra los tres patrones cooperando, cada uno aportando algo distinto, y porque la función `procesar` es **directamente testeable**: sin tocar archivos, sin tocar IO, sin mocks. En un test, los tres handlers tienen otras implementaciones: el `Log` acumula en una lista en vez de imprimir, el `Fail` propaga en un `Result`, el `State` parte del valor que el test quiera.

12.13 Filosofía: tres ideas que vale recordar

Si esto te parece muchas piezas, vale fijar las tres ideas que todo lo demás sostiene:

1. **Los efectos son visibles en el tipo.** Si una función puede fallar, suspenderse, mutar, o llamar a IO, su firma lo dice. Sin excepciones invisibles, sin `async` infeccioso, sin dependencias ocultas.
2. **El handler decide qué pasa.** El cuerpo de una función declara que necesita una capacidad. El handler en el contexto decide cómo materializarla. Ese desacople es lo que reemplaza inyección de dependencias, mocking, configuración global.
3. **Costo cero cuando no se usa.** El compilador resuelve los handlers en tiempo de compilación cuando puede (la mayoría de los casos), y el código generado es tan rápido como si hubieras escrito un `if` directo. No hay sobrecarga estructural por tener

efectos en el tipo. Es la misma promesa de las unidades de medida y los contratos: información rica en el tipo, código eficiente abajo.

Los efectos algebraicos vienen de la academia (Pretnar, Plotkin, Power) y aparecieron en lenguajes como Koka, Eff y Effekt antes que en kaikai. Lo que kaikai aporta es una sintaxis legible (la notación `/ Eff` en la firma), una integración con el resto del lenguaje (filas en vez de listas, alias), y un modelo de defaults que no tiene casos especiales: los handlers del stdlib están declarados en kaikai con la misma forma que los tuyos. Pero la idea de base es vieja y sólida.

Si después de este capítulo todavía no estás cómodo, no te preocupes. Los efectos son la pieza que más tiempo toma asentar. Volverás a leer este capítulo varias veces. Cada lectura agarra una capa más.

Ejercicios

12.1. Escribe un efecto `Clock` con una operación `now() : Int` (milisegundos desde el inicio). Escribe una función `medir` que ejecute un bloque y devuelva cuánto tiempo tomó usando `Clock`. Después escribe dos handlers: uno real (consulta el reloj del sistema) y uno simulado (avanza un contador). ¿Para qué sirve el segundo?

12.2. Modifica `suma` de §12.6 para que devuelva tanto el total como la cantidad de elementos sumados, sin agregar parámetros. Pista: cambia `return(x) -> x`.

12.3. El caso de estudio §12.12 imprime con `[LOG]` cada entrada. Cambia el handler de `Log` para que en vez de imprimir, acumule los mensajes en una lista y los devuelva como parte del resultado final, junto con `n`. Pista: necesitas otro `State`.

12.4. Escribe `fn contar_pares(xs: [Int]) : Int` de dos formas: una con `var` y `list.foreach`, otra sin `var`, usando `list.filter` y `list.length`. ¿Cuál te parece más clara? ¿Por qué la versión con `var` no agrega efectos a la firma?

12.5. Construye un efecto `Choice` con una operación `choose(opciones: [Int]) : Int` que entrega "alguna" de las opciones. Escribe un handler que siempre elija la primera y otro que elija la última. ¿Cómo cambiaría la implementación si quisieras un handler que explore **todas** las opciones (backtracking)? Pista: necesitarías llamar a `resume` más de una vez. Eso es *multi-shot* y vive bajo `resume_multishot`.

12.6. Toma un programa que tengas en otro lenguaje donde uses inyección de dependencias para mockear servicios en tests. Anota en pseudocódigo qué efectos declararías y cómo serían los handlers de tests vs producción. ¿Cuánto código del programa original sobrevive sin cambios?

12.7. Investiga la diferencia entre `resume` (one-shot) y `resume_multishot` en la doc del lenguaje. ¿Por qué kaikai hace que el caso common sea cheap y obliga a marcar explícitamente el caso caro?

12.8. Declara un efecto `MyLog` con una operación `info(msg: String) : Unit` y un bloque `default { }` que puentee a un símbolo C ficticio `my_log_info_to_stderr` vía `$extern_handler`. Después escribe una función envoltorio `with_quiet_log` que silencie los mensajes. Un `main` sin envoltorio dispara el default; un `main` envuelto en `with_quiet_log` no. Compara con cómo harías lo mismo en un lenguaje con inyección de dependencias clásica.

12.9. ¿Por qué `Fail` no trae bloque `default{}`? Anota tres efectos hipotéticos (los tuyos o de stdlib que imagines) y para cada uno decide si llevaría default. Argumenta en una línea por qué sí o por qué no.

12.10. Lee la declaración de `Stdout` en `stdlib/io/console.kai` del repositorio del lenguaje. ¿Qué hace la cláusula del default cuando el pipe está cerrado (`EPIPE`)? ¿Dónde vive esa lógica: en kaikai o en el símbolo C al que puentea?

12.11. El capítulo 13 cubre fibras: tareas concurrentes manejadas como efectos. Anota antes de leerlo: ¿qué operaciones tendría que tener un efecto `Spawn`? ¿Qué decisión tomarías como handler cuando una fibra hija aborta?

Capítulo 13 · Concurrencia y memoria

La concurrencia es donde la mayoría de los lenguajes acumulan deuda. Threads con shared memory llevan a races que aparecen una vez al mes y se arreglan tres veces; `async` / `await` introduce colores de funciones; los actores corrigen lo anterior pero históricamente vienen con GC y un runtime pesado.

kaikai apuesta por una combinación poco usual: **fibras cooperativas + memoria por fibra + Perceus**. La estructura tiene tres consecuencias que valen la pena nombrar antes de la sintaxis:

- **No hay shared memory entre fibras.** Cada fibra tiene su propio heap. Lo que una pasa a otra se copia o se mueve. Adiós a las data races por construcción.
- **No hay GC ni borrow checker.** Perceus libera la memoria cuando el último uso de cada valor termina, sin un colector asincrónico y sin pedirle al programador que anote lifetimes. El compilador descubre dónde poner los `free`s analizando el programa.
- **La concurrencia es un efecto.** `spawn` es una operación de un efecto `Spawn`, no una palabra clave. Crear y esperar fibras se compone con el resto del sistema (`State`, `Fail`, `Cancel`) usando la maquinaria del cap. 12.

Vamos por partes.

13.1 El modelo: fibras aisladas

Una **fibra** es una unidad de ejecución parecida a un thread, pero mucho más liviana: del orden de cientos de bytes en vez de megabytes. Una aplicación kaikai puede tener miles o cientos de miles de fibras vivas sin reventar.

Las fibras son **cooperativas**. Cada una corre hasta que llega a un **punto de yield**: una llamada que voluntariamente le pasa el control al scheduler. Los puntos de yield son explícitos:

- `spawn.yield()`: "ya pude correr un rato, prueba otra".
- `spawn.await(f)`: "espera a que termine la fibra `f`".
- Las operaciones de IO que el scheduler intercepta (lecturas de red, sleep, etc.).

Sin yield, una fibra corre hasta terminar. Eso es **determinismo local**: dentro de un bloque sin yields, sabes exactamente qué pasa. Comparado con threads preemptivas, esto te quita una clase entera de bugs: no hay carrera sobre datos que toques entre dos yields porque nadie te va a interrumpir.

A cambio, una fibra que nunca cede el control bloquea a todas las demás. Es responsabilidad del programador poner yields donde haga sentido. En la práctica, las llamadas a IO ya los traen, y el único caso donde hay que pensar en yields manuales es en bucles puros de CPU intenso.

Memoria por fibra

Cada fibra tiene su **propio heap**. Cuando una fibra crea un record, una lista, un closure, el espacio sale de ese heap. La otra fibra no puede tocarlo: ni leerlo, ni escribirlo. El sistema de tipos lo garantiza.

¿Cómo se comunican dos fibras entonces? Pasándose valores. Cuando una fibra envía un mensaje a otra (vía mailbox de actor o vía el resultado de un `await`), el valor se copia al heap de la fibra receptora. Para tipos pequeños esto es trivial; para estructuras grandes, kaikai usa Perceus para mover en vez de copiar cuando el emisor ya no va a usar el valor.

Lo importante es la garantía: **no hay manera de que dos fibras tengan un puntero al mismo objeto**. Las data races, los problemas de visibility de memoria, los bugs de cache coherence: todo lo que en threads tradicionales necesita lectura/escritura con `Atomic` o locks no existe acá. La concurrencia es por mensajes, no por memoria compartida.

13.2 Perceus en una página

¿Cómo se libera la memoria? Sin GC y sin borrow checker, hay un tercer enfoque: **reference counting estricto basado en Perceus** (Lorenz, Leijen, Reinking, 2021).

La idea es que el compilador analiza cada función para descubrir en qué punto cada valor deja de ser usado. En ese punto inserta una instrucción que decrementa el contador de referencias del valor: si llega a cero, se libera; si no, se queda para otro uso.

```
fn ejemplo(xs: [Int]) : Int {
  let n = list.length(xs)   # primer uso de xs
  let s = list.sum(xs)      # último uso de xs: aquí se "consume"
  s + n                     # xs ya no existe; n y s sí
}
```

A diferencia de un GC:

- **No hay pausa.** El liberado es síncrono, predecible, parte del código generado.
- **No hay overhead asincrónico.** El compilador conoce el ciclo de vida exacto de cada valor.
- **No hay hilo aparte.** El scheduler no compite con un colector.

A diferencia de un borrow checker:

- **No hay anotaciones de lifetime.** El programador no escribe `'a` ni `&` ni `mut`.

- **No hay limitaciones de pattern de uso.** Si necesitas dos referencias al mismo valor, el compilador inserta los increments y decrements necesarios.

¿El costo? Cuando un valor se usa varias veces, los contadores se mueven. Para valores muy compartidos esto puede agregar overhead, y Perceus tiene optimizaciones agresivas para minimizarlo (reuse in place: si un valor está por liberarse y se necesita uno del mismo shape inmediatamente después, se reusa la misma memoria sin tocar el contador). En la práctica el costo es bajo y predecible.

Por qué esto importa para concurrencia: Perceus funciona por fibra. Cada fibra tiene sus propios contadores, sus propios liberados. No hay sincronización entre fibras para ningún contador: nunca dos fibras se pasan punteros al mismo valor con contador compartido. Esa es la razón por la que el modelo "fibras aisladas" cierra cuando se le agrega Perceus: el mismo invariante que protege contra data races también simplifica el RC.

13.3 Crear y esperar fibras: las operaciones básicas

La forma más simple de usar fibras es con `spawn.spawn` y `spawn.await`:

```
import spawn

fn worker(tag: String, n: Int) : Unit / Stdout + Spawn {
  if n > 0 {
    println(tag)
    spawn.yield()
    worker(tag, n - 1)
  }
}

fn main() {
  let f = spawn.spawn(() => worker("B", 3))
  worker("A", 3)
  spawn.await(f)
}
```

Salida:

```
$ kai run ejemplos/cap13/01_dos_fibras.kai
A
B
A
B
A
B
```

Lectura literal:

- `import spawn` trae las operaciones de fibras.
- `spawn.spawn(() => worker("B", 3))` crea una fibra nueva que va a correr el lambda cuando el scheduler la elija.
- `worker("A", 3)` corre en la fibra actual (la del `main`).
- `spawn.yield()` dentro de `worker` cede el control. Cada vuelta, la otra fibra toma su turno.

- `spawn.await(f)` espera a que `f` termine antes de que `main` retorne.

`Spawn` aparece en la firma de `worker` porque la función llama a `spawn.yield()`, que es una operación de `Spawn`. La fila contagia hacia arriba como con cualquier efecto del cap. 12.

Por qué los yields son explícitos

En lenguajes con threads preemptivas (Java, Go, Rust con `std::thread`), el scheduler puede interrumpir un thread en cualquier instrucción. Eso obliga a programar como si cualquier línea pudiera ser interrumpida por otra fibra modificando datos compartidos.

En kaikai, **una fibra sigue corriendo hasta que llega a un punto de yield**. Entre yields, tienes determinismo local: si modificas un valor local, nadie más lo va a tocar hasta que tú cedas el control. Esto reduce mucho la carga cognitiva.

A cambio, tienes que **acordarte de poner los yields**. La regla mental es: si tu función tiene un bucle largo de puro cómputo, agrega un `spawn.yield()` cada cierto número de iteraciones. Las funciones de IO ya yieldan por dentro.

13.4 Nurseries: concurrencia estructurada

`spawn.spawn` + `spawn.await` funciona, pero tiene un problema: si te olvidas del `await`, la fibra se queda viva más allá del scope donde la creaste. Y si esa fibra falla, te enteras tarde o no te enteras.

Las **nurseries** atan las fibras a un scope léxico. Una fibra solo puede vivir dentro de un nursery, y el nursery espera a todas sus hijas antes de salir.

```
import spawn

fn worker(tag: String, n: Int) : Unit / Stdout + Spawn {
  if n > 0 {
    println(tag)
    spawn.yield()
    worker(tag, n - 1)
  }
}

fn main() : Unit / Stdout + Spawn + Cancel {
  nursery { n ->
    let a = n.spawn(() => worker("A", 3))
    let b = n.spawn(() => worker("B", 3))
    n.await(a)
    n.await(b)
  }
}
```

`nursery { n -> ... }` abre un scope. Adentro, `n` es la capacidad para crear y esperar fibras:

- `n.spawn(f)` crea una fibra hija. Devuelve un `Fiber[T]` donde `T` es el tipo que `f` devuelve.
- `n.await(f)` espera a esa fibra y devuelve su valor.
- `n.select([a, b, ...])` espera a que cualquiera termine y cancela las demás.
- `n.cancel(f)` cancela una fibra específica.

- `n.cancel_all()` cancela todas las hijas.

Lo que el nursery garantiza:

- **Al salir del bloque, todas las hijas terminaron.** No hay fugas: una fibra no sobrevive al `nursery` que la creó.
- **Si una hija falla con un efecto no manejado, las demás se cancelan.** El nursery acumula la causa y la re-lanza.
- **Si el nursery se cancela desde afuera, propaga la cancelación a todas sus hijas.**

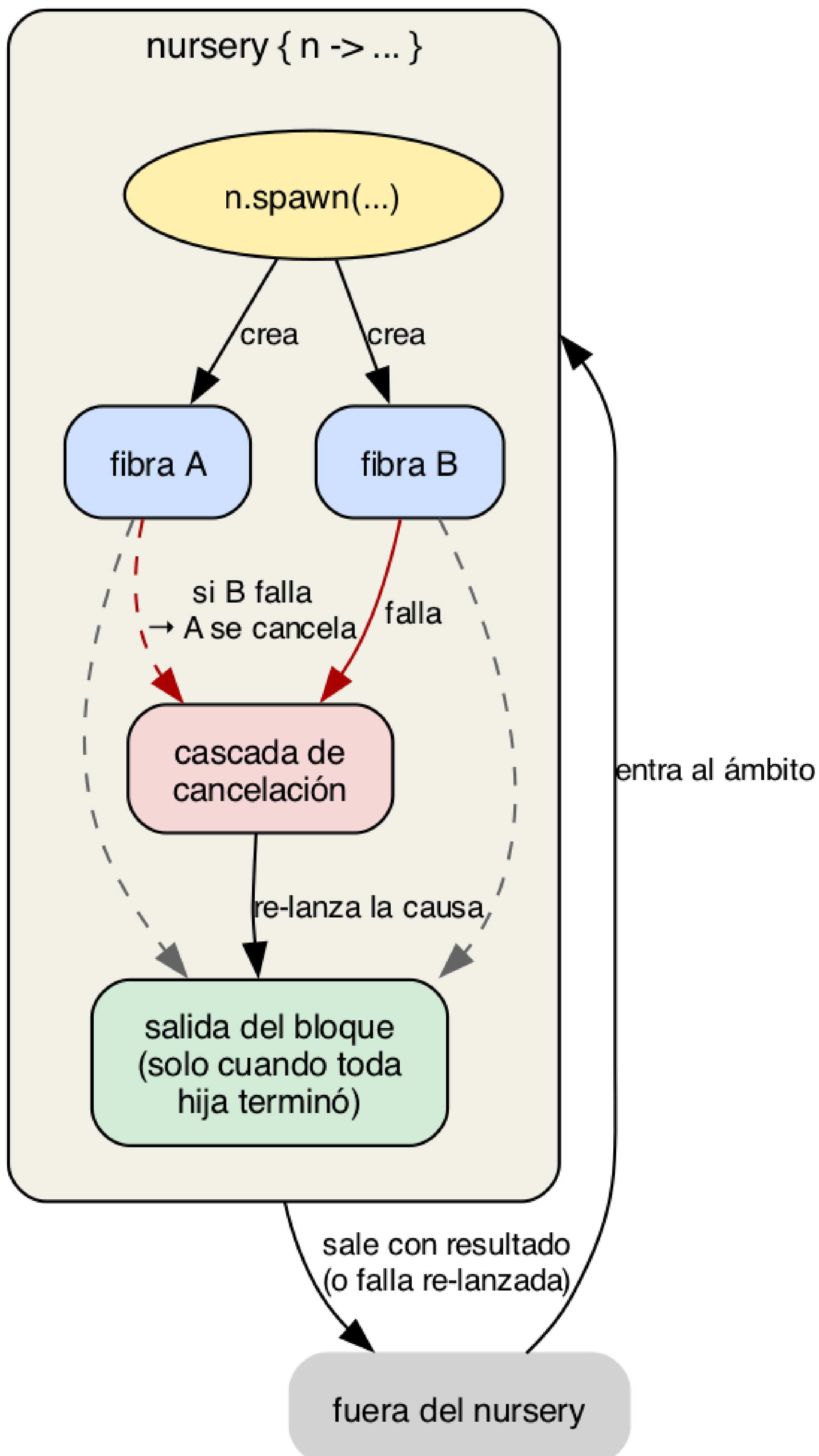
nursery { n -> ... }
n.spawn(...)
crea
crea
fibra A
fibra B
si B falla
→ A se cancela
falla
cascada de
cancelación
re-lanza la causa
salida del bloque
(solo cuando toda
hija terminó)
entra al ámbito
sale con resultado
(o falla re-lanzada)
fuera del nursery

Figura 13.1 · *Concurrencia estructurada en una imagen. El nursery es un ámbito léxico; las fibras hijas viven adentro; nada se escapa. Si una hija falla, el nursery cancela al resto antes de re-lanzar; si al padre lo cancelan desde afuera, la cascada baja.*

Esto se llama **concurrencia estructurada**. La idea es de Nathaniel Smith en su ensayo *"Notes on structured concurrency, or: Go statement considered harmful"* (2018), y aparece también en Trio, Kotlin coroutines, Swift y OCaml 5 Eio. La forma de kaikai integra el patrón en el sistema de efectos: la capacidad `Spawn` solo está disponible dentro de un nursery, y eso es lo que el sistema de tipos exige.

Por qué `Cancel` aparece en la firma de `main`

Fíjate que `main` declara `/ Stdout + Spawn + Cancel`. ¿Por qué `Cancel`? Porque cada `spawn`, `await` y `select` es un punto de yield, y todo punto de yield puede recibir una `Cancel.raise()` desde el scheduler (si alguien cancela el nursery desde afuera, o si una fibra hermana falla). Toda función que use `Spawn` carga implícitamente `Cancel`.

`nursery` es azúcar sobre un `handle`

`nursery { n -> ... }` parece una palabra clave del lenguaje, pero no lo es. Las fibras son **un efecto** llamado `Spawn`, con esta declaración en el stdlib:

```
effect Spawn {
  spawn[T, e](f: () -> T / e) : Fiber[T]
  await[T](f: Fiber[T])       : T
  select[T](fs: [Fiber[T]])   : T
  yield()                     : Unit
  cancel[T](f: Fiber[T])      : Unit
}
```

Es un efecto ordinario, declarado igual que `Log` o `State[T]` del cap. 12. Y `nursery { n -> body }` se reescribe en tiempo de compilación a `handle { body } with Spawn as n { ... }`, con un handler interno que gestiona el árbol de fibras hijas, espera a las pendientes al salir y propaga fallos.

Eso significa que el lenguaje core **no tiene primitivas de concurrencia**: tiene efectos. Las fibras, los nurseries, la cancelación son una biblioteca construida sobre dos efectos del stdlib (`Spawn` y `Cancel`). El cap. 14 va a hacer lo mismo para los actores (efecto `Actor[Msg]`), y el patrón se repite: lo distintivo de kaikai no es la lista de construcciones, sino que **todas son la misma construcción** (efectos algebraicos) con nombres distintos.

13.5 Cancelación cooperativa

La cancelación en kaikai es **cooperativa**: el scheduler no mata a una fibra de golpe. Le entrega una `Cancel.raise()` en el próximo punto de yield. La fibra puede:

- **Desenrollarse limpiamente.** Si no maneja `Cancel`, el unwinding la saca de cualquier `handle`, `nursery`, etc., y los handlers de cancelación por encima toman el control.
- **Manejar `Cancel` para hacer cleanup.** La fibra instala un handler de `Cancel` que corre el cleanup (cerrar archivos, soltar conexiones) y NO llama a `resume`, dejando que el unwinding continúe.

```
fn trabajador_largo(tag: String) : Unit / Stdout + Spawn + Cancel {
  handle {
    contar(tag, 0)
  } with Cancel {
    raise(resume) -> {
      println("#{tag}: cancelado, hago cleanup")
      # no llamamos a resume: la fibra se desenrolla
    }
  }
}

fn contar(tag: String, n: Int) : Unit / Stdout + Spawn + Cancel {
  println("#{tag}: #{n}")
  spawn.yield()
  contar(tag, n + 1)
}
```

`trabajador_largo` cuenta indefinidamente. Si el nursery la cancela, el handler de `Cancel` imprime el mensaje y la fibra sale. No se queda colgada, no aborta el proceso.

La clave conceptual: `Cancel.raise()` es la **operación**, y `handle ... with Cancel { ... }` es el handler. Mismo patrón del cap. 12: la cancelación es un efecto más, con un handler escrito por el usuario o instalado por el runtime.

13.6 Memoria mutable por fibra

El cap. 12 §12.7 cubrió `var` (celdas locales, azúcar sobre `State[T]`) y el efecto `Mutable` (que rige a `Ref[T]` y `Array[T]` cuando la mutación es observable). Toda esa maquinaria funciona igual que en código secuencial, con una sola adición que viene del modelo de fibras: **la memoria mutable vive en el heap de la fibra que la creó**.

Cuando una fibra crea un `Array[T]` o un `Ref[T]`, el espacio sale de su propio heap. Otra fibra no tiene cómo acceder a esa memoria: no hay punteros compartidos, no hay paso por referencia entre fibras. Si una fibra quiere darle un valor mutable a otra, lo manda por mailbox (cap. 14) y el runtime mueve el contenido al heap del receptor.

Esta es la pieza que hace que la mutación no introduzca data races en kaikai. En un lenguaje con threads y memoria compartida, un `Array[T]` mutable necesita locks o atomics para ser tocado desde varios hilos. En kaikai, el sistema de tipos garantiza que ningún `Array[T]` está siendo modificado por dos fibras al mismo tiempo, porque ningún `Array[T]` es accesible desde dos fibras al mismo tiempo. La aislación de memoria del §13.1 cubre también las celdas mutables.

13.7 Por qué las fibras no pueden escapar de su nursery

Un detalle del sistema de tipos: **`Fiber[T]` no es un valor movible**. No puedes devolverlo de una función, no puedes guardarlo en un `Option` o un record, no puedes pasarlo a otra fibra.

```
fn no_compila() : Fiber[Int] {      # ERROR
  nursery { n ->
    n.spawn(() => 42)              # no se puede devolver
  }
}
```

¿Por qué? Porque una fibra solo tiene sentido dentro del nursery que la creó. Si se permitiera devolverla, ¿quién la esperaría? ¿Quién la cancelaría si el nursery termina? La estructura se rompe.

Una lista de fibras dentro del mismo nursery es legal:

```
nursery { n ->
  let fibers = [1, 2, 3] | (x) => n.spawn(() => procesar(x))
  fibers | (f) => n.await(f)
}
```

Pero esa lista vive dentro del nursery. No puede salir.

Esto cierra el modelo: cada fibra tiene un padre conocido, cada fibra termina antes de que su padre termine, y el sistema de tipos lo garantiza al nivel sintáctico, no al nivel de convención. **No hay manera de que una fibra se quede huérfana.**

13.8 Caso de estudio: cola de tareas con pool de trabajadores

Cerramos con un patrón clásico: una cola de tareas servida por un pool de fibras trabajadoras. Cada trabajador toma la siguiente tarea, la procesa, y vuelve por otra. Cuando la cola se vacía, todos terminan.

```
import spawn

effect State[T] {
  get() : T
  set(v: T) : Unit
}

fn siguiente() : Option[String] / State[[String]] {
  match State.get() {
    []          -> None
    [h, ...rest] -> {
      State.set(rest)
      Some(h)
    }
  }
}

fn worker(id: Int) : Unit / Stdout + Spawn + State[[String]] {
  match siguiente() {
    None       -> println("worker #{id}: cola vacía, salgo")
    Some(tarea) -> {
      println("worker #{id}: procesando '#{tarea}'")
```

```
      spawn.yield()
      worker(id)
    }
  }
}
```

Hasta aquí, código puro de efectos: tres operaciones (`get`, `set`, `siguiente`), una recursión que termina cuando la cola se acaba. No hay locks, no hay atomics, no hay `Arc<Mutex<...>>`.

El `main` instala los handlers y arranca el pool:

```
fn main() : Unit / Stdout + Spawn + Cancel {
  handle {
    nursery { n ->
      let a = n.spawn(() => worker(1))
      let b = n.spawn(() => worker(2))
      let c = n.spawn(() => worker(3))
      n.await(a)
      n.await(b)
      n.await(c)
    }
  } with State[[String]](["alpha", "bravo", "charlie", "delta", "echo", "foxtrot"]) {
    get(resume)    -> resume(state)
    set(v, resume) -> resume((), v)
    return(x)      -> x
  }
  println("(todas las tareas procesadas)")
}
```

Salida:

```
$ kai run ejemplos/cap13/06_eco_concurrente.kai
worker 1: procesando 'alpha'
worker 2: procesando 'bravo'
worker 3: procesando 'charlie'
worker 1: procesando 'delta'
worker 2: procesando 'echo'
worker 3: procesando 'foxtrot'
worker 1: cola vacía, salgo
worker 2: cola vacía, salgo
worker 3: cola vacía, salgo
(todas las tareas procesadas)
```

Tres fibras se reparten seis tareas concurrentemente. Cada una accede a la "cola compartida" vía el efecto `State`, pero por dentro no hay shared memory: el handler de `State` vive en el `main`, y las operaciones de las fibras son mensajes a ese handler. La cola es serializada por construcción.

Concurrencia, no paralelismo

Vale ser preciso con una palabra. kaikai en v1 corre **un solo hilo del sistema operativo**: un único scheduler, una sola cola de fibras listas. Las fibras se intercalan cooperativa-

mente, pero nunca dos fibras están ejecutando instrucciones al mismo tiempo. Eso es **concurrencia**, no **paralelismo**.

¿Por qué importa? Porque si tu programa está limitado por CPU (cálculo numérico, compresión, renderizado), correrlo con cien fibras no lo va a hacer más rápido: van a turnarse en el mismo núcleo. Para problemas como esos, las fibras te dan estructura (forma natural de expresar trabajo concurrente, cancelación, timeouts) pero no aceleración.

Donde las fibras sí pagan en velocidad es cuando el cuello de botella es **IO**: leer archivos, esperar red, esperar mensajes. Mientras una fibra está bloqueada esperando bytes, otras fibras corren. El mismo núcleo aprovecha el tiempo que de otra manera estaría ocioso.

El paralelismo real (varios núcleos físicos trabajando a la vez) requiere multi-threading, que está fuera del alcance de v1. Cuando aterrice, será sobre el mismo modelo de actores y fibras: un scheduler por hilo, fibras cooperativas adentro, mensajes entre hilos. Por ahora, la garantía es que cualquier código que escribas hoy con fibras y actores va a seguir funcionando cuando el multi-threading llegue, solo más rápido en máquinas con varios núcleos.

13.9 Filosofía: dos invariantes que vale recordar

Si quieres recordar dos cosas del capítulo, que sean estas:

1. **Cada fibra tiene su propio heap.** No hay shared memory entre fibras. La comunicación es por mensajes (mailbox de actor, resultado de `await`). Las data races no existen por construcción, no por disciplina.
2. **Cada fibra tiene una vida atada a un scope léxico.** No puede escapar del `nursery` que la creó; el sistema de tipos rechaza cualquier intento. Si el padre termina, todas las hijas terminaron. Si una hija falla, las hermanas se cancelan.

Estos dos invariantes se sostienen mutuamente. La aislación de memoria es lo que permite que Perceus funcione por fibra sin sincronización. La estructura léxica es lo que permite liberar la memoria predeciblemente al final del scope. Cualquier modelo que rompa uno de los dos rompe el otro.

A cambio, hay clases enteras de bugs que no tienes que pensar:

- No hay `volatile` ni `Atomic` ni memory ordering.
- No hay `lock` ni `mutex` ni `RwLock`.
- No hay borrow checker, ni `'a` lifetimes, ni `Rc<RefCell<T>>`.
- No hay `async fn` ni `Future` ni colores de funciones.
- No hay GC pause-the-world.

Es un trade-off: pierdes la libertad de tener punteros arbitrarios entre fibras. Pero la ganancia (en seguridad, en predictibilidad, en simplicidad mental) es lo que justifica el modelo.

Ejercicios

13.1. Modifica el ejemplo §13.3 (dos fibras cooperativas) para que `worker("A")` haga 5 iteraciones y `worker("B")` haga

2. ¿Cómo cambia la salida? ¿Qué pasa si quitas los

`spawn.yield()` de uno solo de los dos workers?

13.2. Una fibra crea un `Array[Int]` localmente y lo modifica con `a[i] := v`. Después termina sin pasarlo a nadie. ¿Por qué este programa no introduce data races aunque otra fibra esté corriendo concurrentemente? Da el argumento en dos líneas, en términos del modelo de memoria por fibra de §13.1.

13.3. Implementa una función `with_timeout[T](ms: Int, f: () -> T / Spawn) : Option[T] / Spawn + Cancel + Time`. Usa `n.select` para correr `f` contra una fibra que hace `Time.sleep(ms)` y devuelve `None`. Pista: necesitas un tipo suma local para distinguir "completó" de "timeout".

13.4. En §13.8, el `State` es una cola FIFO sin preferencias. Modifica el ejemplo para que algunas tareas tengan prioridad alta y los workers las procesen primero. Pista: el `State` puede ser un record con dos listas.

13.5. Una fibra que entra a un bucle infinito sin `spawn.yield()` bloquea a todas las demás. Escribe ese código y observa qué pasa. Después agrega yields cada N iteraciones. ¿Cada cuántas? ¿Cómo decides el N?

13.6. En tu lenguaje habitual, busca un programa concurrente que escribiste o que mantienes. Cuenta cuántas líneas son "trabajo real" (la lógica del programa) versus cuántas son "concurrencia plumbing" (mutex, queues, atomics, async/await, callbacks). Estima qué porcentaje del código quedaría con el modelo de kaikai.

Capítulo 14 · Actores

El capítulo 13 mostró cómo arrancar fibras y coordinarlas vía un nursery. Eso resuelve la concurrencia interna: muchas unidades de trabajo dentro de un mismo programa, que se reparten el CPU cooperativamente.

Para muchos casos esa estructura alcanza. Pero hay un patrón que las fibras puras dejan incómodo, y vale la pena nombrarlo antes de la sintaxis.

Una fibra es un cómputo; un actor es un proceso vivo

Imagina dos tareas distintas:

- **Tarea A:** parsea un archivo grande y devuelve la lista de errores que encontró. Arranca, trabaja, termina, devuelve un valor.
- **Tarea B:** mantén una cache en memoria que responde consultas (`get(key)` y `put(key, value)`). Arranca, queda viva, responde mensajes mientras el programa exista, en algún momento termina cuando alguien le pide que se detenga.

Las dos son concurrentes en el sentido de que el programa principal puede seguir trabajando mientras pasan. Pero su forma es muy distinta.

La tarea A es un **cómputo**: tiene una entrada, produce un valor de salida, termina. Eso es una **fibra**. La creas con `spawn`, esperas su resultado con `await`, recibes el valor. Una vez devuelto el resultado, la fibra deja de existir.

```
let f = spawn.spawn(() => parsear_archivo("entrada.txt"))
# ... otro trabajo concurrente ...
let errores = spawn.await(f)   # un solo valor, y se acabó
```

La tarea B es un **proceso vivo**: no tiene un único valor de retorno, tiene una secuencia interminable de interacciones. Para esto kaikai trae el modelo de **actores**, heredado en espíritu de Erlang y BEAM.

```
let cache = spawn_actor(() => bucle_cache())
Actor.send(cache, Put("usuario:42", "ada"))
Actor.send(cache, Put("usuario:43", "turing"))
match Actor.receive() {
  Found(v) -> ...
```

```
  Missing -> ...
}
```

Un actor es **una fibra con un mailbox tipado encima**. La fibra es el sustrato (un hilo cooperativo de ejecución); el mailbox es lo que la hace un actor (un canal donde se acumulan mensajes para que la fibra los procese en orden).

Comparación lado a lado:

Aspecto	Fibra	Actor
¿Qué es?	Unidad de ejecución	Fibra con mailbox tipado
Comunicación	Un valor de retorno vía `await`	Mensajes vía `send` / `receive`
Ciclo de vida	Arranca, calcula, devuelve, muere	Arranca, queda en bucle procesando, muere cuando decide
Cómo se crea	`spawn.spawn` o `n.spawn`	`spawn_actor`
Cómo le hablas	`await` para obtener su `T`	`send` cualquier cantidad de veces
Cuándo elegirlo	Cálculo discreto concurrente	Servicio de larga vida, estado interno, consultas

La regla mental:

- **¿La tarea termina con un valor que el padre necesita?** Fibra.
- **¿La tarea vive y responde mensajes a varios clientes?** Actor.

Ambos modelos son **concurrentes pero no paralelos** en v1: el runtime corre un solo hilo del sistema, y fibras y actores se intercalan cooperativamente en él (cap. 13 §13.8 *Concurrencia, no paralelismo*). La ganancia es estructural y para cargas limitadas por IO, no aceleración multinúcleo.

Casos donde el actor es lo natural: un servidor de cache, un controlador de conexiones, un supervisor de procesos, un router de notificaciones, una cola de tareas, un actor logger. Casos donde la fibra basta: un cómputo que el padre quiere hacer en paralelo conceptual mientras hace otra cosa, un `with_timeout` que mide cuánto tarda algo, un map concurrente sobre una lista de IO.

Los actores no son primitivos del lenguaje

Otra cosa que vale fijar antes de la sintaxis: en kaikai, los actores no son una construcción del core. Son una **capa construida con efectos algebraicos**: el efecto `Actor[Msg]` declara las operaciones (`self`, `send`, `receive`); el stdlib provee funciones como `with_mailbox` y `spawn_actor` que instalan el handler de `Actor[Msg]` sobre una fibra ordinaria.

Es el mismo principio que con `nursery` del cap. 13: el lenguaje core tiene solo efectos; los patrones de uso (fibras, actores, supervisión) aparecen en bibliotecas que cualquier lector puede leer. Si después de este capítulo te preguntas cómo funciona `spawn_actor` por dentro, la respuesta es un `spawn.spawn` más un `handle ... with Actor[Msg]`.

14.1 `Actor[Msg]`: el efecto

Un actor es una fibra dentro de un `handle ... with Actor[Msg]` que le da acceso a tres operaciones. El efecto viene declarado en el stdlib:

```
# Declarado en stdlib/actor.kai, accesible vía `import actor`.
pub effect Actor[Msg] {
  self()                          : Pid[Msg]
  send(pid: Pid[Msg], msg: Msg)   : Unit / Cancel
  receive()                       : Msg / Cancel
}
```

- `self()` devuelve el `Pid` del actor actual. El `Pid` es el handle con el que otros le mandan mensajes.
- `send(pid, msg)` encola `msg` en el mailbox de `pid`. Si el mailbox está lleno, el comportamiento depende de la policy (lo veremos en §14.4).
- `receive()` saca el siguiente mensaje del mailbox del actor actual. Si no hay nada, la fibra se suspende hasta que llegue uno. Como suspende, es un punto de yield y carga `Cancel`.

`Msg` es el tipo concreto de los mensajes que ese actor recibe. **Un actor maneja un solo tipo de mensaje.** Si necesitas mezclar shapes, los unificas con un sum type:

```
type ServerMsg
  = Ping(Pid[Pong])
  | Stop
  | Tick
```

El actor sabe que solo va a recibir uno de esos tres constructores, y el `match` exhaustivo en `receive()` te garantiza que cubres todos los casos.

`Pid[Msg]`: handle tipado

Un `Pid[Msg]` es un identificador de mailbox. Tiene tipo, así que el compilador no te deja mandarle a un `Pid[Tarea]` un mensaje de tipo `Notificación`. Esta es la diferencia más fuerte con Erlang: en Erlang los PIDs son no tipados; en kaikai son específicos al tipo de mensaje.

Como `Fiber[T]` del cap. 13, `Pid[Msg]` está **atado al scope que lo creó**. No puedes guardarlo en un record que sobreviva al nursery, devolverlo de una función fuera de la familia estándar, ni pasarlo entre estructuras de datos no aprobadas. El compilador lo rechaza. Esto cierra el modelo: cada PID tiene un padre conocido, y muere con él.

14.2 `with_mailbox`: dar mailbox a la fibra actual

La forma más simple de empezar es darle mailbox a la fibra en la que ya estás. `with_mailbox` instala el handler de `Actor[Msg]` y entrega el control a su body:

```
import actor

fn main() : Unit / Console {
```

```
  with_mailbox {
    Actor.send(Actor.self(), "hola")
    Actor.send(Actor.self(), "mundo")
    Stdout.print(Actor.receive())
    Stdout.print(Actor.receive())
  }
}
```

Salida:

```
$ kai run ejemplos/cap14/01_with_mailbox.kai
hola
mundo
```

`with_mailbox { ... }` es una llamada con sintaxis de trailing lambda: el bloque entre llaves es el cuerpo que se ejecuta con el mailbox instalado. Como `with_mailbox` no le pasa argumentos al body (es una lambda de cero parámetros), el bloque no lleva flecha ni binder. Adentro, `Actor` es la capacidad disponible: `Actor.self()` devuelve el `Pid` del mailbox recién creado, `Actor.send(pid, msg)` encola, `Actor.receive()` saca el siguiente.

En este ejemplo el actor se manda a sí mismo. Es la versión "hola mundo" del modelo; el ejercicio real es comunicar dos actores distintos.

14.3 `spawn_actor`: crear un actor nuevo

Para arrancar un actor que corre en su propia fibra:

```
import actor

fn trabajador() : Unit / Actor[String] + Console {
  let t1 = Actor.receive()
  Stdout.print("trabajando: " ++ t1)
  let t2 = Actor.receive()
  Stdout.print("trabajando: " ++ t2)
  let t3 = Actor.receive()
  Stdout.print("trabajando: " ++ t3)
}

fn main() : Unit / Console + Spawn + Cancel + Actor[String] {
  with_mailbox {
    let pid = spawn_actor(() => trabajador())
    Actor.send(pid, "tarea-1")
    Actor.send(pid, "tarea-2")
    Actor.send(pid, "tarea-3")
    spawn.yield()
    spawn.yield()
    spawn.yield()
    spawn.yield()
  }
}
```

`spawn_actor` arranca una fibra nueva, le instala un mailbox, y devuelve el `Pid` para que el padre pueda mandarle mensajes. La firma del trabajador declara `Actor[String]`: necesita el efecto para llamar a `Actor.receive()`.

Fíjate que `main` también tiene `Actor[String]` en su fila. ¿Por qué? Porque `Actor.send(pid, "tarea-1")` es una invocación a una operación del efecto `Actor[String]`, y como toda invocación a un efecto, requiere que el efecto esté disponible en el contexto. El `with_mailbox` de `main` provee esa capacidad. El `pid` ya identifica al mailbox destino; el handler solo dispatcha la operación.

Los `spawn.yield()` al final son para darle al scheduler la chance de correr al trabajador. Sin ellos, `main` saldría antes de que el trabajador procesara nada.

14.4 Policies de mailbox: qué pasa cuando se llena

Por defecto, `with_mailbox` y `spawn_actor` crean un mailbox **unbounded**: nunca se llena, los mensajes se acumulan mientras no se lean. Es razonable para empezar, pero peligroso: si un productor manda más rápido de lo que un consumidor procesa, la memoria crece sin tope.

Para casos reales, eliges una policy. El stdlib (módulo `actor`) las expone como dos sum types:

```
# Definidos en stdlib/actor.kai, accesibles vía `import actor`.
pub type MailboxPolicy = Unbounded | Bounded(Int, Overflow)
pub type Overflow      = DropOldest | DropNewest | BlockSender
```

`Bounded(capacity, on_full)` da un mailbox de tamaño fijo. El parámetro `on_full` decide qué hacer cuando llega un mensaje y ya no hay espacio:

- **`DropOldest`**: el mensaje más viejo se evicciona, el nuevo entra. Útil para snapshots: solo importa el estado más reciente (telemetría, tick events, GPS).
- **`DropNewest`**: el nuevo se rechaza, el mailbox queda como estaba. Útil para "el primero gana" (elección de líder, adquisición de lock).
- **`BlockSender`**: el emisor se suspende hasta que se libere espacio. Útil para **backpressure**: el productor frena cuando el consumidor no da abasto. Es punto de yield, así que un sender bloqueado puede recibir `Cancel.raise()`.

```
import actor

fn main() : Unit / Console {
  with_mailbox_policy(Bounded(2, DropOldest)) {
    Actor.send(Actor.self(), "a")
    Actor.send(Actor.self(), "b")
    Actor.send(Actor.self(), "c")
    Stdout.print(Actor.receive())
    Stdout.print(Actor.receive())
  }
}
```

Salida:

```
$ kai run ejemplos/cap14/04_mailbox_policy.kai
b
c
```

El mailbox tiene capacidad 2. Mandamos `a`, `b`, `c` sin leer nada. Cuando llega `c`, no hay espacio, y `DropOldest` saca `a`. Las dos lecturas recuperan `b` y `c`.

`DropOldest` y `DropNewest` **no notifican al emisor** que su mensaje se descartó. Si necesitas saberlo, usa `BlockSender` (o diseña el protocolo con un acuse de recibo). El silencio es deliberado: la policy expresa una preferencia global del mailbox, no una negociación por mensaje.

14.5 Patrón request/reply

El patrón más común entre actores es pedirle algo a uno y esperar respuesta. Cada lado del diálogo tiene su propio tipo de mensaje: el cliente manda un `Request` y recibe un `Reply`; el servidor recibe `Request` y manda `Reply`. El `Request` incluye el `Pid[Reply]` del cliente para que el servidor sepa dónde responder.

```
import actor

type Request = Query(String, Pid[Reply])
type Reply   = Answer(String)

fn servidor() : Unit / Actor[Request] + Actor[Reply] + Console {
  match Actor.receive() {
    Query(p, cliente) -> {
      Stdout.print("servidor: recibí '#{p}'")
      Actor.send(cliente, Answer("respuesta a '#{p}'"))
      servidor()
    }
  }
}

fn main() : Unit / Console + Spawn + Cancel + Actor[Reply] {
  with_mailbox {
    let server = spawn_actor(() => servidor())

    Actor.send(server, Query("dos+dos", Actor.self()))
    match Actor.receive() {
      Answer(r) -> Stdout.print("cliente: " ++ r)
    }
  }
}
```

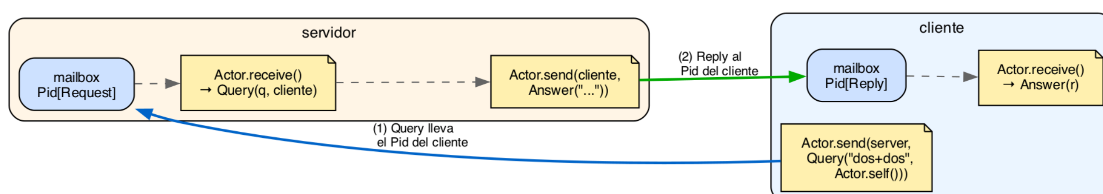

Figura 14.1 · *Request/reply entre dos actores. El cliente tiene un mailbox* `Pid[Reply]`*; el servidor tiene un mailbox* `Pid[Request]`*. La flecha (1) lleva la* `Query` *— que incluye el PID del cliente — hasta el mailbox del servidor; la flecha (2) devuelve el* `Answer` *al mailbox del cliente. Dos mailboxes tipados, dos mensajes, una ida y vuelta.*

Lo importante de la estructura:

- **Dos tipos distintos, `Request` y `Reply`**, cada uno con su propio mailbox. El servidor declara `Actor[Request] + Actor[Reply]` en su fila: recibe `Request` desde su propio mailbox y envía `Reply` al mailbox del cliente. El cliente declara solo `Actor[Reply]`: él tiene mailbox de `Reply`, no de `Request`. Los tipos te dicen exactamente qué mailbox es qué.
- **`Query` incluye el `Pid[Reply]` de retorno.** Sin eso, el servidor no sabe a quién contestarle. El tipo del `Pid` garantiza que solo se le pueden enviar mensajes de tipo `Reply`.
- **El servidor recursa** después de procesar el mensaje. Sin esa recursión, el servidor procesaría un solo mensaje y terminaría. La recursión por cola se compila a un loop (cap. 6), así que el actor puede correr indefinidamente sin reventar el stack.

Esto reemplaza, en términos prácticos, las llamadas sincrónicas a una API: pides algo, esperas respuesta, sigues. La diferencia es que aquí el "servidor" puede estar atendiendo a varios clientes a la vez, su estado interno está encapsulado, y el modelo de tipos te garantiza que ningún cliente se queda esperando una respuesta del tipo equivocado.

14.6 Supervisión: links y monitores

En BEAM, los actores se supervisan con **links** (bidireccional) y **monitores** (unidireccional). Cuando un actor cae, los actores que lo observan se enteran y deciden qué hacer.

kaikai trae el mismo modelo, expresado como dos efectos del módulo `actor` del stdlib:

```
# Declarados en stdlib/actor.kai, junto con Actor[Msg].
pub effect Link {
  link(pid: Pid[_]) : Unit
}

pub effect Monitor {
  monitor(pid: Pid[_]) : MonitorRef
  demonitor(ref: MonitorRef) : Unit
}
```

`Pid[_]` es un PID "existencial": cualquier tipo de mensaje sirve. Eso porque `link` y `monitor` no envían ni reciben mensajes; solo registran observación sobre la vida del actor.

Links: bidireccionales

`Link.link(pid)` declara que el actor actual y `pid` están ligados: si cualquiera de los dos termina con un fallo, el otro recibe `Cancel.raise()`. Es el patrón para dos actores que dependen simétricamente uno del otro (un worker y su cola, los dos lados de un handshake). No es lo que quieres para "supervisor observa worker": ese es el caso de monitores.

Monitores: unidireccionales

`Monitor.monitor(pid)` declara que el actor actual quiere saber cuándo `pid` termina, sin acoplar la vida del observador a la del observado. Cuando `pid` termina (normal, crash o cancelación), el observador recibe un mensaje `MonitorDown` en su mailbox. El stdlib expone los tipos relevantes:

```
# Definidos en stdlib/actor.kai junto con el efecto Monitor.
pub type MonitorDown = MonitorDown(MonitorRef, TerminationCause)
pub type TerminationCause
  = Normal
  | Crashed(String)
  | Cancelled
```

Para que un actor pueda recibir `MonitorDown`, su tipo de mensaje debe incluirlo como variante:

```
type SupervisorMsg
  = Tick
  | Stop
  | Down(MonitorDown)
```

El supervisor hace `Monitor.monitor(worker)` después de crearlo, y cualquier terminación del worker llega como `Down(ev)` al `match` principal del supervisor. El supervisor decide qué hacer (reiniciar, escalar, ignorar) sin que su vida quede atada a la del worker.

Cuándo elegir cada uno

- **Sin `Link` ni `Monitor`**, cuando el flujo natural es que el actor mismo reporte cómo le fue antes de terminar. Manda un mensaje `Done(...)` o `Failed(...)` a su supervisor y termina limpio. El supervisor lo ve como cualquier otro mensaje. Es el patrón del caso de estudio en §14.7.
- **`Monitor`**, cuando el supervisor necesita reaccionar a terminaciones que el actor no controla: crashes, cancelación desde afuera, panics. El supervisor sigue vivo y decide.
- **`Link`**, cuando dos actores forman una unidad y no tiene sentido que uno sobreviva sin el otro.

El patrón de §14.7 que sigue usa notificación explícita por ser lo más simple. Las versiones que mencionan `Monitor` o `Link` son refinamientos que conviene introducir solo cuando el protocolo de mensajes se vuelve insuficiente.

14.7 Caso de estudio: supervisor con reintentos

Cerramos con un programa completo: un supervisor que lanza un trabajador con un lote de tareas, observa si el lote fue exitoso, y reintenta con un lote alternativo si falló.

```
import actor

#[derive(Show)]
type ResultadoLote
```

```
  = Done(Int)           # suma total exitosa
  | Failed(String)      # razón de la falla

fn procesar(tareas: [(Int, Int)], acc: Int) : ResultadoLote {
  match tareas {
    []                -> Done(acc)
    [(a, 0), ..._]      -> Failed("división por cero en (#{a}, 0)")
    [(a, b), ...resto]  -> procesar(resto, acc + (a / b))
  }
}

fn trabajador(supervisor: Pid[ResultadoLote], tareas: [(Int, Int)])
    : Unit / Actor[ResultadoLote] + Console {
  let r = procesar(tareas, 0)
  Stdout.print("trabajador: lote terminó como #{r}")
  Actor.send(supervisor, r)
}

fn intento(me: Pid[ResultadoLote], lote: [(Int, Int)])
    : ResultadoLote / Console + Spawn + Cancel + Actor[ResultadoLote] {
  let _ = spawn_actor(() => trabajador(me, lote))
  Actor.receive()
}

fn supervisor() : Unit / Console + Spawn + Cancel + Actor[ResultadoLote] {
  with_mailbox {
    let me = Actor.self()
    let primer_lote = [(10, 2), (20, 4), (30, 0)]
    let segundo_lote = [(10, 2), (20, 4), (30, 5)]
    match intento(me, primer_lote) {
      Done(total)    -> Stdout.print("supervisor: éxito al primer intento, total=#{total}")
      Failed(motivo) -> {
        Stdout.print("supervisor: primer intento falló (#{motivo}), reintento")
        match intento(me, segundo_lote) {
          Done(total)    -> Stdout.print("supervisor: éxito al segundo intento, total=#{total}")
          Failed(motivo) -> Stdout.print("supervisor: segundo intento falló también (#{motivo}), me rindo")
        }
      }
    }
  }
}

fn main() : Unit / Console + Spawn + Cancel {
  supervisor()
}
```

Salida:

```
$ kai run ejemplos/cap14/05_supervisor.kai
trabajador: lote terminó como Failed(división por cero en (30, 0))
supervisor: primer intento falló (división por cero en (30, 0)), reintento
trabajador: lote terminó como Done(16)
supervisor: éxito al segundo intento, total=16
```

Tres piezas vale comentar:

- **`procesar` no toca el sistema de actores.** Es lógica pura sobre listas: pattern match, recursión, devuelve un valor. Eso significa que `procesar` es completamente testeable sin arrancar fibras. La capa de actores queda solo en `trabajador`, `intento` y `supervisor`.
- **`intento` separa una preocupación.** Antes de extraerlo, el código del segundo intento estaba inline dentro del primer `match`. Llevar esa lógica a una función separada hace explícito el patrón "spawn + receive".
- **El supervisor decide la política.** El trabajador es ciego a la decisión: solo reporta. Cambiar la política de "dos intentos con datos distintos" a "tres intentos con backoff exponencial" toca una sola función. Esa separación es la ventaja real del modelo.

¿Qué le falta a este programa para ser de producción? Algunas cosas:

- **Tiempo máximo por intento.** Hoy si el trabajador se cuelga, el supervisor se cuelga con él. La solución es un `with_timeout` (ejercicio del cap. 13) alrededor del `intento`.
- **Cancelación del trabajador si el supervisor decide rendirse.** Si decimos "me rindo" mientras el trabajador sigue procesando, queremos que el trabajador termine también. Con `Link.link(worker)` después del spawn, la decisión del supervisor de retornar cancela el worker automáticamente. La alternativa explícita es agregar un `Stop` al protocolo del trabajador y mandárselo antes de rendirse.
- **Logging persistente.** En vez de `Stdout.print`, mandar a un actor logger con su propio mailbox bounded (`Bounded(1024, DropOldest)`).

Cada uno es un capítulo aparte. Pero la base está: tres actores, dos tipos de mensaje, un sistema de tipos que garantiza que las mailboxes se respetan.

14.8 Filosofía: actores son una biblioteca

Si quieres recordar dos cosas del capítulo, que sean estas:

1. **Los actores no son primitivos del lenguaje.** Son una biblioteca construida sobre el efecto `Actor[Msg]`, que a su vez es una declaración ordinaria de efecto. El stdlib provee `with_mailbox`, `spawn_actor`, las policies. El código de esa biblioteca está disponible para leer. Si alguna vez te preguntas "qué hace `spawn_actor` por dentro", la respuesta es un `handle` con un `spawn.spawn`.

2. **Cada actor tiene un tipo de mensaje fijo.** El compilador te garantiza que no mandas el mensaje equivocado al mailbox equivocado. Los `Pid[Msg]` son tipados, no strings. Y por eso un sistema de actores en kaikai es más verificable estáticamente que su equivalente en Erlang.

La consecuencia más importante de la primera idea: cuando quieras un patrón de supervisión que el stdlib no provee, puedes escribirlo. La sintaxis no oculta nada: `handle`, `receive`, `send`. Es difícil que un patrón de actores que hayas visto en Erlang, Akka, o cualquier framework de actores no sea expresable como una función ordinaria en kaikai con estas piezas.

Ejercicios

14.1. Modifica el ejemplo §14.3 para que el trabajador procese cinco tareas en vez de tres. ¿Qué pasa si reduces los `spawn.yield()` del padre? ¿Cuál es el mínimo que hace que todas las tareas se procesen?

14.2. Toma el `BoundedDropOldest` de §14.4 y cámbialo a `BoundedDropNewest`. ¿Cuál es la salida esperada? Justifica con un razonamiento sobre qué mensajes quedan en el mailbox cuando llega cada `send`.

14.3. En el patrón request/reply de §14.5, el cliente manda una sola pregunta y se va. Si quisieras un cliente que hace cinco preguntas en serie, ¿qué cambiarías? ¿Y si las quisieras lanzar todas a la vez y recibir las respuestas conforme lleguen (concurrentemente, no en paralelo: el scheduler las intercala en el mismo hilo)? Pista: necesitas abrir varias fibras dentro de un nursery o agrupar respuestas con un correlation id.

14.4. En el caso de estudio §14.7, el supervisor reintenta dos veces. Generaliza: escribe una función `con_reintentos[T, e](n: Int, intento: () -> ResultadoLote / e) : ResultadoLote / e` que reintente `n` veces antes de rendirse.

14.5. Un actor "logger" recibe `Info(String)` y los imprime a stdout. Diseña su tipo de mensaje, su firma, y un `with_mailbox_policy` apropiado. Justifica la elección de policy considerando: ¿qué pasa si el productor manda más rápido de lo que se puede imprimir?

Capítulo 15 · Holes y kaikai con agentes IA

Este capítulo trata sobre una herramienta pequeña con una idea grande detrás. La herramienta se llama **hole**: un agujero que dejas en el código en lugar de una expresión, con un `?` o un `?nombre`. La idea grande es que el compilador puede **hablarte** mientras escribes: decirte qué tipo se espera ahí, qué nombres están en alcance, qué expresiones podrían encajar. El programa sigue compilando con holes adentro; solo aborta si la ejecución llega a uno.

Los holes son útiles aunque nunca uses un LLM. Te permiten diseñar de arriba hacia abajo (escribir la firma primero, completar el cuerpo después) y avanzar en partes sin que el archivo entero deje de compilar. Eso sirve al lector humano.

Pero los holes son también la pieza con la que kaikai se diseña para **agentes IA**. El compilador puede emitir su reporte como JSON, y un LLM lee ese JSON para entender qué se le está pidiendo. Esta es la apuesta estratégica del lenguaje (`design.md` la llama Tier 3): un lenguaje nuevo, sin corpus de entrenamiento grande, puede ser autoreable por un agente si las herramientas están bien diseñadas.

Vamos por partes. Primero los humanos.

15.1 Holes tipados: `?` y `?nombre`

Un hole es una expresión legal en kaikai. Se escribe con `?` solo, o con `?nombre` si quieres darle identidad:

```
fn area_circulo(r: Real) : Real = ?formula
```

Esto compila. La función `area_circulo` existe, tiene la firma correcta, y se puede llamar desde otras partes del programa. Lo que pasa cuando la ejecución llega al `?formula` es que el programa aborta con un mensaje claro:

```
$ kai run ejemplos/cap15/01_hole_basico.kai
panic: unfilled hole: ?formula
```

No es un error de compilación. Es una **promesa diferida**: te dejaste un agujero que vas a llenar después, y el sistema te acompaña hasta entonces.

La diferencia entre `?` y `?nombre` es que el nombre te sirve para identificar el hole en mensajes y, sobre todo, para hacer que **dos holes con el mismo nombre dentro de la misma función compartan tipo**:

```
fn clasificar(n: Int) : String {
  if n < 0 {
    ?palabra
  } else {
    ?palabra
  }
}
```

Los dos `?palabra` se unifican: si decides que uno es `String`, el otro también lo es. Eso reduce la tentación de escribir implementaciones inconsistentes entre brazos de un `if` o de un `match`.

Anónimos (`?` sin nombre) cada uno es independiente.

15.2 La conversación con el compilador

La gracia de los holes no está en abortar bonito, sino en lo que el compilador te dice de ellos. Para cada hole, emite un **reporte**:

```
$ kai build ejemplos/cap15/01_hole_basico.kai --holes
ejemplos/cap15/01_hole_basico.kai:1:32: type hole

  expected: Real

  in scope:
    r : Real

  candidates that fit:
    r
    real_mul(r, r)

  replace `?formula` with one of the candidates or a literal Real.
```

Cuatro piezas de información:

- **`expected`**: el tipo que la posición del hole exige. El compilador lo deduce del contexto: aquí, la función devuelve `Real`, el cuerpo es una expresión sola, entonces el hole tiene que ser `Real`.
- **`in scope`**: cada nombre alcanzable desde el punto del hole, con su tipo. Aquí solo `r : Real` (el parámetro).
- **`candidates that fit`**: expresiones que el compilador puede sintetizar y que tienen el tipo esperado. Para `Real` con `r` en alcance: `r` mismo, `real_mul(r, r)` que es `Real` también. La síntesis es **bounded**: a lo más una aplicación de función. No te da el cuerpo completo, te da pistas.
- **`replace`**: la sugerencia final, en una línea.

Esta es la conversación. Mientras la firma es lo único que sabes, el compilador te ayuda a ver qué se puede poner adentro.

Como cualquier reporte del compilador, el costo de invocarlo es bajo: corres `kai build --holes` y lees. No tienes que adivinar.

15.3 Diseño top-down: empieza por la firma

El uso más natural de los holes es el **diseño de arriba hacia abajo**. Empiezas escribiendo la firma de lo que quieres, sin saber cómo se va a implementar. Pones un `?` en el cuerpo. El programa compila. Pasas a la siguiente función.

```
fn tokenizar(s: String) : [Token] = ?tokens

fn parsear(ts: [Token]) : Expr = ?ast

fn evaluar(e: Expr) : Int {
  match e {
    Lit(n)        -> n
    Suma(a, b)    -> evaluar(a) + evaluar(b)
    Mul(a, b)     -> evaluar(a) * evaluar(b)
  }
}

fn calcular(s: String) : Int = evaluar(parsear(tokenizar(s)))
```

Tres firmas, una sola función completa (`evaluar`). El archivo compila. Puedes correr tests sobre `evaluar` con ASTs hechos a mano, antes de implementar `parsear` o `tokenizar`:

```
fn main() : Unit / Console {
  let ast = Suma(Lit(3), Mul(Lit(4), Lit(5)))
  println("3 + 4*5 = #{evaluar(ast)}")
}
```

Salida:

```
$ kai run ejemplos/cap15/05_diseno_top_down.kai
3 + 4*5 = 23
```

El programa corre. La evaluación funciona. Ahora vas a llenar los dos holes uno a uno: implementas `parsear` (toma `[Token]`, da `Expr`), después `tokenizar` (toma `String`, da `[Token]`). Y cuando llegues a llenar el último, el `main` puede llamar a `calcular("3 + 4*5")` directamente.

Esto contrasta con dos formas de trabajo más comunes:

- **Bottom-up:** implementas las piezas más pequeñas primero, las combinas. Funciona, pero a veces descubres que las piezas no encajan al final.
- **Big bang:** escribes todo de una vez, no compila hasta el final. Funciona si tienes el problema clarísimo. Si no, es doloroso.

Top-down con holes es un punto intermedio: la estructura va existiendo desde el principio, la corrección de cada pieza se verifica conforme la rellenas.

15.4 Programas parciales: avanzar con el resto compilando

Una consecuencia del diseño con holes es que **siempre puedes correr lo que ya tienes**. Si una función está completa, puedes testearla sin esperar a que el archivo entero esté listo:

```
fn duplicar(x: Int) : Int = x * 2

fn promedio(a: Int, b: Int) : Int = ?formula

fn main() : Unit / Console {
  println("duplicar(5) = #{duplicar(5)}")
  # promedio no está, pero el archivo compila.
}
```

`duplicar` funciona y `kai run` la imprime. `promedio` espera ser implementada; mientras no llamemos a `promedio`, el programa no se topa con el hole y corre limpio.

Esto reduce mucho la fricción de mantener un programa "casi compilando" mientras lo desarrollas. Otros lenguajes te empujan a escribir stubs con `unimplemented()`, `todo()` o `return null`; en kaikai el `?` es la primitiva del idioma, y el compilador entiende que tiene tipo.

15.5 Holes en patrones: el match incompleto

Un hole en posición de patrón funciona también:

```
type Forma
  = Circulo(Real)
  | Cuadrado(Real)
  | Triangulo(Real, Real)

fn area(f: Forma) : Real {
  match f {
    Circulo(r)      -> 3.14 * r * r
    Cuadrado(l)     -> l * l
    Triangulo(b, h)  -> ?formula_triangulo
  }
}
```

Aquí el `match` ya cubre los tres constructores, lo único que falta es la expresión del último brazo. El compilador verifica exhaustividad (cap. 5 §5.4), te dice que el `match` está completo, y reporta el hole con `Real` como tipo esperado y `b`, `h` en alcance.

Si todavía no decidiste si quieres `Triangulo` en el tipo, lo borras y el compilador te avisa que ahora `match` no es exhaustivo. Los holes son ortogonales a la verificación de patrones; cada uno hace su trabajo.

15.6 La apuesta LLM: lenguaje diseñado para agentes

Hasta aquí los holes son una herramienta para humanos. La parte más estratégica del cap. 15 es que **los holes son también la puerta de entrada para que un agente IA escriba kaikai**.

El razonamiento es simple. Un LLM aprende un lenguaje del **corpus** disponible en su entrenamiento. Lenguajes con mucho corpus (Python, JavaScript) son fáciles de generar para los modelos; lenguajes nuevos, con poco código público, son difíciles. Si kaikai esperara a tener un corpus grande, perdería el experimento.

Pero hay una alternativa: diseñar el lenguaje para que el compilador sea quien le enseñe al agente, no el corpus. Si cuando el LLM produce código incorrecto, el compilador puede decir con precisión qué falta y dónde, el LLM puede iterar hasta llegar al programa correcto en pocos pasos.

La pieza clave es la **salida estructurada**: el compilador emite JSON que el agente lee directamente, sin parseo heurístico. Tres canales son relevantes:

1. `kai build --holes-json`: el reporte de holes en formato JSON.
2. `kai type --json`: el tipo de cualquier expresión.
3. **Diagnósticos del compilador en JSON**: errores de tipo, matches no exhaustivos, efectos no manejados.

El cap. 5 §5.4 mostró cómo se ve un mensaje de match no exhaustivo en formato humano. La versión JSON contiene los mismos campos como datos: tipo del escrutinio, lista de variantes faltantes, lista de variantes cubiertas, sugerencia. Un agente puede parsearlo y producir el código que cubre la variante faltante sin necesidad de leer texto natural.

15.7 La salida JSON de los holes

El reporte JSON tiene un esquema estable:

```
[
  {
    "file": "area.kai",
    "line": 1, "col": 32,
    "name": "formula",
    "expected_type": "Real",
    "in_scope": [
      {"name": "r", "type": "Real"}
    ],
    "candidates": [
      {"expr": "r", "kind": "local"},
      {"expr": "real_mul(r, r)", "kind": "application"}
    ]
  }
]
```

Cada hole es un objeto. El array contiene tantos elementos como holes haya en el archivo. Los campos son los mismos del reporte humano del §15.2, pero como datos estructurados.

Para un humano, esto es ruidoso. Para un agente, es exacto. Y exacto importa: la diferencia entre "el agente acertó al tercer intento" y "el agente acertó al primero" es la diferencia entre una herramienta práctica y una que se siente mágica.

15.8 Más allá de holes: información rica como interfaz

Los holes son la primera pieza del patrón general. Otras herramientas del compilador siguen el mismo principio: emitir información estructurada que un agente puede consumir.

- **`kai type <expr>`** devuelve el tipo de una expresión en el contexto de un archivo. Con `--json`, el resultado es un objeto con campos `type`, `effects` (la fila), y `notes` (referencias a la definición).
- **`kai check`** corre las propiedades y tests del archivo y reporta resultados. Con `--json`, cada test/check/bench es un objeto con `name`, `status`, `duration_ms`, y un campo `counterexample` para los `check` que fallaron (con el valor exacto que rompió la propiedad).
- **Diagnósticos del compilador** (errores y warnings) tienen forma `--json`: tipo del error, ubicación, span, mensaje humano, mensaje estructurado, lista de sugerencias.

La regla común: el agente nunca tiene que parsear texto. La información llega ya estructurada.

15.9 Un loop de trabajo con un agente

Cuando un programador trabaja con un agente IA sobre kaikai, el flujo razonable es éste:

1. **El humano escribe la firma y los tests.** Define qué se espera del programa. Pone holes en los cuerpos.
2. **El agente lee la salida `--holes-json`.** Sabe qué tipo se espera, qué bindings están en alcance, qué candidatos son razonables. Genera una propuesta de implementación.
3. **El humano corre `kai check`.** Si los tests pasan, sigue adelante. Si no, el contraejemplo le dice al agente qué está mal.
4. **El agente itera.** Lee el contraejemplo, ajusta, propone otra implementación.

Tres cosas vale fijar de este flujo:

- **El humano decide el qué.** La firma, los tests, las propiedades son las que dicen qué debe hacer el programa.
- **El agente decide el cómo.** El cuerpo de las funciones, las estructuras intermedias, los detalles de implementación.
- **El compilador media.** Es el árbitro: dice si la propuesta cumple con los tipos, con los tests, con las propiedades. El agente nunca acepta nada sin que el compilador haya dicho que pasa.

Es lo opuesto al patrón "el LLM escribe el código y el humano revisa". Acá el humano escribe **la especificación** (firma + tests), el agente escribe **el código**, el compilador **verifica** que el código satisface la especificación.

15.10 Lo que el lenguaje no automatiza

Los holes son una herramienta de comunicación con el compilador. No son una herramienta de comunicación con el juicio del programador. Hay cosas que el compilador no puede decir, y que ningún `--holes-json` va a entregarte:

- **Qué función necesita tu programa.** Si decides que `area_circulo` debe existir, eso es decisión tuya. El compilador no va a inventar la firma por ti.
- **Cómo se llaman las cosas.** Si nombras tu función `process_data` en vez de `transform_records`, ningún hole te va a corregir. El gusto y la legibilidad son tuyos.
- **Si la arquitectura tiene sentido.** Que el compilador acepte una pieza no significa que la pieza esté en el lugar correcto. Decidir qué pertenece a cada módulo, qué efectos expone cada función, cuándo extraer una abstracción: eso es diseño, y el diseño es humano.

Hay una idea recurrente en el blog del autor sobre esto: las herramientas no reemplazan el juicio, lo apalancan cuando están bien diseñadas. El compilador con holes y agentes lee una parte mecánica del trabajo (los detalles que satisfacen las restricciones de tipo). La otra parte (qué construir, qué abstraer, qué priorizar) sigue siendo del programador.

15.11 Caso de estudio: completar una función no trivial

Cerramos con un ejercicio realista. Imagina que quieres escribir una función que toma una lista de pares `[(String, Int)]` representando notas de estudiantes, y devuelve los nombres de quienes aprobaron (nota >= 4) en orden alfabético.

El humano escribe la firma y dos tests:

```
fn aprobados(notas: [(String, Int)]) : [String] = ?cuerpo

test "lista vacía" {
  assert aprobados([]) == []
}

test "filtra y ordena" {
  let r = aprobados([("Carmen", 5), ("Ana", 3), ("Berta", 6)])
  assert r == ["Berta", "Carmen"]
}
```

El humano corre `kai build --holes-json`. El agente recibe:

```
{
  "name": "cuerpo",
  "expected_type": "[String]",
  "in_scope": [
    {"name": "notas", "type": "[(String, Int)]"}
  ],
  "candidates": [
    {"expr": "[]", "kind": "literal"}
  ]
}
```

El agente sabe: tipo esperado `[String]`, una entrada `notas` de tipo `[(String, Int)]`. Los candidatos son magros porque la síntesis del compilador es bounded; el agente tiene que proponer algo más sustantivo. Una primera propuesta:

```
fn aprobados(notas: [(String, Int)]) : [String] =
  notas
   |? . .1 >= 4
   | . .0
   |> list.sort
```

El agente corre `kai check`. El primer test pasa. El segundo falla: el orden es `["Berta", "Carmen"]` y devuelve eso, pero también necesita probar que filtra. Veámoslo a la inversa: el test asume orden lexicográfico ascendente; `list.sort` sobre `[String]` debería hacer eso. Si pasa, el agente termina.

El humano nunca tocó el cuerpo. El compilador validó tipos y tests. El agente iteró si hizo falta. Cada uno hizo lo que mejor sabe hacer.

¿Es siempre así de limpio? No. Hay funciones donde el agente falla tres veces antes de acertar. Hay funciones donde el contraejemplo del test es ambiguo y el agente no sabe qué ajustar. Pero el costo de cada iteración es bajo (segundos), y el costo de equivocarse es transparente (el compilador o el test reporta exactamente qué está mal).

15.12 Filosofía: tres ideas que vale recordar

1. **Holes son una primitiva de diálogo.** No son `null` ni `unimplemented()`: son una forma legal de expresión que compila, que el compilador entiende, y para la que emite información estructurada. El humano la usa para diseñar top-down; el agente la usa como punto de entrada.
2. **El compilador es la interfaz del lenguaje.** Lo que el compilador dice (tipos esperados, errores, contraejemplos, candidatos) es lo que define qué se puede hacer en kaikai. Diseñar bien esa salida (humana y JSON) es lo que vuelve al lenguaje accesible a humanos y a agentes por igual.
3. **El humano dice el qué; el agente dice el cómo; el compilador verifica.** Es el reparto que kaikai propone para el trabajo asistido por IA. Cada parte hace lo que mejor sabe hacer. Ninguna reemplaza a las otras.

Ejercicios

15.1. Toma una función simple que conozcas (digamos `fn sumar_pares(xs: [Int]) : Int`). Escribe la firma con un `?cuerpo` y los tests que esperarías. Sin mirar la implementación, escribe tres propuestas de cuerpo a mano y córrelas. ¿Cuántos intentos te toma encontrar la versión correcta?

15.2. Escribe una función con dos brazos de un `if`, donde cada brazo es un `?nombre` con el mismo nombre. Verifica que el compilador unifica el tipo de los dos. Después cambia uno de los nombres y observa: ahora cada brazo tiene tipo independiente. ¿En qué casos te conviene cada forma?

15.3. Lee `kai build --holes-json` sobre un archivo con tres holes en posiciones distintas. ¿Qué información comparten todos los holes? ¿Qué información es específica de cada uno?

15.4. (Requiere acceso a un agente IA.) Toma una función del cap. 5 (digamos el evaluador de expresiones del §5.7) y borra el cuerpo de una de las ramas del `match`, reemplazándolo por un `?nombre`. Pídele al agente que la complete usando solo la salida JSON del compilador como entrada. ¿Qué tan rápido lo resuelve?

15.5. Discute con un colega: ¿qué partes del trabajo que haces hoy programando son las que un agente podría hacer si le das suficiente información estructurada del compilador? ¿Qué partes seguro que no? ¿Por qué?

Capítulo 16 · Tooling: el binario `kai`

Hasta aquí cada capítulo se concentró en el lenguaje: la sintaxis, los tipos, los efectos, el modelo de memoria. Pero un lenguaje sin tooling no se usa. Este capítulo cubre el otro lado: el binario `kai`, que es la cara con la que todo programador interactúa todos los días.

Es un capítulo corto y de referencia. No hay ejercicios. La idea es que sepas qué comando usar cuándo, y que tengas a mano la lista para volver a ella.

16.1 Compilar y correr: `kai run`, `kai build`

El comando con el que vas a vivir es `kai run`:

```
$ kai run hola.kai
hola, kaikai
```

`kai run` compila el archivo a un binario nativo, lo ejecuta, y reenvía cualquier argumento adicional al programa. Es el ciclo edit-save-run del día a día. Bajo la capa hay un compilador (`kaic2`) que produce C, después invoca `cc` para compilar a un ejecutable, y al final corre el ejecutable.

Si quieres el binario sin correrlo, usa `kai build`:

```
$ kai build hola.kai
$ ./hola
hola, kaikai

$ kai build hola.kai -o build/hola
$ ./build/hola
hola, kaikai
```

`kai build` no corre el programa: solo deja el ejecutable en disco. Con `-o` especificas dónde. El binario es **estático en lo esencial**: no depende del compilador kaikai, solo de la libc del sistema. Lo puedes copiar a otra máquina con el mismo sistema operativo y arquitectura y va a correr.

Compilación rápida

`kai run` y `kai build` están diseñados para sentirse inmediatos. Un programa de unos cientos de líneas compila en menos de un segundo en una máquina razonable. Esa velocidad no es

accidental: el compilador es self-hosted (kaikai compilado en kaikai), evita pases costosos como inferencia global de tipos sin necesidad, y emite C directo en vez de pasar por LLVM. Para programas grandes hay un cache (cap. 8 §8.8 cubre el cache de paquetes; el cache de compilación del propio archivo `.kai` es otra historia).

Si quieres un sentido del tiempo: un programa de Rust de tamaño comparable puede tardar 30 segundos en compilar. Un programa de kaikai del mismo tamaño tarda menos de un segundo. La diferencia se nota.

16.2 Tests, propiedades y benchmarks

Tres subcomandos cubren las tres construcciones de verificación del cap. 7:

- **`kai test`** corre los bloques `test "..." { ... }`.
- **`kai check`** corre los bloques `check "..." with x: T { ... }` (propiedades verificadas con valores generados al azar).
- **`kai bench`** corre los bloques `bench "..." { ... }` y reporta tiempos.

```
$ kai test calculadora.kai
  ok   suma de cero
  ok   producto unitario
  ok   evaluación de literal

3/3 tests passed
```

```
$ kai check calculadora.kai
  conmutatividad de la suma: 100 iter, OK
  asociatividad de la multiplicación: 100 iter, OK

2/2 checks passed
```

```
$ kai bench calculadora.kai
  evaluación de árbol pequeño: 1000 iter / median 12 ns / MAD 1 ns / mean 13 ns / range [10, 45]
  evaluación de árbol grande: 1000 iter / median 8.4 us / MAD 0.2 us / mean 8.5 us / range [8.0, 12.1]

2 benches
```

`kai bench` admite `--iters N` para fijar el número de iteraciones (por defecto, 1000). Para benchmarks costosos conviene bajar; para mediciones más estables, subir.

Los tres comandos comparten dos propiedades importantes:

- **Solo corren los bloques relevantes.** `kai test` ignora `check` y `bench`; `kai check` ignora `test` y `bench`.
- **Los bloques no llegan al binario de producción.** `kai run` y `kai build` los descartan completamente.

16.3 Formateo: `kai fmt`

`kai fmt` es el formateador canónico. Estilo `gofmt`:

- Una sola forma correcta de imprimir cualquier archivo.
- Sin opciones de configuración. El proyecto no quiere guerras de estilo.
- Idempotente: formatear un archivo ya formateado no lo cambia.

Tres formas de uso:

```
$ kai fmt archivo.kai              # reescribe el archivo in-place
$ kai fmt --check archivo.kai      # exit 0 si está formateado, 1 si no
$ cat archivo.kai | kai fmt --stdin  # lee stdin, escribe en stdout
```

La forma `--check` está pensada para CI: si el código no está formateado, el job falla y te obliga a correr `kai fmt` antes de mergear.

La forma `--stdin` está pensada para editores: tu editor pasa el buffer al formateador antes de guardar, recibe el resultado canónico, lo escribe.

16.4 Gestión de paquetes: `init`, `add`, `install`, `update`

El cap. 8 §8.5-8.8 cubrió el modelo de paquetes (manifest `kai.toml`, lockfile `kai.lock`, cache compartido, minimum-version selection). Acá listamos los subcomandos que orquestan ese modelo:

```
$ kai init miapp
kai-pkg: wrote kai.toml for package 'miapp'

$ kai add github.com/kaikailang-org/manutara@v0.1.0
$ kai install
$ kai update              # refresca todas las deps
$ kai update manutara     # refresca solo manutara
$ kai show                # imprime kai.toml parseado
```

`kai run` y `kai build` invocan `kai install` automáticamente si detectan que hay dependencias declaradas en `kai.toml` pero no resueltas en `kai.lock`. En la práctica, después de clonar un proyecto kaikai, basta con `kai run` para que descargue lo que falte.

16.5 Modo de desarrollo: `kai watch`

`kai watch` es útil cuando estás iterando sobre un programa de forma intensa:

```
$ kai watch main.kai
[watching main.kai...]
```

Cada vez que guardas el archivo, el watcher detecta el cambio, recompila, y corre. Te permite tener el resultado a la vista sin tener que volver al terminal y teclear `kai run`. Es la forma más rápida de explorar un cambio en un demo o un script.

16.6 Integración con editores: `kai lsp`

`kai lsp` es el Language Server que kaikai expone para editores. Implementa el Language Server Protocol estándar, así que cualquier editor con soporte LSP (VS Code, Neovim, Emacs, IntelliJ con plugin) puede conectarse y obtener:

- Type-on-hover: pasas el cursor por una expresión y ves su tipo.
- Goto-definition: saltas al lugar donde un nombre fue declarado.
- Diagnósticos en vivo: los errores y warnings del compilador aparecen en el buffer mientras escribes.

La configuración exacta del editor varía. Para VS Code, hay una extensión oficial que arranca `kai lsp` automáticamente. Para Neovim, configurar `nvim-lspconfig` con `kai lsp` como comando.

El LSP es la pieza que vuelve el desarrollo en kaikai comparable, en ergonomía cotidiana, al de Rust o TypeScript: la retroalimentación es instantánea, sin necesidad de ir al terminal para descubrir un error.

16.7 Variables de entorno

Unas cuantas variables de entorno controlan el comportamiento del binario `kai` para casos especiales:

- **`CC`** (valor por defecto: `cc`): el compilador de C que `kai` invoca para producir el ejecutable final. Si tienes varias versiones de C en el sistema, o quieres usar `clang` específicamente, lo defines aquí: `CC=clang kai run archivo.kai`.
- **`CFLAGS`** (valor por defecto: vacío): flags adicionales para el compilador C. Útil para optimización (`CFLAGS=-O3`) o warnings (`CFLAGS=-Wall`).
- **`KAI_NO_STDLIB=1`**: salta la carga automática del stdlib. Para casos avanzados: bootstrap del compilador, embebidos sin libc completa, experimentos.
- **`KAI_STDLIB`**: override de la raíz del stdlib. Por defecto, `kai` autodetecta dónde vive (instalado vs checkout de desarrollo). Si quieres usar una versión alternativa, apuntas aquí.
- **`KAI_INCLUDE`**: override de la raíz de los headers del runtime (`runtime.h`). Mismo principio que `KAI_STDLIB`.

Para uso normal no necesitas tocar nada de esto. El binario viene preconfigurado para encontrar todo lo suyo.

16.8 Estructura típica de un proyecto

Un proyecto kaikai estándar se ve así:

```
miapp/
├── kai.toml          # manifest del paquete
├── kai.lock          # lockfile (commit junto al código)
├── main.kai          # entry point
├── lib/              # módulos públicos (si es biblioteca)
│   ├── core.kai
```

```
│   └── parser.kai
├── tests/              # tests pesados que no caben en línea
│   └── integration.kai
└── examples/           # demos que usan la biblioteca
    └── basico/
        ├── kai.toml    # con `mibib = { path = ".." }`
        └── main.kai
```

Convenciones:

- `main.kai` en la raíz si el proyecto produce un ejecutable. La firma debe ser `fn main() : ... = ...`.
- `lib/` para el código importable de un proyecto que es biblioteca. Cuando alguien instala tu paquete con `kai add`, lo que verá vía import es lo que vive bajo `lib/`.
- `tests/` para tests que prefieres tener separados (por ejemplo, porque son lentos o usan IO). Los tests en línea en el archivo del código fuente siguen siendo el patrón primario.
- `examples/<nombre>/` para demos. Cada demo tiene su propio `kai.toml` que declara una dependencia local hacia el paquete principal. Eso te permite probar la biblioteca como si fueras un usuario externo.

No es obligatorio. `kai run archivo.kai` corre cualquier archivo .kai sin importar dónde esté. Pero cuando el proyecto crece, esta estructura paga.

16.9 Hablar con C: `extern "C"` y el efecto `Ffi`

Tarde o temprano vas a necesitar una librería que ya existe en C: un driver de base de datos, un framework gráfico, un paquete numérico. La interfaz de funciones foráneas (**FFI**) de kaikai es cómo la llamas desde código kaikai sin perder el sistema de tipos ni la fila de efectos.

Declarar una función externa

El caso más simple es atarse a una función de libc directo:

```
extern "C" fn llabs(n: Int) : Int / Ffi

fn main() : Unit / Console + Ffi {
  print("|-7| = #{llabs(0 - 7)}")
}
```

```
$ kai run abs.kai
|-7| = 7
```

Línea por línea:

- `extern "C" fn name(args) : T / Ffi` declara un símbolo externo. El compilador emite una declaración forward para que el linker de C la resuelva. El cuerpo es implícito: el call site se compila a una llamada directa a función C.

- `/ Ffi` es el efecto. Cualquier función que llame a un `extern "C"` — directa o transitivamente — tiene `Ffi` en su fila. Misma disciplina que `Stdout` o `File`: una función que habla con C lo declara en su firma.

El compilador mapea los tipos primitivos de kaikai a sus equivalentes en C en el cruce:

kaikai	C
`Int`	`int64_t`
`Real`	`double`
`Bool`	`int8_t` (0 / 1)
`Char`	`int32_t` (codepoint)
`String`	`const char *` (terminado en NUL, lo posee kaikai)
`Unit`	`void` (solo retorno)

Cualquier cosa más estructurada — records, listas, tipos suma — **no** cruza directo en FFI v1. Volvemos sobre eso en un momento.

Renombrar el símbolo de C

A veces el nombre del símbolo en C choca con un identificador de kaikai o simplemente se lee mal en línea. Usa el override entre paréntesis:

```
extern "C"("strlen") fn c_strlen(s: String) : Int / Ffi
```

El nombre del lado kaikai es `c_strlen`; el linker resuelve contra `strlen`. Útil cuando el nombre C es palabra reservada en kaikai, cuando quieres un nombre con sabor kaikai sobre uno genérico de C, o cuando necesitas dos bindings kaikai que apuntan al mismo símbolo C con firmas distintas.

Enlazar contra tu propio código C

Para librerías que no estén en libc, la forma típica es: escribes un archivo C chico con las funciones que necesitas, y dejas que `kai build` invoque a su compilador C con ese archivo incluido. El gestor de paquetes no automatiza la compilación de C en v1, así que lo conectas vía la variable de entorno `CFLAGS` que `kai` pasa al compilador C anfitrión.

Un ejemplo mínimo. El lado C:

```
// shim.h
#include <stdint.h>
int64_t my_double(int64_t x);
```

```
// shim.c
#include "shim.h"
int64_t my_double(int64_t x) { return x * 2; }
```

El lado kaikai:

```
extern "C" fn my_double(x: Int) : Int / Ffi

fn main() : Unit / Console + Ffi {
  print("doble(21) = #{my_double(21)}")
}
```

Compilando:

```
$ CFLAGS="-include shim.h shim.c" kai build --backend=c app.kai -o app
$ ./app
doble(21) = 42
```

El valor de `CFLAGS` te deja inyectar cualquier cosa que el compilador C acepte: `-include` para exponer declaraciones, fuentes `.c` extra para compilar adentro, `-l<lib>` para enlazar contra librerías instaladas, `pkg-config --cflags --libs <paquete>` para usar la información de una librería del sistema. Cuando crece más allá de una línea, lo envuelves en un `Makefile`.

El flag `--backend=c` aquí es necesario porque el backend LLVM (capítulo 16 §16.1) no expone la misma plomería de `CFLAGS` en v1.

Lo que FFI v1 no hace

La lista es corta pero importante:

- **Records / structs por valor cruzando el borde.** No puedes declarar `extern "C" fn dibujar(c: Color)` donde `Color` sea un record kaikai que calza con un `struct` C. v1 pasa solo primitivos.
- **Parámetros de salida y argumentos puntero.** Nada de `int *out`: todo cruza por valor.
- **Funciones variádicas de C.** Sin binding directo a la familia de `printf`; las envuelves en un helper C de aridad fija.
- **Callbacks de C de vuelta a kaikai.** Una función C que recibe un puntero a función no puede llamar de vuelta a una función kaikai. Pospuesto para FFI v2.

El workaround canónico para el caso de structs es un **shim C**: una función C delgada que aplana el struct en primitivos en el borde kaikai y lo reconstruye antes de llamar a la librería real. El costo es una función C por cada entrada de la librería que use struct. Vale la pena para v1; FFI v2 quitará la capa de shim para el caso común.

Cuándo agarrar FFI

La regla honesta: **solo cuando realmente necesites la librería C**. Cada `extern "C"` es un hueco en las garantías del lado kaikai. El compilador no puede chequear qué hace la función C con sus argumentos, no puede probar sus efectos, no puede razonar sobre su modelo de memoria. El efecto `Ffi` al menos hace visible el hueco en la firma, pero el peso de auditoría de esa firma es "confiar en quien escribió la librería C" más "confiar en el compilador C".

Para computación pura, prefiere una implementación kaikai. Para IO y facilidades del SO, prefiere los efectos del stdlib (`Stdout`, `File`, `NetTcp`, etc.) — esos ya están conectados a C por dentro pero en una forma que los diseñadores del lenguaje controlan. FFI es la

herramienta correcta para atar ecosistemas C existentes que no quieres reescribir: drivers, toolkits nativos de UI, librerías específicas de hardware.

Una pequeña heurística: si te encuentras escribiendo más de diez `extern "C"` para envolver algo, y la librería tiene una API C estable, eso es candidato a un paquete kaikai propio cuando aterrice `kai bindgen`. Mientras tanto, el enfoque manual (un `extern "C"` por función, un shim C por cada entrada que use struct) funciona bien.

16.10 Ediciones: estabilidad sin estancamiento

Hay una decisión que el resto del libro asume sin explicarla del todo: kaikai usa **ediciones** para separar *qué prometemos que no cambia* de *qué nos reservamos el derecho de mover*. La idea no es nueva — Rust la formalizó en 2014 — pero kaikai la toma en serio desde temprano.

Qué es una edición

Una edición es un nombre — `tongariki`, `hanga-roa`, `orongo` — que fija una versión del **contrato del lenguaje** entre kaikai y tu código. Dentro de una edición, estas cosas no cambian de manera incompatible:

- la sintaxis y las palabras reservadas;
- la semántica del sistema de tipos y los efectos;
- las firmas `pub` del stdlib;
- los flags y el comportamiento del binario `kai`;
- el esquema de `kai.toml`.

Fuera del contrato — y por lo tanto libre de cambiar entre versiones — está todo lo que no toca tu código fuente: representación interna de variantes, layout de stacks de fibras, formato del caché en disco, texto exacto de los diagnósticos, fases del typer, internals del Perceus, performance.

El compromiso para ti es simple: **subir el compilador es indoloro**. Lees las notas de release, instalas la versión nueva, recompilas. El compromiso para el equipo de kaikai es también simple: podemos iterar fuerte por dentro mientras no rompamos lo de afuera. Las dos partes ganan.

Cómo se declara

En tu `kai.toml`:

```
name = "miapp"
version = "0.1.0"
edition = "hanga-roa"

[dependencies]
```

Y para verificar la edición activa de tu instalación:

```
$ kai --version
kaikai 0.76.0 - hanga-roa (stage 2, self-hosted)
```

Si el `kai.toml` omite el campo, el compilador asume la edición default de la instalación. Recomendación: en cuanto un paquete va a tener vida más allá de un fin de semana, fíjala explícitamente. Es la diferencia entre "esto compila hoy" y "esto va a seguir compilando".

Multi-edición: viejo código, compilador nuevo

El compilador kaikai acepta **cualquier edición que conozca**. Si tu paquete declara `edition = "tongariki"` y la instalación está en `hanga-roa`, el compilador aplica las reglas de tongariki para ese paquete, aunque otro paquete de la misma máquina compile contra hanga-roa. Esa es la mecánica de "estabilidad sin estancamiento": no tienes que migrar todo tu mundo al mismo tiempo.

Cuando una edición se cierra (cuando todo el ecosistema ya migró), las versiones siguientes de kaikai pueden dejar de aceptarla. Hasta entonces, viejo y nuevo conviven.

El escape hatch: `#[unstable]`

A veces un módulo quiere exponer un API nuevo *en serio* pero todavía no se compromete con la firma exacta. La anotación `#[unstable]` marca declaraciones que están fuera del contrato de la edición:

```
#[unstable]
pub fn from_stdin() : Source[String, Stdin + Spawn] / Spawn =
  ?from_stdin

#[unstable]
pub type Source[t, e] = { pid: Pid[Demand] }
```

Quien consume un API `#[unstable]` tiene que decirlo en **su propio** `kai.toml`:

```
[unstable]
ahu = true
```

La idea: nadie usa una API en evolución sin enterarse. El contrato de la edición sigue cubriendo lo demás.

Ediciones existentes

Al cierre de este libro, kaikai conoce tres:

Edición	Estado	Notas
`tongariki`	cerrada	Fase de iteración rápida pre-2026. Solo paquetes que aún no migraron.
`hanga-roa`	activa (default)	La primera edición pública. El libro está escrito contra esta.
`orongo`	futura	Próxima edición. Lo que se difiera de Hanga Roa aterriza acá.

Los nombres siguen la cantera rapanui del resto del ecosistema: lugares de Rapa Nui en orden cronológico. Cuando se cierre `hanga-roa`, llegará un anuncio, una guía de migración, y `kai migrate` para automatizar los cambios mecánicos. Mientras tanto, el código que escribiste contra hanga-roa va a seguir compilando.

16.11 Filosofía: tres principios del tooling

Si quieres recordar el tono general del tooling, son tres ideas:

1. **Velocidad primero.** Compilar y correr deben sentirse inmediatos. Si el ciclo edit--save-run es lento, el programador escribe menos código, prueba menos, y construye menos confianza. Todo el tooling de kaikai se diseña con ese reloj corriendo.
2. **Una forma correcta para cada cosa.** Un formateador canónico sin opciones. Un gestor de paquetes con MVS sin resolución compleja. Un sistema de tests integrado con el lenguaje. La filosofía es la misma que en Go: minimizar las decisiones que el programador tiene que tomar sobre cómo usar las herramientas, para liberar tiempo para decidir qué construir.
3. **Lo que vale es lo que sale del compilador.** Diagnósticos precisos, contraejemplos exactos, formatos estructurados (JSON para holes y tipos). El compilador es la interfaz real del lenguaje. Hacerlo claro y rápido es lo que vuelve a kaikai usable, antes que cualquier IDE sofisticada.

Estas no son palabras vacías. Cada vez que el lenguaje crece una característica, la pregunta de "¿cómo se va a sentir en el tooling?" se hace antes que la pregunta "¿es elegante teóricamente?". A veces gana la elegancia (los efectos algebraicos pagan algo de complejidad de tooling); a veces gana el tooling (las reglas de exhaustividad y la inferencia local se ajustan para que los mensajes sean buenos). El balance es vivo, y este capítulo es la cara visible de él.

Capítulo 17 · Caso de estudio: servidor HTTP

Llegamos al primero de dos casos de estudio que cierran el libro. La idea es ver, en un solo lugar, cómo las piezas de los capítulos anteriores encajan en algo que se parece a software real.

Este capítulo cubre **un servidor HTTP**: la familia de problemas donde lo que pesa es la concurrencia, la modularidad y la separación entre lógica de dominio e IO. El capítulo 18 cubrirá el otro extremo del espectro de la industria: **un libro mayor contable**, donde lo que pesa son los tipos precisos (monedas con unidades), las invariantes de negocio (contratos `requires` / `ensures`), y la inmutabilidad estricta. Dos casos, mismo lenguaje, distintos énfasis.

El programa es un **servidor HTTP de notas**. Tiene una interfaz HTTP simple (`GET /notas`, `POST /notas`, `GET /notas/<id>`, `DELETE /notas/<id>`), mantiene las notas en memoria, y escribe cada cambio a un archivo de log. La parte "real" no es la lógica (que es simple), sino cómo se arma: efectos en las firmas, actores para encapsular estado, fibras para servir conexiones concurrentes, módulos para separar dominio, parser HTTP, almacenamiento, persistencia.

Tamaño del programa: unas 250 líneas, repartidas en cinco archivos.

17.1 La forma del programa

Antes del código, veamos las piezas y sus responsabilidades:

```
notas/
├── kai.toml          # manifest del proyecto
├── main.kai          # punto de entrada y bucle de accept
├── dominio.kai       # tipos: Nota, Comando, Respuesta
├── almacen.kai       # actor que guarda las notas
├── persistencia.kai  # actor que escribe el log a disco
└── web.kai           # parser y serializador HTTP mínimos
```

Cinco archivos, cinco preocupaciones distintas:

- `dominio.kai` es el centro. Tipos puros, sin efectos, sin IO. Lo que el dominio "es": qué es una nota, qué comandos se pueden ejecutar, qué respuestas se pueden producir.

- `almacen.kai` es un actor. Recibe comandos, mantiene la lista de notas como estado interno, responde al que pregunta. Por dentro la lógica de manipulación es pura (función `procesar`), envuelta en un bucle `Actor.receive()` que la conecta al mundo.
- `persistencia.kai` es otro actor. Recibe eventos (creación, borrado), los escribe a un archivo de log. Aislar el disco en un actor nos permite que el almacén siga respondiendo aunque la escritura sea lenta.
- `web.kai` son funciones puras: parsear bytes HTTP en una estructura `ReqHttp`, traducir requests en comandos de dominio, serializar respuestas a bytes. Sin actores, sin IO.
- `main.kai` arma todo: arranca los actores, levanta el socket TCP, abre un nursery, y por cada conexión nueva lanza una fibra que la maneja.

Esta separación es la forma natural en kaikai. Cada módulo es ortogonal: la lógica pura del dominio se puede testear sin arrancar fibras, el parser HTTP sin abrir sockets, el almacén sin tocar el disco. El `main` solo conecta las piezas.

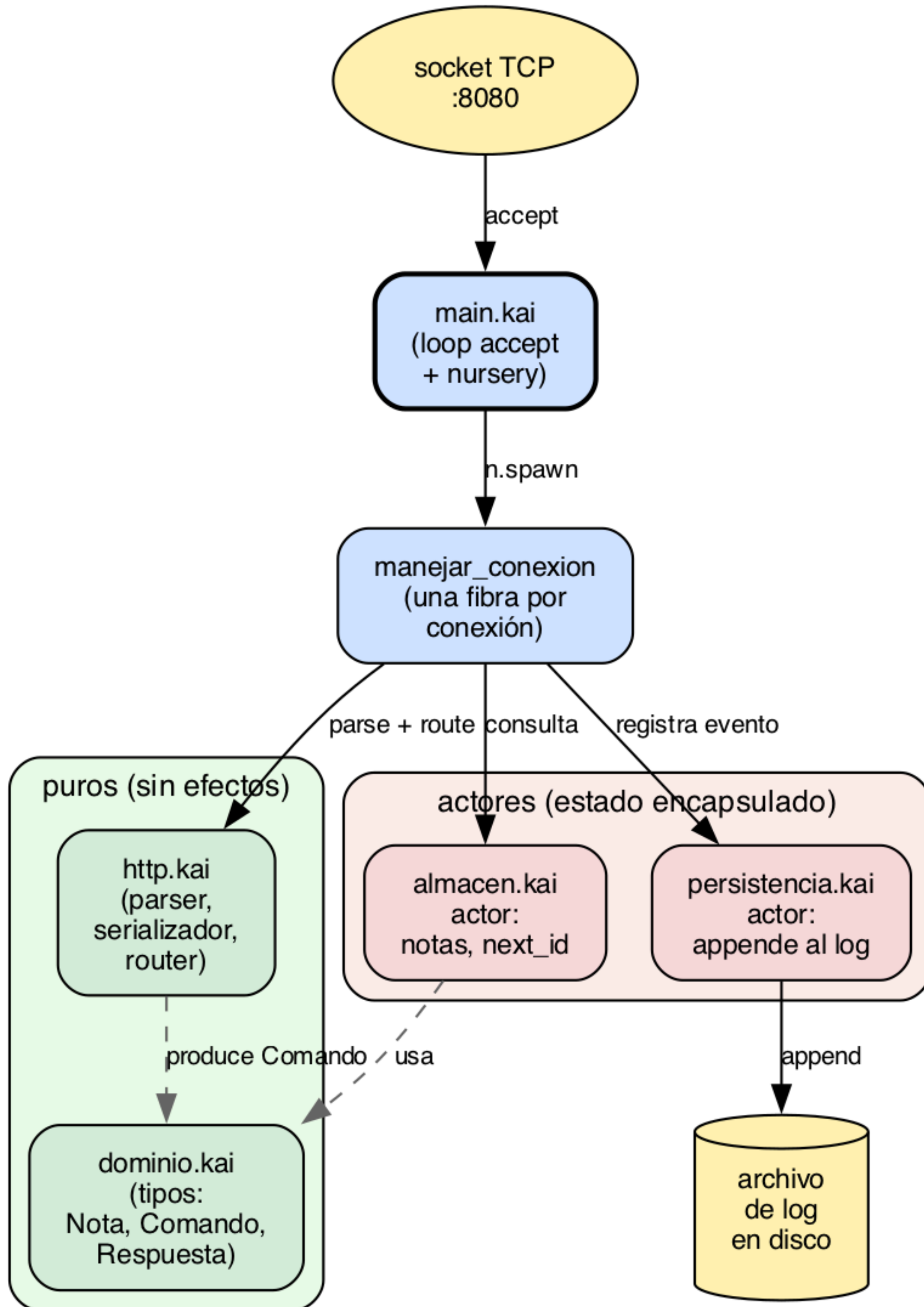

Figura 17.1 · *Arquitectura del servidor de notas. Cinco módulos, dos actores (almacén y persistencia), una fibra por conexión entrante. Los módulos puros (*`dominio.kai`*,* `web.kai`*, cluster verde) no tienen efectos; los módulos con estado (*`almacen.kai`*,* `persistencia.kai`*, cluster rojo) esconden su mutación detrás de un mailbox;* `main.kai` *es pegamento.*

17.2 El dominio: tipos puros

Empezamos por el centro. `dominio.kai`:

```
#[derive(Show)]
pub type Nota = { id: Int, cuerpo: String }

pub type Comando
  = Listar
  | Obtener(Int)
  | Crear(String)
  | Borrar(Int)

pub type Respuesta
  = Ok(String)
  | Creado(Nota)
  | NoEncontrado
  | ErrorCliente(String)
  | ErrorServidor(String)
```

Tres declaraciones. Una nota tiene id y cuerpo. Hay cuatro comandos que se pueden ejecutar contra el dominio (listar, obtener uno, crear, borrar). Hay cinco respuestas posibles, que mapean conceptualmente a códigos HTTP 200, 201, 404, 400 y 500.

Lo que **no** hay en este archivo: nada de HTTP, nada de fibras, nada de archivos. Si un día decides exponer la API por gRPC en vez de HTTP, este archivo no cambia. Si decides cambiar el almacenamiento de memoria a SQLite, este archivo no cambia. Es el invariante del programa.

El `#[derive(Show)]` sobre `Nota` es lo que nos permite interpolar `#{nota}` en un string (cap. 9). Sin él, tendríamos que escribir un `impl Show for Nota` a mano.

17.3 El almacén: actor con estado

`almacen.kai` define un actor que mantiene la lista de notas y responde comandos. Su tipo de mensaje es el comando más el `Pid` para responder:

```
import actor
import dominio

pub type AlmacenMsg = Pregunta(dominio.Comando, Pid[AlmacenResp])
pub type AlmacenResp = Respuesta(dominio.Respuesta)
```

`AlmacenMsg` es lo que el almacén recibe; `AlmacenResp` es lo que devuelve. El cliente, antes de mandar, abre su propio mailbox con `with_mailbox`, mete el `Pid` en el mensaje, y después espera la respuesta. Es el patrón request/reply del §14.5.

El corazón del módulo es la **lógica pura** de procesamiento:

```
pub fn procesar(c: dominio.Comando, notas: [dominio.Nota], next_id: Int)
    : (dominio.Respuesta, [dominio.Nota], Int) {
  match c {
```

```
    Listar -> {
      let cuerpos = list.map(notas, (n) => n.cuerpo)
      (dominio.Ok(serializar_lista(cuerpos)), notas, next_id)
    }
    Obtener(id) ->
      match buscar(notas, id) {
        Some(n) -> (dominio.Ok(n.cuerpo), notas, next_id)
        None    -> (dominio.NoEncontrado, notas, next_id)
      }
    Crear(cuerpo) -> {
      let nueva = dominio.Nota { id: next_id, cuerpo: cuerpo }
      (dominio.Creado(nueva), [nueva, ...notas], next_id + 1)
    }
    Borrar(id) ->
      match buscar(notas, id) {
        Some(_) -> {
          let restantes = list.filter(notas, (n) => n.id != id)
          (dominio.Ok("borrada"), restantes, next_id)
        }
        None -> (dominio.NoEncontrado, notas, next_id)
      }
  }
}
```

Una sola función, sin efectos en su firma. Recibe el comando, las notas actuales y el próximo id; devuelve la respuesta, la nueva lista de notas y el nuevo próximo id. Pattern match exhaustivo sobre las cuatro variantes de `Comando`. Listas construidas con `[h, ...tail]`. `list.map` y `list.filter`. Nada de esto es nuevo del cap. 17: son las construcciones del cap. 5 (sum types y match) y el cap. 6 (funciones y pipelines) puestas a trabajar.

Como `procesar` es pura, es **directamente testeable**:

```
test "crear y obtener" {
  let (r1, n1, id1) = procesar(dominio.Crear("primera"), [], 1)
  let creado_ok = match r1 {
    dominio.Creado(_) -> true
    _                 -> false
  }
  assert creado_ok
  assert id1 == 2

  let (r2, _, _) = procesar(dominio.Obtener(1), n1, id1)
  let obtener_ok = match r2 {
    dominio.Ok(c) -> c == "primera"
    _             -> false
  }
  assert obtener_ok
}
```

Cero fibras, cero IO, cero sockets. Solo lógica. Si el día de mañana queremos paralelizar la creación de notas, agregar índices, cambiar el algoritmo de búsqueda, todos los cambios pasan por esta función pura y se prueban acá.

Encima de `procesar` viene el **bucle del actor**, que la conecta a `Actor.receive()`:

```
fn bucle(notas: [dominio.Nota], proximo_id: Int)
    : Unit / Actor[AlmacenMsg] + Actor[AlmacenResp] {
  match Actor.receive() {
    Pregunta(comando, cliente) -> {
      let (resp, notas_nuevas, id_nuevo) =
        procesar(comando, notas, proximo_id)
      Actor.send(cliente, Respuesta(resp))
      bucle(notas_nuevas, id_nuevo)
    }
  }
}
```

Tres líneas de trabajo:

1. Recibe una pregunta.
2. Procesa (lógica pura).
3. Responde y recursa con el estado nuevo.

La recursión por cola se compila a un loop (cap. 6), así que el actor puede correr indefinidamente. Y la firma declara los dos efectos que el actor produce: `Actor[AlmacenMsg]` para recibir, `Actor[AlmacenResp]` para responder.

El helper `arrancar` arma todo:

```
pub fn arrancar() : Pid[AlmacenMsg] / Spawn + Cancel + Actor[AlmacenMsg] + Actor[AlmacenResp] {
  spawn_actor(() => bucle([], 1))
}
```

Y un wrapper sincrónico para clientes:

```
pub fn preguntar(almacen: Pid[AlmacenMsg], c: dominio.Comando)
    : dominio.Respuesta / Actor[AlmacenMsg] + Actor[AlmacenResp] + Cancel {
  Actor.send(almacen, Pregunta(c, Actor.self()))
  match Actor.receive() {
    Respuesta(r) -> r
  }
}
```

`preguntar` es lo que llaman los handlers HTTP del `main`: "hazle esta pregunta al almacén y dame la respuesta". Por adentro es un send seguido de un receive. Lo expone como una función simple, no como un protocolo abierto.

17.4 Persistencia: actor de escritura

`persistencia.kai` es más simple. Un actor que recibe líneas de log y las agrega a un archivo:

```
import actor
import fs.file
```

```
pub type Evento = Linea(String)

fn bucle(path: String) : Unit / Actor[Evento] + File {
  match Actor.receive() {
    Linea(s) -> {
      file.append(path, s ++ "\n")
      bucle(path)
    }
  }
}

pub fn arrancar(path: String)
    : Pid[Evento] / Spawn + Cancel + Actor[Evento] + File {
  file.write(path, "")    # trunca al inicio
  spawn_actor(() => bucle(path))
}
```

Aislar la escritura a archivo en su propio actor tiene dos beneficios:

- **El almacén no espera al disco.** Cuando el almacén procesa un `Crear`, manda un mensaje al actor de persistencia y vuelve a su trabajo. La escritura ocurre en otra fibra.
- **El orden de las escrituras está garantizado.** Todos los eventos pasan por el mismo mailbox, que se procesa en orden FIFO. No hay races aunque varios handlers escriban al log al mismo tiempo.

En un sistema real, este actor tendría un mailbox `Bounded(N, DropOldest)` para protegerse de inundación. Acá usamos el mailbox predeterminado (Unbounded) por simplicidad del demo. La decisión es explícita y vive en una sola línea, fácil de cambiar.

17.5 Parser HTTP

`web.kai` es código puro de string manipulation. La pieza central es `enrutar`, que traduce un request HTTP en un comando del dominio:

```
pub fn enrutar(req: ReqHttp) : Result[dominio.Respuesta, dominio.Comando] {
  if req.metodo == "GET" {
    if req.path == "/notas" {
      Ok(dominio.Listar)
    } else {
      enrutar_id(req.path, (id) => dominio.Obtener(id))
    }
  } else if req.metodo == "POST" {
    if req.path == "/notas" {
      Ok(dominio.Crear(req.cuerpo))
    } else {
      Err(dominio.NoEncontrado)
    }
  } else if req.metodo == "DELETE" {
    enrutar_id(req.path, (id) => dominio.Borrar(id))
  } else {
```

```
    Err(dominio.NoEncontrado)
  }
}
```

El tipo de retorno usa `Result` de forma poco ortodoxa: `Ok` contiene un comando para ejecutar; `Err` contiene una respuesta inmediata (404, 400). Esa convención mantiene la firma compacta: o el request se traduce a un comando válido, o tenemos la respuesta directamente.

Hay también un parser de la primera línea HTTP (`GET /path HTTP/1.1`) y un serializador que produce los bytes de respuesta. Son funciones puras sin efectos, testeables con strings de entrada y comparación de salida.

17.6 El main: armar todas las piezas

`main.kai` es el pegamento que arma las piezas:

```
import actor
import spawn
import fs.file
import dominio
import almacen
import persistencia
import net.tcp
import web

const PUERTO : Int = 8080
const PATH_LOG : String = "notas.log"

fn main() : Unit / Console + NetTcp + File + Spawn + Cancel + Actor[almacen.AlmacenMsg] +
Actor[almacen.AlmacenResp] + Actor[persistencia.Evento] {
  let almacen_pid = almacen.arrancar()
  let log_pid     = persistencia.arrancar(PATH_LOG)

  match NetTcp.listen("0.0.0.0", PUERTO) {
    Err(msg) -> println("error al levantar el servidor: " ++ msg)
    Ok(listener) -> {
      println("servidor escuchando en puerto #{PUERTO}")
      aceptar_loop(listener, almacen_pid, log_pid)
    }
  }
}
```

Cuatro líneas de "negocio":

1. Arrancar el almacén (actor que mantiene las notas).
2. Arrancar el persistor (actor que escribe el log).
3. Abrir un socket TCP en el puerto.
4. Entrar al bucle de aceptación.

La fila de efectos del `main` lista todo lo que el programa usa: `Console` para imprimir, `NetTcp` para sockets, `File` para escribir, `Spawn + Cancel` para fibras, `Actor[X]` para cada uno de los tres

canales de mensajes. La firma no oculta nada: si el `main` hiciera más cosas, su fila crecería en consecuencia.

El bucle de aceptación abre un nursery y por cada conexión nueva lanza una fibra. Ojo: `n` no es un valor de tipo `Nursery` que pueda viajar a otra función — el compilador reescribe cada `n.spawn(...)` en `Spawn.spawn(...)` etiquetado con el brand de *este* nursery, así que el `spawn` tiene que aparecer léxicamente dentro del bloque. Por eso el bucle accept va inline:

```
nursery { n ->
  forever(() => match NetTcp.accept(listener) {
    Err(_)   -> ()
    Ok(conn) -> {
      let _ = n.spawn(() => manejar_conexion(conn, almacen_pid, log_pid))
      ()
    }
  })
}
```

Cada conexión vive en su propia fibra. El nursery garantiza que cuando el bucle termine (porque alguien cancela el listener, o el programa recibe SIGINT), las fibras hijas también terminan. No hay handlers de conexión zombies.

Y por cada conexión, el handler:

```
fn manejar_conexion(conn, almacen_pid, log_pid) {
  let raw = leer_request(conn)
  let resp = match web.parsear_request(raw) {
    Err(msg) -> dominio.ErrorCliente(msg)
    Ok(req)  -> match web.enrutar(req) {
      Err(r)       -> r
      Ok(comando)   -> {
        registrar(log_pid, comando)
        almacen.preguntar(almacen_pid, comando)
      }
    }
  }
  NetTcp.send(conn, string_to_bytes(web.serializar_respuesta(resp)))
  NetTcp.close(conn)
}
```

Lee bytes, parsea HTTP, enruta a un comando, registra en el log, consulta al almacén, serializa la respuesta, escribe al socket, cierra. Cada paso es una función pura o un mensaje a un actor. No hay shared memory, no hay locks.

17.7 Lo que está ocurriendo, en términos del libro

Vale enumerar qué piezas del libro se usan, una a una:

- **Cap. 2** (pensar en kaikai): las funciones son expresiones; `procesar` devuelve una tupla en una sola expresión.
- **Cap. 4** (tipos compuestos): tuplas de retorno (`(Respuesta, [Nota], Int)`), records (`Nota`), listas con pattern de cabeza y cola.

- **Cap. 5** (sum types y match): `Comando`, `Respuesta`, `Evento` son sum types; los match cubren todas las variantes; el exhaustividad lo verifica el compilador.
- **Cap. 6** (funciones y pipelines): `list.map`, `list.filter` sobre la lista de notas; closures pasadas a esas funciones.
- **Cap. 7** (pruebas): tests sobre `procesar` que verifican la lógica sin arrancar fibras.
- **Cap. 8** (módulos): cinco archivos cada uno con su `pub`, imports entre ellos.
- **Cap. 9** (protocolos): `#[derive(Show)]` para interpolar notas.
- **Cap. 12** (efectos): cada función declara su fila; `handle` no aparece directamente porque los `handle`s viven dentro de `with_mailbox` y `spawn_actor` del stdlib.
- **Cap. 13** (fibras): `nursery` para estructurar el bucle de aceptación; cada conexión es una fibra.
- **Cap. 14** (actores): el almacén y la persistencia son actores; `with_mailbox` en cada cliente; `spawn_actor` para arrancarlos; mensajes tipados.
- **Cap. 16** (tooling): `kai run main.kai` lo arranca; `kai test` corre los tests del módulo `almacen`.

No hay nada nuevo aquí en términos de sintaxis. Lo nuevo es la combinación: piezas pequeñas, ortogonales, encajando en un programa con responsabilidades reales.

17.8 Cómo extenderlo

Hay varias direcciones donde el lector puede llevar este programa para profundizar lo que aprendió:

- **Persistencia con recovery.** Hoy el log es write-only. Si el servidor se reinicia, las notas se pierden. Una extensión natural: al arrancar, leer el log y reconstruir el estado.
- **Búsqueda por contenido.** El comando `Obtener` busca por id. Agregar un `Buscar(String)` que filtre por substring del cuerpo. La lógica pura va en `procesar`; el `match` del `enrutar` HTTP gana un brazo.
- **TTL por nota.** Cada nota tiene una expiración. El almacén, en cada `Obtener`, verifica si la nota expiró y la borra si sí. El campo `created_at` se agrega a `Nota`; el efecto `Time` aparece en la fila del `bucle`.
- **Múltiples instancias.** Hoy hay un solo almacén. Para un servicio más grande, particionar las notas en varios actores por hash del id. El `main` arranca N almacenes y enruta cada request al actor que corresponda.
- **Métricas.** Un cuarto actor que recibe eventos (`request_recibido`, `nota_creada`, `error_emitido`) y acumula contadores. El `main` lo arranca, los handlers le mandan eventos, un endpoint `GET /metricas` lee.
- **Test de integración.** Un programa cliente que abre una conexión TCP al servidor, manda un request, lee la respuesta, verifica que sea lo esperado. Pone el servidor en un nursery, corre el cliente, cierra.

Cada una es una sesión de tarde. Ninguna requiere cambiar la estructura básica: un dominio puro, actores con estado, fibras para concurrencia, módulos para separación.

17.9 Lo que muestra este caso

Hay un patrón claro en lo que acabamos de armar:

- **El dominio es puro.** `Comando`, `Respuesta`, `procesar`: tipos y funciones sin efectos. Se testean con entradas y salidas, sin arrancar nada.
- **Los actores envuelven el estado mutable.** El almacén mantiene la lista de notas; el persistor mantiene el archivo de log. La mutación queda encerrada dentro de cada actor, invisible para el resto del programa.
- **Las fibras paralelizan el trabajo concurrente (cooperativamente).** Una fibra por conexión. El nursery garantiza que ninguna sobreviva al servidor.
- **Los módulos separan responsabilidades.** Cinco archivos, cinco temas. Cada uno se puede reemplazar sin tocar los otros tres.

El cap. 18 va a aplicar exactamente el mismo patrón a un dominio muy distinto (contabilidad financiera) y vas a ver cómo la estructura se mantiene aunque el problema cambie. El cierre del libro viene allá.

Capítulo 18 · Caso de estudio: ledger contable

El capítulo 17 mostró un servidor HTTP: muchos clientes, una fibra por conexión, actores para encapsular estado. Es la familia de problemas donde lo que pesa es la concurrencia.

Este capítulo cierra el libro con un caso muy distinto: **un libro mayor contable**. Donde pesa la precisión, no la concurrencia. Donde un débito que no calza con su crédito es un bug que tu auditor va a encontrar antes que tú. Donde "funciona y pasa los tests" no es suficiente: hay que poder mostrar **por qué** funciona, y el sistema de tipos tiene mucho que decir al respecto.

Por qué fintech merece su propio capítulo: es uno de los dominios donde más caro sale equivocarse y donde más rinde tener garantías en el tipo. Mezclar USD con EUR cuesta plata. Sumar débitos con créditos cuesta plata. Permitir un retiro sin verificar saldo cuesta plata. Las herramientas del lenguaje (unidades de medida, contratos, branding) están hechas para que estas equivocaciones no compilen, no para que las descubras en producción.

El programa: un ledger de doble entrada que mantiene cuentas con saldo, registra transacciones (cada una con débitos y créditos), valida que las transacciones cuadren, y persiste un log de auditoría inmutable. Tamaño: unas 280 líneas en cuatro módulos.

18.1 La forma del programa

Cuatro archivos:

```
ledger/
├── kai.toml          # manifest
├── main.kai          # entry point, ejecuta operaciones de ejemplo
├── dominio.kai       # tipos: Cuenta, Movimiento, Transaccion
├── cuadre.kai        # validación de la invariante débito = crédito
├── almacen.kai       # actor que guarda cuentas y transacciones
└── persistencia.kai  # actor que escribe el log de auditoría
```

La estructura es deliberadamente similar al cap. 17. Lo que cambia es el énfasis:

- `dominio.kai` es más rico que el del cap. 17. Aparecen unidades de medida (`Real<USD>`), branded types (`Int<CuentaId>`, `Int<TransaccionId>`), y un sum type `Movimiento` que distingue débitos de créditos.
- `cuadre.kai` es nuevo. Es un módulo dedicado a una invariante: la suma de débitos en una transacción debe igualar la suma de créditos. Aparece como función pura testeable Y como **contrato** (`requires`) sobre la operación de registrar.
- `almacen.kai` es el actor de siempre, pero ahora valida cuadre antes de aceptar una transacción y mantiene saldos por cuenta. Las transacciones son **inmutables**: solo se agregan, nunca se modifican.
- `persistencia.kai` es el log de auditoría. Idea conceptual fuerte: en contabilidad, lo escrito queda. El archivo es evidencia legal.

18.2 El dominio: unidades, branding, tipos algebraicos

El centro del programa son los tipos:

```
pub unit USD

pub unit CuentaId
pub unit TransaccionId

#[derive(Show)]
pub type Cuenta = {
  id:     Int<CuentaId>,
  nombre: String,
  saldo:  Real<USD>,
}

#[derive(Show)]
pub type Movimiento
  = Debito(Int<CuentaId>, Real<USD>)
  | Credito(Int<CuentaId>, Real<USD>)

#[derive(Show)]
pub type Transaccion = {
  id:          Int<TransaccionId>,
  descripcion: String,
  movimientos: [Movimiento],
}
```

Tres decisiones del sistema de tipos vale enumerar:

- **`Real<USD>` en vez de `Real`.** El cap. 10 cubrió las unidades de medida. En este dominio importan especialmente: un programa contable que mezcla USD con EUR sin convertir produce números que parecen correctos pero no significan nada. Con UoM, esa mezcla no compila. Para extender el programa a múltiples monedas, declaras `unit EUR`, defines una tasa de cambio como `Real<USD / EUR>`, y el sistema de tipos te lleva de la mano. Sin UoM, lo descubres cuando un cliente reclama.
- **`Int<CuentaId>` vs `Int<TransaccionId>`.** El cap. 10 cubrió también los branded types. Tener `Int` puro como id significa que pasar un id de transacción donde se espera un id de

cuenta compila igual y revienta en runtime (o peor, sin avisar produce un resultado equivocado). Con branding, el sistema de tipos te lo dice antes.

- `Movimiento` **como sum type.** Un débito y un crédito tienen los mismos campos físicos (cuenta + monto), pero significan cosas distintas. Modelarlos como dos constructores del mismo sum type tiene dos efectos: el pattern match exhaustivo asegura que cualquier operación los trate explícitamente, y el sistema de tipos no nos deja sumar "monto de débito" con "monto de crédito" sin que pasemos por una conversión clara.

Compárese con la versión sin tipos ricos: un record `{ cuenta: Int, monto: Real, tipo: String }` donde `tipo` es `"debito"` o `"credito"`. Funciona, pero el `tipo: String` permite `"DEBITO"`, `"CR"`, `""`, todas inválidas. Cada función que toca movimientos tiene que validarlo. Con el sum type, el inválido **no se puede construir**.

18.3 La invariante central: cuadre

El módulo `cuadre.kai` es chico pero importante. Define la invariante del ledger de doble entrada: en cada transacción, la suma de los débitos debe igualar la suma de los créditos.

```
pub fn total_debitos(ms: [dominio.Movimiento]) : Real<USD> {
  match ms {
    []                          -> 0.0<USD>
    [dominio.Debito(_, m), ...rest]  -> m + total_debitos(rest)
    [dominio.Credito(_, _), ...rest] -> total_debitos(rest)
  }
}

pub fn total_creditos(ms: [dominio.Movimiento]) : Real<USD> { ... }

pub fn cuadra(ms: [dominio.Movimiento]) : Bool =
  total_debitos(ms) == total_creditos(ms)
```

Tres funciones puras, una invariante boolean. Los tests verifican el contrato pieza por pieza:

```
test "cuadra con una entrada y una salida" {
  let ms = [
    dominio.Debito(1<CuentaId>, 100.0<USD>),
    dominio.Credito(2<CuentaId>, 100.0<USD>),
  ]
  assert cuadra(ms)
}

test "no cuadra cuando los montos difieren" { ... }
test "cuadra con múltiples líneas" { ... }
```

Sin actores, sin IO, sin sockets. Solo lógica. Si en seis meses cambiamos cómo se representan los movimientos, estos tests aseguran que la invariante sigue cumpliéndose.

Y donde el cap. 11 paga: declaramos también una **versión con contrato**:

```
pub fn aplicar_si_cuadra(ms: [dominio.Movimiento]) : [dominio.Movimiento]
  requires cuadra(ms)
  ensures  cuadra(result)
= ms
```

`requires cuadra(ms)` declara que **llamar a esta función con un grupo de movimientos que no cuadran es un error**. Si el compilador puede probarlo estáticamente, rechaza la llamada en compile time; si no, inserta un assert en runtime. El `ensures cuadra(result)` declara que **el resultado de la función también cuadra** (trivial acá: devuelve la misma lista). Esos dos contratos juntos forman la firma legal de la función: las precondiciones que exige y la postcondición que garantiza.

En un sistema contable real, este patrón se replica: cada operación que toca movimientos lleva en su firma los contratos del dominio.

18.4 El almacén: actor con invariantes

El almacén mantiene el estado del ledger: cuentas conocidas, transacciones registradas, contadores de próximos IDs.

```
type Estado = {
  cuentas:        [dominio.Cuenta],
  transacciones:  [dominio.Transaccion],
  proxima_cuenta: Int,
  proxima_tx:     Int,
}
```

La función `procesar` toma un comando y el estado, devuelve una respuesta y el estado nuevo. Lo importante: **toda transacción pasa por validación de cuadre antes de registrarse**.

```
Registrar(desc, movs) ->
  if not cuadre.cuadra(movs) {
    (dominio.ErrorDescuadre("..."), s)
  } else {
    match verificar_cuentas(movs, s.cuentas) {
      Some(id_faltante) -> (dominio.ErrorCuentaInexistente(id_faltante), s)
      None -> {
        let tx = dominio.Transaccion { id: s.proxima_tx<TransaccionId>, ... }
        let cuentas_actualizadas = aplicar_movs(s.cuentas, movs)
        let s2 = Estado { ...s, transacciones: [tx, ...s.transacciones], ... }
        (dominio.TransaccionRegistrada(tx), s2)
      }
    }
  }
```

Dos validaciones antes de aceptar:

1. **Cuadre**: la suma de débitos iguala la suma de créditos.
2. **Cuentas existentes**: todas las cuentas mencionadas en los movimientos están registradas.

Si una falla, devolvemos un error sin modificar el estado. Eso es **transacción atómica**: o se aplica completa, o no se aplica nada. No hay "media transacción registrada con cuadre roto".

Las transacciones **se acumulan, nunca se modifican**. La lista crece. Eso es deliberado: el ledger es por diseño un historial inmutable. Borrar una transacción del pasado no existe; corregir errores es escribir una **nueva** transacción inversa, que también queda en el historial.

Esa inmutabilidad sale gratis en kaikai. Las listas son inmutables por construcción; agregar un elemento crea una lista nueva. No hay "modificación destructiva" disponible para el bucle del actor a menos que se quisiera, con `var` o `Array[T]`. Y como el actor recursa con el estado nuevo, cada "versión" del ledger sobrevive intacta hasta el próximo paso.

18.5 El log de auditoría

El módulo `persistencia.kai` es prácticamente idéntico al del cap. 17: un actor que recibe líneas de log y las agrega al archivo. La diferencia conceptual: para un sistema contable, **el archivo es la verdad**.

En contabilidad real, los registros de transacciones son **append-only**: una vez escrito un asiento, queda. Las correcciones se hacen agregando nuevos asientos inversos, no modificando los originales. Esto es regulatorio (los auditores lo exigen) pero también arquitectónico: un sistema event-sourced que persiste todos los eventos permite reconstruir cualquier estado intermedio.

```
pub type Evento = Linea(String)

fn bucle(path: String) : Unit / Actor[Evento] + File {
  match Actor.receive() {
    Linea(s) -> {
      file.append(path, s ++ "\n")
      bucle(path)
    }
  }
}
```

Cada evento que el almacén produce (cuenta creada, transacción registrada) se manda al log via `Actor.send`. El log lo escribe en orden estricto FIFO. Si la escritura a disco se atrasa, los eventos se acumulan en el mailbox; el almacén sigue respondiendo.

Una mejora para producción que dejamos como ejercicio: en vez de strings, persistir un formato estructurado (JSON, CBOR, TLV) que se pueda volver a leer al arrancar el sistema y reconstruir el estado. Esto es **event sourcing** y kaikai lo permite naturalmente.

18.6 El main: ejecutar un escenario

`main.kai` no abre un socket esta vez: simplemente ejecuta una secuencia de operaciones para mostrar el sistema en acción.

```
fn main() : Unit / Console + File + Spawn + Cancel + ... {
  let almacen_pid = almacen.arrancar()
  let log_pid     = persistencia.arrancar(PATH_LOG)

  paso(almacen_pid, log_pid, dominio.CrearCuenta("caja"))
  paso(almacen_pid, log_pid, dominio.CrearCuenta("ventas"))
  paso(almacen_pid, log_pid, dominio.CrearCuenta("gastos"))

  paso(almacen_pid, log_pid, dominio.Registrar("venta de tarjeta", [
    dominio.Debito(1<CuentaId>, 50.0<USD>),
    dominio.Credito(2<CuentaId>, 50.0<USD>),
  ]))

  paso(almacen_pid, log_pid, dominio.Registrar("café del equipo", [
    dominio.Debito(3<CuentaId>, 8.0<USD>),
    dominio.Credito(1<CuentaId>, 8.0<USD>),
  ]))

  # Intento descuadrado: el almacén lo rechaza.
  paso(almacen_pid, log_pid, dominio.Registrar("error", [
    dominio.Debito(1<CuentaId>, 100.0<USD>),
    dominio.Credito(2<CuentaId>, 50.0<USD>),
  ]))

  paso(almacen_pid, log_pid, dominio.ConsultarSaldo(1<CuentaId>))
  ...
}
```

`paso` es un helper que llama al almacén, imprime la respuesta, y registra el evento en el log de auditoría. Después de correr el `main`, el ledger tiene tres cuentas, dos transacciones registradas (la venta y el café), una rechazada (el intento descuadrado), y los saldos finales reflejan los asientos. El archivo `ledger.audit.log` contiene una línea por cada cuenta y transacción.

18.7 Lo que hace este caso distinto del cap. 17

Mismo patrón general, distinto énfasis. Lo que aparece en este capítulo y no aparecía en el anterior:

- **Unidades de medida.** `Real<USD>` en cada monto. El sistema de tipos rechaza mezclar monedas sin conversión explícita (cap. 10).
- **Branded types.** `Int<CuentaId>` vs `Int<TransaccionId>`. El compilador no nos deja confundirlos (cap. 10).
- **Contratos.** `requires cuadra(ms)` en `aplicar_si_cuadra`. La invariante del dominio queda en la firma de la función, verificable estática o dinámicamente (cap. 11).
- **Inmutabilidad por construcción.** Las transacciones nunca se modifican, solo se agregan. La estructura del ledger garantiza event sourcing.

Lo que aparece igual:

- **Sum types y match exhaustivo.** `Comando`, `Respuesta`, `Movimiento`. El compilador asegura que cada operación trata todos los casos.

- **Actores con estado.** El almacén y la persistencia siguen siendo actores, igual que en el cap. 17.
- **Modularidad.** Tipos puros separados de la maquinaria de actores. La función `procesar` se testea sin fibras.
- **Audit log via actor.** El patrón "un actor encapsula el IO costoso" se repite.

Esa **simetría entre los dos casos** es la lección. Cambia el dominio (HTTP vs contabilidad), cambia qué herramientas del lenguaje pesan más, pero la estructura del programa sigue siendo la misma: dominio puro, actores con estado, persistencia separada, módulos por responsabilidad.

18.8 Cómo extenderlo

Ideas para profundizar el ejemplo, en orden de dificultad:

- **Múltiples monedas.** Declarar `unit EUR`, `unit CLP`, cada cuenta tener su moneda como parámetro de tipo. Las transacciones entre monedas exigen una tasa de cambio explícita (`Real<USD / EUR>`).
- **Tipos de cuenta.** Distinguir activos, pasivos, patrimonio, ingresos, gastos. Cada tipo tiene reglas distintas para qué significa "débito" y "crédito" (en una cuenta de activo, débito suma; en pasivo, débito resta). Branded types como `Cuenta<Activo>` capturan eso.
- **Períodos contables.** Cerrar el ejercicio: todas las cuentas de ingresos y gastos se transfieren a resultado, y el ledger arranca el período nuevo con saldos iniciales.
- **Reconstrucción desde el log de auditoría.** Al arrancar, leer el archivo de log y replicar los eventos para reconstruir el estado en memoria. Esto es event sourcing canónico.
- **Reportes.** Estado de resultados, balance, libro mayor por cuenta. Cada uno es una función pura que recorre las transacciones y produce un string formateado.
- **Validación de cuenta antes de retirar.** Si se modela una cuenta de "saldo no negativo" como `Real<USD> where self >= 0.0<USD>` (refinement type del cap. 11), el sistema de tipos rechaza cualquier movimiento que la deje en negativo.
- **Servir por HTTP.** Combinar este programa con el del cap. 17: el bucle de accept queda igual, el handler que llamaba al almacén ahora llama al almacén del ledger. La estructura del programa cambia muy poco.

Cada una vuelve el sistema más cercano a uno de producción. Ninguna obliga a reescribir las piezas anteriores. La ortogonalidad paga.

18.9 Por qué fintech es un buen banco de pruebas

Fintech es uno de los pocos dominios donde la industria acepta de buen grado que las herramientas pesen más en el proceso de desarrollo. Si una función paga, la empresa prefiere que el compilador la rechace antes de empujarla a producción. Si el sistema garantiza que los débitos cuadran con los créditos, el regulador duerme tranquilo. Si los tipos distinguen monedas, el cambio del próximo trimestre no introduce un bug sutil.

Lenguajes que ofrecen esas garantías (Haskell, OCaml, F#) han tenido buena recepción en fintech histórica. kaikai intenta acercar el mismo nivel de garantías con menos ceremonia:

unidades sin paquetes externos, contratos en la firma, branding sin macros. La promesa es que el código se puede escribir igual de directo que en Python o Go, con las garantías que da el sistema de tipos.

¿Funciona la promesa? Eso lo decide quien escribe el código. Este capítulo intentó mostrar una sección de un dominio donde la apuesta vale la pena.

18.10 Filosofía: el cierre del libro

Hay un patrón que el libro ha venido proponiendo, capítulo a capítulo, sin nombrarlo explícitamente hasta ahora. Vale nombrarlo al final.

El programa real está hecho de pequeñas piezas ortogonales. Tipos puros que describen el dominio. Funciones puras que transforman el dominio. Actores que envuelven el estado mutable que necesita persistir entre llamadas. Fibras que paralelizan trabajo concurrente. Módulos que separan responsabilidades. Contratos que ponen las invariantes del dominio en la firma de las funciones que las preservan.

Cada pieza se prueba en aislamiento. Cada pieza declara en su firma todo lo que hace. Cada pieza puede reemplazarse sin tocar el resto.

Esto no es exclusivo de kaikai. Lo describen, con palabras distintas, *No Silver Bullet* de Brooks, *Simple Made Easy* de Hickey, *Out of the Tar Pit* de Marlow y Goldsmith. Lo que kaikai hace es ofrecer una sintaxis y un sistema de tipos que **vuelven natural** este estilo. Las firmas que no esconden nada salen gratis porque los efectos están en el tipo. Las funciones puras son baratas porque la inmutabilidad es por defecto. Los actores son una biblioteca porque los efectos algebraicos lo permiten. Las invariantes del dominio están en la firma porque los contratos están en el lenguaje.

Si después de leer el libro te quedas con una sola idea, que sea esta: **el lenguaje no es lo que importa; lo que importa es qué te permite construir, y qué te ayuda a evitar construir mal**. kaikai apuesta a que con efectos, fibras, contratos, unidades y holes en su lugar, el programador escribe menos código equivocado y más código que merece estar en producción. Si la apuesta funciona para ti, este libro cumplió su propósito.

Gracias por leer hasta acá. El compilador, el stdlib, los documentos de diseño y los ejemplos viven en `github.com/kaikailang-org/kaikai`. Hay una comunidad emergente, hay issues que cerrar, hay piezas del lenguaje que están todavía tomando forma. Si encuentras este experimento interesante, hay lugar para que ayudes a hacerlo mejor.

Apéndices

Apéndice A · Bootstrap de tres etapas

Este apéndice cuenta cómo se construye el compilador de kaikai desde cero, partiendo de un compilador de C corriente. No es parte del lenguaje que el lector necesita conocer para programar; es parte del relato del proyecto, una decisión de ingeniería con consecuencias durables.

Si te preguntas "¿por qué importa cómo se compila el compilador?", la respuesta corta: porque define a quién le crees y por qué. Un lenguaje cuyo compilador se construye desde una `cc` y nada más es un lenguaje que cualquiera puede auditar, reproducir, y mantener. Eso es libertad técnica de la que pocos lenguajes modernos gozan.

A.1 El problema del bootstrap

Un compilador es un programa. Como cualquier programa, necesita ser compilado para correr. Los compiladores nuevos tropiezan con una paradoja inmediata: ¿qué compila al compilador la primera vez?

Tres respuestas históricas:

- **Escribirlo en C (o ensamblador) la primera vez.** Es la ruta clásica. GCC, MRI Ruby, CPython, V8 nacieron así. El costo: el compilador del lenguaje nuevo carga con la superficie de C para siempre, o se reescribe en sí mismo más tarde con una migración costosa.
- **Escribirlo en un lenguaje existente que ya tenga compilador.** Es lo que hizo Rust con OCaml al principio, lo que hizo Swift con C++. Hereda las dependencias del lenguaje huésped: compilar Rust requiere instalar OCaml, hasta que Rust se reescribió en sí mismo.
- **Self-hosting incremental.** Empezar con un compilador chico (de un subconjunto del lenguaje), escrito en algo portable, usarlo para compilar un compilador más grande (escrito en el lenguaje), y así sucesivamente. Es lo que hace kaikai con sus tres etapas.

A.2 Stage 0: el compilador en C

`stage0` es un compilador escrito en C estándar. Su única dependencia es un compilador de C cualquiera:

```
$ cc stage0/*.c -o kaic0
```

Sin frameworks, sin generadores, sin librerías exóticas. C plano. El archivo `stage0/runtime.h` es el runtime de los programas compilados: contadores de referencia, primitivas de listas y strings, panic. Todo eso entra en unos pocos miles de líneas.

¿Qué compila `kaic0`? **kaikai-minimal**, un subconjunto deliberado del lenguaje. La gramática y las construcciones que caben en kaikai-minimal están documentadas en `docs/kaikai-minimal.md`. Lo que **no** está en kaikai-minimal: efectos algebraicos, fibras, protocolos, contratos, unidades de medida. Lo mínimo para escribir un compilador.

Stage 0 está hecho para ser auditable. Un programador que quiera entender qué es lo que está pasando puede leer el código C línea por línea: lexer, parser recursive-descent, chequeo de tipos simple, emisor de C. Cinco archivos.

A.3 Stage 1: el compilador en kaikai-minimal

Una vez que `kaic0` funciona, escribimos un compilador nuevo **en kaikai-minimal**. Este compilador, `stage1`, hace algo que `kaic0` no puede: compila el lenguaje **completo**. Efectos, fibras, protocolos, contratos, todo.

```
$ kaic0 stage1/main.kai -o kaic1
```

`kaic0` compila `stage1` produciendo un ejecutable `kaic1`. A partir de ahí, el compilador de C ya no participa: `kaic1` es suficiente para procesar programas que usan todo kaikai.

Es la primera "auto-validación": el compilador escrito en kaikai-minimal demuestra que kaikai-minimal es lo bastante expresivo como para implementar un compilador completo. Si no lo fuera, este paso no terminaría.

A.4 Stage 2: kaikai completo, self-hosted

`stage2` es la versión definitiva. Está escrito en **kaikai completo** (no en el subconjunto minimal), usando efectos, fibras, y todo lo que el lenguaje ofrece. Es código kaikai idiomático de extremo a extremo.

```
$ kaic1 stage2/main.kai -o kaic2
```

`kaic1` compila `stage2`, produciendo `kaic2`. Este es el compilador que se distribuye. El usuario que instala kaikai recibe `kaic2`, no `kaic0` ni `kaic1`.

Las tres etapas, de punta a punta:

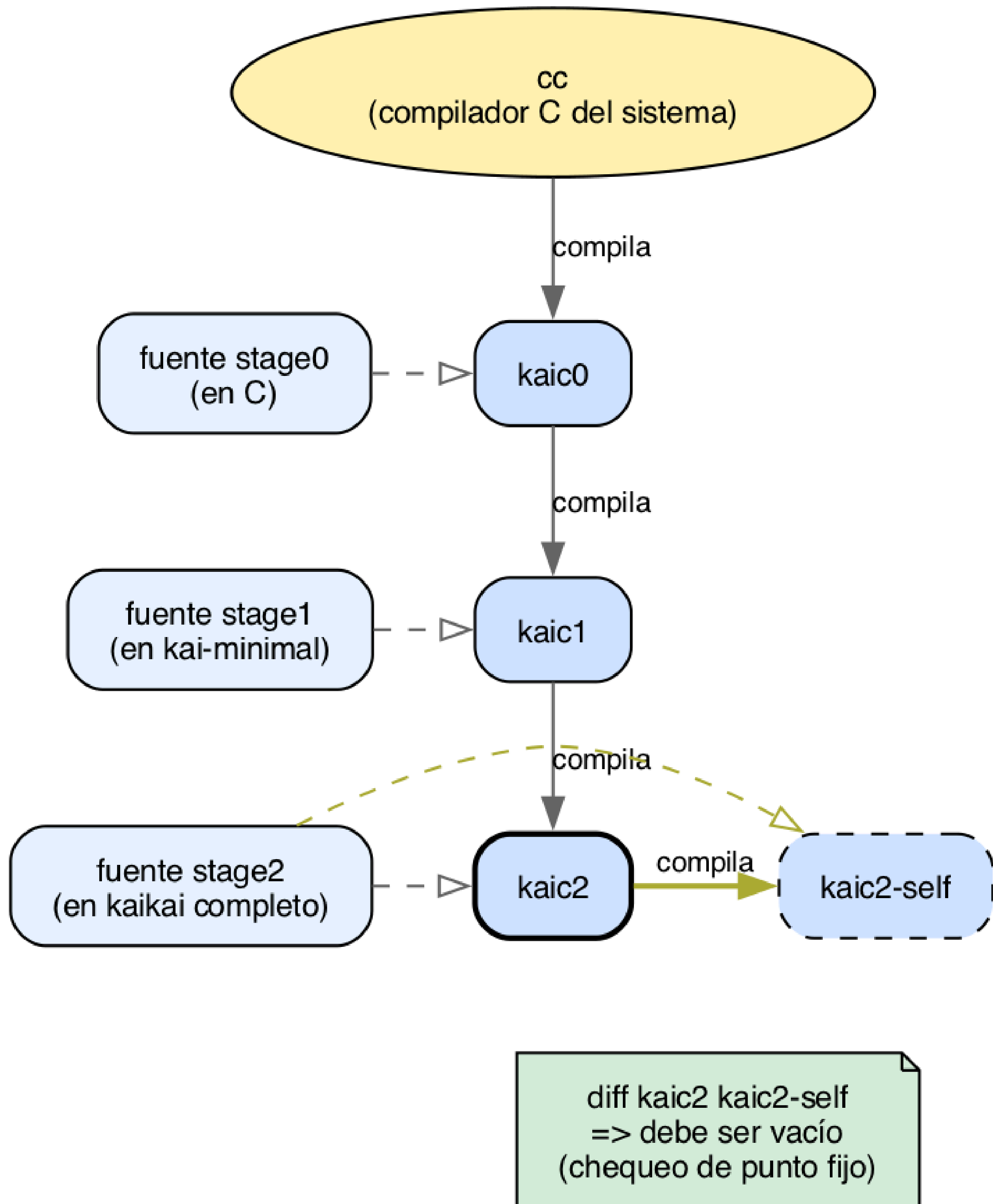

Figura A.1 · *El bootstrap en tres etapas. Las flechas continuas muestran qué compilador produce cada binario; las flechas punteadas van desde el código fuente de cada stage al binario que termina siendo. Solo* `cc` *es externo; una vez existe* `kaic0`*, la cadena se autosustenta. El chequeo de punto fijo de la fila inferior (*`kaic2` *compilándose a sí mismo) es lo que cubre §A.5.*

A.5 El punto fijo: validación del bootstrap

Hay una verificación crítica que hace al final del proceso:

```
$ kaic2 stage2/main.kai -o kaic2-self
$ diff kaic2 kaic2-self
```

Tienen que ser **bit-por-bit idénticos**. Esto significa que `kaic2` compilado por `kaic1` produce exactamente lo mismo que `kaic2` compilado por sí mismo. En otras palabras: `kaic2` es un punto fijo del proceso de compilación.

¿Por qué importa esto? Por dos razones:

- **Detección de errores sutiles en el compilador.** Si dos versiones del compilador (`kaic1` y `kaic2`) producen binarios distintos a partir del mismo código fuente, una de las dos tiene un bug. El punto fijo confirma que la cadena entera converge a la misma respuesta.
- **Resistencia al ataque de Thompson.** Ken Thompson publicó en 1984 un ensayo famoso ("Reflections on Trusting Trust") donde mostraba que un compilador malicioso puede insertar backdoors invisibles que sobreviven a recompilaciones. La defensa clásica es **diverse double-compiling**: compilar con dos cadenas distintas y comparar. El punto fijo es la versión moderna de esa idea: si la cadena entera converge a un mismo binario, y ese binario es reproducible desde fuentes auditables, la confianza está justificada.

A.6 Reproducible desde una `cc`

La consecuencia práctica del bootstrap de tres etapas es esta: **con solo un compilador de C, puedes reconstruir kaikai entero desde el código fuente**. No hace falta tener kaikai previamente instalado. No hace falta confiar en un binario descargado de una página web. No hace falta `curl | bash`.

```
$ git clone https://github.com/kaikailang-org/kaikai
$ cd kaikai
$ make
```

Por dentro, `make` ejecuta los tres pasos: stage 0 con cc, stage 1 con stage 0, stage 2 con stage 1. Termina con un `kaic2` en el directorio `bin/`, y la verificación de punto fijo asegura que es el correcto.

Esto es libertad técnica que pocos lenguajes modernos te dan. Rust requiere un Rust previo (descargado como blob). Go requiere un Go previo. Swift requiere un Swift previo. kaikai no: tu compilador de C es el punto de entrada.

A.7 Costos y trade-offs

El bootstrap de tres etapas tiene costos. Vale enumerarlos porque toda decisión de ingeniería los tiene:

- **Hay que mantener stage 0 cada vez que cambia kaikai-minimal.** Si una característica nueva del lenguaje cae dentro del subconjunto minimal, stage 0 tiene que aprenderla. Eso significa código C nuevo, escrito a mano.

- **Cuesta agregar dependencias al compilador.** Stage 2 podría beneficiarse de una biblioteca externa (parser combinator, estructura de datos exótica). Pero esa biblioteca tendría que estar disponible en stage 1, que solo entiende kaikai-minimal. La presión es hacia mantener stage 2 autosuficiente.
- **Compilar todo desde cero toma tiempo.** No mucho (los tres pasos juntos toman menos de un minuto en una máquina razonable), pero más que un solo `cargo build`. Para CI contínuo, importa.

¿Vale la pena? La respuesta del proyecto es sí, y la razón es filosófica: el compilador es la pieza de software más confiada del ecosistema. Si la cadena de confianza se puede auditar de extremo a extremo, partiendo de C plano, entonces el resto sigue.

A.8 Para profundizar

Las decisiones de diseño viven en `docs/design.md` de `github.com/kaikailang-org/kaikai`. Los archivos físicos están bajo `stage0/`, `stage1/`, `stage2/` del repositorio. Hay un ensayo de Ken Thompson, *Reflections on Trusting Trust* (CACM 1984), que vale la pena leer si te interesa la justificación filosófica de este enfoque.

Apéndice B · Perceus a fondo

El §13.2 cubre la idea base de Perceus en una página: el compilador analiza el programa, sabe en qué punto cada valor deja de usarse, e inserta ahí un `drop` que decrementa el contador de referencias y, si llega a cero, libera la memoria. No hay GC, no hay borrow checker, no hay anotaciones de lifetime.

Este apéndice se mete en el detalle. Para qué sirve: si vienes de Rust y te preguntas por qué kaikai no necesita borrow checker; si vienes de Java y te preguntas por qué kaikai no necesita pausas de GC; o si simplemente quieres entender cómo una idea publicada en 2021 cambió el equilibrio entre RC, GC y ownership.

No hace falta leer este apéndice para programar en kaikai. El modelo de §13.2 alcanza. Este texto es para quien quiere ver los engranajes.

B.1 El paper, en una frase

El paper que inventó Perceus es **"Perceus: Garbage Free Reference Counting with Reuse"** (Reinking, Xie, de Moura, Leijen, PLDI 2021). La frase central:

> *Inserting reference count operations* after *type checking, using* precise last-use information*, makes RC competitive with tracing GC while keeping deterministic deallocation.*

Tres palabras pesan ahí:

- **After type checking.** El análisis se hace sobre el programa ya tipado, no sobre el AST crudo. Esto da información suficiente para razonar sobre forma y unicidad.
- **Precise last-use.** Para cada variable, el compilador identifica el lugar exacto donde se usa por última vez. Después de ese punto, el `drop` es seguro.
- **Reuse.** Es el truco que vuelve a Perceus competitivo con GC: cuando un valor se va a liberar y un valor del mismo shape se va a crear, el compilador reusa la misma memoria.

Vamos por partes.

B.2 Análisis paso a paso: dónde van los drops

Toma esta función simple:

```
fn ejemplo(xs: [Int]) : Int {
  let n = list.length(xs)
  let s = list.sum(xs)
  s + n
}
```

¿Qué tiene que pasar con `xs` cuando la función termina? Depende de quién la creó. Si la lista es propiedad de la función (la creó adentro, o fue movida como argumento), hay que liberarla. Si es compartida con otros, no.

Perceus analiza el cuerpo y encuentra:

- `xs` se usa en la línea 2 (`list.length`).
- `xs` se usa otra vez en la línea 3 (`list.sum`).
- Después no se vuelve a usar.

El compilador inserta:

```
fn ejemplo(xs: [Int]) : Int {
  let n = list.length(dup(xs))   # dup: incrementa rc de xs
  let s = list.sum(xs)           # último uso: pasa ownership
  s + n
}
```

`dup(xs)` incrementa el RC porque `list.length` va a "consumir" una referencia (haciendo su propio drop al final). El segundo uso, `list.sum(xs)`, ya no necesita `dup`: es el último uso, y la referencia que ya tenía la función se transfiere.

`list.sum` y `list.length` por dentro hacen lo mismo: cada vez que recorren la cola de la lista, deciden si están en el último uso. Si lo están, no duplican. Si no, sí.

El programa final tiene **el mínimo número de operaciones posibles** sobre los contadores. Esa minimización es el corazón del paper.

B.3 Reuse in place

La parte más interesante de Perceus es el **reuse in place**. Cuando una función va a liberar un valor de cierto shape y crear inmediatamente otro del mismo shape, el compilador reutiliza el mismo bloque de memoria. No se libera, no se asigna: se sobreescribe.

El ejemplo canónico es el `map` sobre listas:

```
fn map[a, b](xs: [a], f: (a) -> b) : [b] {
  match xs {
    []           -> []
    [h, ...rest] -> [f(h), ...map(rest, f)]
  }
}
```

Sin reuse, este `map` haría:

1. Liberar el cons `[h, ...rest]` (después de extraerlos).

2. Asignar un cons nuevo `[f(h), ...]`.

Con reuse:

1. Reescribir el cons existente con el nuevo head.

Mismo cons, misma posición en memoria, mismo costo que una mutación. Pero **el resultado del programa es idéntico**: la función sigue siendo pura desde el punto de vista del programador.

Las condiciones para reuse in place son tres, todas verificables en tiempo de compilación:

1. **El receptor es de uso único** (RC == 1).
2. **El valor nuevo tiene el mismo shape**: mismo tag, mismos campos.
3. **No hay aliasing en vivo en otra parte**: nadie más tiene un puntero al cons original.

Cuando las tres se cumplen, el compilador emite código que es indistinguible de una mutación destructiva. Pero conceptualmente sigue siendo inmutable: si el programa cambiara y aparecieran dos referencias al cons, el reuse se desactivaría automáticamente y caería al patrón "liberar + asignar". El programador no nota.

Esta optimización es lo que hace que algoritmos como `map`, `filter`, AVL trees, parseo de listas, sean **tan rápidos como las versiones con mutación** en lenguajes imperativos. Es la razón por la que Koka y Lean 4 (que también usan Perceus) pueden compilar competitivo con C.

B.4 Comparación con `Rc<RefCell<T>>` de Rust

Si vienes de Rust, te suena familiar: hay RC en Rust también (`Rc<T>`, `Arc<T>`). ¿En qué se diferencia Perceus?

Aspecto	Rust `Rc<T>`	Perceus en kaikai
Quién decide cuándo incrementar	El programador (clone)	El compilador
Quién decide cuándo decrementar	El destructor automático	El compilador
Anotaciones requeridas	`Rc<T>`, `Arc<T>`, `Rc::clone(&x)`	ninguna
Mutabilidad de contenido	Necesita `RefCell<T>`	Por defecto, inmutable
Ciclos	Memory leak silencioso	Imposibles (sin mutación)
Costo en single-use	Pago el `Rc` siempre	Sin costo: el compilador no emite RC

La diferencia más fuerte es la última. En Rust, cuando declaras `Rc<T>` pagas el contador SIEMPRE, incluso en los casos donde el valor tiene un solo uso. En kaikai, **si el valor tiene un solo uso, no se emite RC**. El compilador inserta los `dup`s solo donde hacen falta, después del análisis de last-use.

Eso significa que un programa kaikai funcional puro, donde nada se comparte de verdad entre varias variables, tiene cero overhead de RC. Las listas, los records, los closures, todos

se liberan al instante en su último uso sin pasar por un contador. El RC solo aparece cuando el programa de verdad necesita compartir.

B.5 Y los ciclos, ¿qué?

La crítica clásica al RC es que **no maneja ciclos**: si dos valores se referencian mutuamente y nadie más los referencia, ninguno baja a 0 nunca, y leakan. Es por eso que Python tiene un "cycle collector" arriba de su RC, y Rust te obliga a usar `Weak` manualmente.

¿Por qué Perceus no necesita ninguno de los dos?

Porque los valores en kaikai son inmutables.

Para crear un ciclo necesitas mutación: A apunta a B, después modificas B para que apunte a A. Sin mutación, no puedes construir el segundo paso. Cuando construyes B, A todavía no existe; cuando construyes A apuntando a B, A no es accesible desde B.

Las únicas excepciones son los mecanismos de mutación que kaikai provee deliberadamente: `var` local (que el compilador enmascara y limita), `Ref[T]`, mailboxes de actores, arrays mutables. Todos ellos tienen disciplinas específicas que evitan ciclos (las mailboxes, por ejemplo, viven en un actor específico y el GC del runtime las maneja).

En la práctica: un programa kaikai típico no construye ciclos. Las estructuras de datos son árboles (listas, AVL, JSON, ASTs). Cuando un programador quiere algo con ciclos (un grafo, un cache con LRU), recurre a estructuras explícitas que codifican la adjacencia sin punteros directos: índices en un array, identificadores en un mapa. Es más trabajo, pero el costo intelectual queda visible donde corresponde.

B.6 Por qué la separación por fibra simplifica el RC

El cap. 13 menciona que cada fibra tiene su propio heap. Esto no es solo aislación de errores: **es lo que hace que Perceus sea libre de locks**.

Si dos fibras compartieran punteros al mismo valor, los incrementos y decrementos del contador tendrían que ser **atómicos**. Atómico significa que la CPU emite una barrera de memoria, sincroniza con otros núcleos, paga overhead. En programas de muchas fibras, ese overhead es enorme.

Cuando cada fibra tiene su propio heap, los contadores son locales. **No hay sincronización**. Un `dup` es un `++` sobre un entero, sin barreras. Un `drop` es un `--`, idem.

Esto es por qué el modelo de fibras de kaikai y Perceus se diseñan juntos: cada uno habilita al otro. Las fibras aisladas permiten RC sin sincronización; el RC eficiente permite millones de fibras sin pagar GC.

Cuando dos fibras necesitan compartir un valor, lo hacen via una operación explícita (`send` a un mailbox, devolver de un `await`). El runtime copia el valor (o lo mueve si es seguro) al heap de la fibra receptora. La transferencia es explícita en el programa, y por eso predecible.

B.7 Lo que cuesta y lo que se gana

Perceus tiene costos. Vale enumerarlos:

- **El análisis estático es trabajo del compilador.** Más lento que tipos crudos. En kaikai, el análisis está integrado con la inferencia y se ejecuta en milisegundos para programas típicos.
- **Cuando un valor se duplica de verdad, paga el RC.** Si tienes muchas estructuras compartidas pesadas, el `dup` / `drop` cuesta.
- **Reuse in place depende de la unicidad estática.** Si el análisis no puede probar unicidad, la optimización no aplica y caes a la versión segura.

Lo que se gana:

- **Determinismo.** Sabes exactamente cuándo se libera cada valor. Sin pausas, sin "GC corrió ahora", sin no-determinismo entre runs.
- **Memory predictability.** El peak de memoria se puede estimar mirando el programa. No hay "el GC esperó demasiado y se acumuló".
- **Cero anotaciones.** Sin `'a`, sin `&mut`, sin `Rc::clone(&x)`. El compilador hace el trabajo.
- **Composición con efectos.** El RC no interactúa con efectos en formas raras. `handle` y `resume` no tienen overhead oculto de GC.

B.8 Para seguir

Si este apéndice te dejó con ganas de más, las fuentes:

- **Reinking, Xie, de Moura, Leijen. "Perceus: Garbage Free Reference Counting with Reuse"**. PLDI 2021. El paper.
- **Lorenz, Leijen. "Reference Counting with Frame Limited Reuse"**. ICFP 2023. Una extensión que mejora reuse cuando el shape no calza exactamente.
- **Koka language documentation** (Daan Leijen et al.). Koka es el lenguaje donde Perceus nació; mucho del vocabulario ("reuse in place", "borrowed binds", "drop specialisation") viene de ahí.
- **Lean 4 RC implementation** (de Moura et al.). Lean 4 también usa Perceus, con un enfoque más cercano a certificación formal.

En el código de kaikai, el lugar donde vive el análisis es `stage2/perceus.kai`. Si quieres ver cómo se implementa, ese es el punto de partida.

Apéndice C · Tabla de operadores y precedencia

Este apéndice resume todos los operadores del lenguaje con su precedencia y asociatividad. Está pensado como referencia rápida: cuando dudas cómo parsea `a | f |> g`, lo consultas y sigues.

La precedencia está numerada de más alta (1, aprieta primero) a más baja (8, aprieta último). Cada nivel se resuelve contra el nivel inmediatamente superior.

C.1 Tabla principal

<table>
<tr><th>Nivel</th><th>Operadores</th><th>Asociatividad</th></tr>
<tr><td>1</td><td>llamada f(args), campo ., índice [i]</td><td>Postfija</td></tr>
<tr><td>2</td><td>unario -, not</td><td>Prefija</td></tr>
<tr><td>3</td><td>*, /, //, %</td><td>Izquierda</td></tr>
<tr><td>4</td><td>+, - (binarios), ++ (concat de strings)</td><td>Izquierda</td></tr>
<tr><td>5</td><td>==, !=, <, >, <=, >=</td><td>No-asociativa</td></tr>
<tr><td>6</td><td>and</td><td>Izquierda (corto)</td></tr>
<tr><td>7</td><td>or</td><td>Izquierda (corto)</td></tr>
<tr><td>8</td><td>|>, |, ||, |?</td><td>Izquierda</td></tr>
</table>

Detalles por nivel

- **Nivel 1 (postfijos).** Se asocian por izquierda: `a.b.c` significa `(a.b).c`; `f(x)(y)` significa `(f(x))(y)`.
- **Nivel 2 (prefijos).** Unario `-x` y `not x`. Aplican antes que cualquier operador binario.
- **Nivel 3 (multiplicativos).** `*` y `/` sobre `Int` y `Real`, `//` es división entera, `%` es resto. Asociación izquierda: `a / b / c` es `(a / b) / c`.
- **Nivel 4 (aditivos).** `+` y `-` binarios sobre números, `++` para concatenar strings.
- **Nivel 5 (comparaciones). No se encadenan.** Escribir `a < b < c` es error de sintaxis. La forma correcta es `a < b and b < c`. Esto elimina la clase de bugs donde `1 == 1 == true` parsea inesperadamente.
- **Nivel 6 (`and`) y 7 (`or`).** Lógicos con corto-circuito. `and` aprieta más fuerte que `or`, así que `a or b and c` es `a or (b and c)`.

- **Nivel 8 (pipes).** Cuatro operadores al mismo nivel: `|>` (apply), `|` (map), `||` (flat--map), `|?` (filter). Todos asocian a izquierda. `xs | f |> g` es `(xs | f) |> g`.

C.2 Operadores que no son operadores

Algunas formas que parecen operadores en otros lenguajes son construcciones de sintaxis aparte en kaikai:

- **`=` (asignación de `let`)**: parte de la sintaxis de `let`, no es un operador. `let x = expr` es una declaración.
- **`:=` (escritura a celda mutable)**: parte de la sintaxis de `var`. `nombre := expr` es la forma azucarada de `State.set(nombre, expr)`. No participa en la tabla de precedencia.
- **`@nombre` (lectura de celda mutable)**: lectura azucarada de `State.get(nombre)`. Es un prefijo del nombre, no participa en la tabla.
- **`->` (en `match`, en lambdas, en `handle`)**: separador sintáctico, no operador.
- **`/` (en firma de efecto)**: separa el tipo de retorno de la fila de efectos: `fn f() : Int / Log`. Solo aparece en posición de firma.
- **`!` (propagación de `Option`/`Result`)**: postfijo sobre expresiones que devuelven `Option[T]` o `Result[E, T]`, hace propagación temprana del error o el `None`. Equivale más o menos al `?` de Rust. No es un operador binario.
- **`?` y `?nombre` (holes)**: marcadores de agujero, no operadores.
- **`...` (spread)**: aparece en literales de record y de lista (`[h, ...t]`, `Punto { ...p, x: 10 }`); también en patrones. No es operador.
- **`<>` (anotación de unidad de medida)**: 1.50<USD>, Real<EUR>. La unidad va entre los paréntesis angulares; no son los operadores `<` y `>`.

C.3 Equivalencias útiles

Algunos operadores son azúcar sintáctica para llamadas a funciones del stdlib. Conocer las equivalencias ayuda cuando el operador no parece funcionar o cuando uno quiere ver el mecanismo subyacente.

Operador	Equivale a
`x	> f`
`x	> f(_, b)`
`xs	f`
`xs	
`xs	? p`
a ++ b	string_concat(a, b) (cuando son String)
a[i]	Mutable.array_get(a, i)
a[i] := v	Mutable.array_set(a, i, v)
@nombre	State.get(nombre) (con var)
nombre := v	State.set(nombre, v) (con var)

C.4 Recordatorio: comparación no asociativa

La regla del nivel 5 merece énfasis aparte porque es donde el lector que viene de otros lenguajes más se sorprende.

En la mayoría de los lenguajes:

```
1 == 1 == 1     # Python: false (1 == 1 → true, true == 1 → false)
1 < 2 < 3       # C: 1 (1 < 2 → 1, 1 < 3 → 1; no es lo que parece)
```

En kaikai, ninguna de las dos compila. El compilador exige que escribas la intención explícitamente:

```
1 == 1 and 1 == 1      # claramente: dos comparaciones
1 < 2 and 2 < 3        # rango: cada lado se compara explícito
```

Esto elimina una clase pequeña pero molesta de bugs. Para quien venga de Python y le moleste tener que escribir más, vale recordar que la regla es la misma de Pascal, Eiffel y SQL. No es una rareza de kaikai.

Apéndice D · Catálogo de efectos del stdlib

Este apéndice resume los efectos que el stdlib expone. Es material de referencia: ante una firma `: Unit / X` en la documentación de una función, vienes acá a confirmar qué provee `X`.

La especificación completa vive en `github.com/kaikailang-org/kaikai/docs/effects-stdlib.md`. Acá mostramos la declaración del efecto y para qué sirve.

D.1 IO básico

`Console`

```
effect Console {
  print(s: String)  : Unit
  eprint(s: String) : Unit
}
```

Imprimir a stdout y stderr. Cada operación agrega un newline. El handler por defecto del runtime escribe al descriptor de archivo correspondiente.

`Stdin`

```
effect Stdin {
  read_line() : Result[String, String]
}
```

Leer una línea de entrada estándar. Devuelve `Err(motivo)` en caso de EOF o error de lectura.

`Env`

```
effect Env {
  get(name: String) : Option[String]
  args()            : [String]
}
```

Acceso a variables de entorno (`PATH`, `HOME`, etc.) y a los argumentos de línea de comando (`argv`).

`File`

```
effect File {
  read_file(path: String)                 : Result[String, String]
  write_file(path: String, content: String): Result[Unit, String]
  append(path: String, content: String)   : Result[Unit, String]
  exists(path: String)                    : Bool
  delete(path: String)                    : Result[String, Unit]
  rename(from: String, to: String)        : Result[String, Unit]
}
```

Operaciones sobre archivos. Para todo lo que sea no-trivial (streams, directorios, permisos), ver `fs.dir` y los módulos auxiliares.

D.2 Tiempo y aleatoriedad

`Clock`

```
effect Clock {
  now()           : Int         # nanosegundos desde epoch
  sleep(ms: Int)    : Unit / Cancel
}
```

Reloj y sleep. `sleep` es punto de yield (carga `Cancel`).

`Random`

```
effect Random {
  int(min: Int, max: Int)   : Int
  real()                    : Real
  shuffle[a](xs: [a])       : [a]
}
```

Generación pseudo-aleatoria, no apta para criptografía. Para secretos, ver `SecureRandom`.

`SecureRandom`

```
effect SecureRandom {
  bytes(n: Int) : [Byte]
}
```

Bytes aleatorios criptográficamente seguros (vía `/dev/urandom` o equivalente del sistema).

D.3 Red

`NetTcp`

```
effect NetTcp {
  connect(host: String, port: Int)   : Result[Conn, String]
```

```
  listen(host: String, port: Int)    : Result[Listener, String]
  accept(l: Listener)                : Result[Conn, String]
  send(c: Conn, data: [Byte])        : Result[Int, String]
  recv(c: Conn, max: Int)            : Result[[Byte], String]
  close(c: Conn)                     : Unit
}
```

Sockets TCP byte-level. Las operaciones bloqueantes (`connect`, `accept`, `send`, `recv`) suspenden la fibra vía el reactor del runtime cuando aterrice (v1 las hace bloqueantes al OS thread).

`NetUdp` y `NetDns`

UDP (`bind` / `send` / `recv` / `close`) y DNS (`resolve`). Mismo estilo que `NetTcp`. El alias `Net = NetTcp + NetUdp + NetDns` es útil cuando una función usa los tres.

D.4 Procesos y señales

`Process`

```
effect Process {
  run(cmd: String, args: [String]) : Result[ProcessResult, String]
  pid()                            : Int
}
```

Ejecutar comandos externos como subprocesos. `ProcessResult` contiene `exit_code`, `stdout` y `stderr`.

`Signal`

```
effect Signal {
  handle(sig: SignalKind, action: () -> Unit) : Unit
}
```

Registrar handlers para señales del sistema (`SIGINT`, `SIGTERM`, etc.). Útil para limpieza ordenada al apagar el proceso.

D.5 Estado

`State[T]`

```
effect State[T] {
  get() : T
  set(v: T) : Unit
}
```

El efecto canónico para llevar estado mutable de forma encapsulada. La sintaxis `var nombre = init` es azúcar sobre un `handle` de `State[T]` (cap. 12 §12.7).

`Reader[T]` y `Writer[W]`

```
effect Reader[T] {
  ask() : T
}

effect Writer[W] {
  tell(w: W) : Unit
}
```

Entorno de lectura (configuración inmutable) y acumulación de salida (logging, traza). Patrones clásicos del cálculo de efectos.

`Mutable`

```
effect Mutable {
  ref_make[T](init: T)      : Ref[T]
  ref_get[T](r: Ref[T])     : T
  ref_set[T](r: Ref[T], v: T): Unit
  array_make[T](n: Int, init: T) : Array[T]
  array_length[T](a: Array[T])   : Int
  array_get[T](a: Array[T], i: Int)        : T
  array_set[T](a: Array[T], i: Int, v: T)  : Unit
  array_grow[T](a: Array[T], n: Int, init: T): Unit
}
```

El efecto detrás de `Ref[T]` y `Array[T]`. Sigue la disciplina de **efectos observables** (cap. 12 §12.7): un `array_set` requiere `Mutable` en la fila solo cuando la mutación es visible para quien llama. Un array creado local y devuelto no requiere `Mutable`.

D.6 Errores y control

`Fail`

```
effect Fail {
  fail(msg: String) : Nothing
}
```

Abortar con un mensaje. La operación devuelve `Nothing` (el tipo vacío), así que el sistema de tipos garantiza que no se puede llamar a `resume` después de `Fail.fail`. Es el patrón canónico para "excepción ligera" en kaikai.

`Cancel`

```
effect Cancel {
  raise() : Nothing
}
```

Cancelación cooperativa. El scheduler inyecta `Cancel.raise()` en una fibra cancelada en el próximo punto de yield. La fibra puede instalar un handler de `Cancel` para limpieza (cap. 13).

D.7 Concurrencia

`Spawn`

```
effect Spawn {
  spawn[T, e](f: () -> T / e) : Fiber[T]
  await[T](f: Fiber[T])       : T
  select[T](fs: [Fiber[T]])   : T
  yield()                     : Unit
  cancel[T](f: Fiber[T])      : Unit
}
```

Crear fibras, esperarlas, racear, ceder, cancelar. `nursery { n -> ... }` del cap. 13 es azúcar sobre `handle ... with Spawn as n { ... }`.

`Actor[Msg]`

```
effect Actor[Msg] {
  self()                         : Pid[Msg]
  send(pid: Pid[Msg], msg: Msg)  : Unit / Cancel
  receive()                      : Msg / Cancel
}
```

El efecto que da forma al modelo de actores del cap. 14. `with_mailbox { ... }` y `spawn_actor(...)` son los wrappers del stdlib que instalan este handler.

`Link` y `Monitor`

```
effect Link {
  link(pid: Pid[_]) : Unit
}

effect Monitor {
  monitor(pid: Pid[_])         : MonitorRef
  demonitor(ref: MonitorRef)   : Unit
}
```

Supervisión al estilo BEAM (cap. 14 §14.6). Links son bidireccionales: si uno cae, el otro recibe `Cancel.raise()`. Monitores son unidireccionales: el observador recibe un mensaje `MonitorDown` cuando el observado termina.

D.8 Interoperabilidad

`Ffi`

```
effect Ffi
```

El efecto que cargan todas las funciones declaradas con `extern "C" fn`. Sin operaciones propias: es un marcador para que el sistema de tipos sepa qué funciones tocan código no auditado por kaikai. El capítulo 16 §16.9 cubre la sintaxis de declaración, el mapeo de tipos en el borde, el enlazado con shims C, y qué soporta y qué no FFI v1.

D.9 Composición: el alias `Io`

```
type Io = Console + Stdin + Env + File
```

Bundle de los efectos más comunes para IO al sistema operativo. Una función que dice `/ Io` está declarando que puede tocar consola, leer stdin, leer variables de entorno y manipular archivos. Es el equivalente a "esta función no es pura, hace cosas con el sistema".

D.10 Handlers por defecto

Cuando `main` declara uno de estos efectos en su fila, el runtime instala automáticamente un handler por defecto:

- `Console`, `Stdin`, `Env`, `File` → IO al sistema.
- `Clock`, `Random`, `SecureRandom` → reloj y RNG del sistema.
- `NetTcp`, `NetUdp`, `NetDns` → POSIX sockets.
- `Process`, `Signal` → llamadas POSIX.
- `Mutable` → asignaciones reales en heap.
- `Spawn`, `Cancel` → el scheduler de fibras del runtime.

Estos handlers se pueden interceptar: cualquier `handle ... with X { ... }` que el usuario instale gana sobre el handler del runtime para todo el bloque del `body`. Eso es lo que permite mocking en tests, capturar la salida, simular el reloj, etc.

Apéndice E · Glosario

Términos que el libro usa con un significado específico, con su equivalencia en inglés cuando corresponde. Los términos en inglés están así a propósito: son los nombres que usan la documentación del lenguaje y la comunidad de los lenguajes funcionales, y traducirlos confunde más que ayuda. El glosario los nombra para que el lector que entra desde otro lenguaje encuentre rápido a qué se refieren.

A

Actor. Una fibra con un mailbox tipado encima. Procesa mensajes que recibe en orden, mantiene estado interno entre mensajes. En kaikai, los actores son una biblioteca construida sobre el efecto `Actor[Msg]`.

Algebraic effect *(efecto algebraico)*. Un efecto en el sentido del cap. 12: una construcción del lenguaje que declara operaciones (con su firma) sin decidir cómo se implementan, y permite que un `handle` decida en cada punto del programa qué significa cada operación.

Array *(arreglo)*. Estructura de datos con acceso por índice en tiempo constante. En kaikai, `Array[T]` es mutable; la mutación está bajo el efecto `Mutable`.

Assert *(aserción)*. Construcción dentro de bloques `test`, `check`, contratos. Si la condición es `false`, el bloque o el programa abortan.

B

Backpressure *(retropresión)*. Mecanismo para que un consumidor lento haga frenar a un productor rápido. En kaikai, la policy `Bounded(N, BlockSender)` de un mailbox bloquea el emisor cuando el mailbox está lleno.

Bottom type *(tipo bottom)*. Tipo sin habitantes, escrito `Nothing`. Una función que devuelve `Nothing` no puede regresar normalmente: o no termina, o aborta, o llama a un efecto que no devuelve. Por eso `Fail.fail(...) : Nothing` es la firma natural de una operación que no resume.

Bounded *(acotado)*. Una policy de mailbox con capacidad fija. Cuando se llena, el comportamiento depende de la regla de overflow: `DropOldest`, `DropNewest` o `BlockSender`.

Branded type *(tipo con marca)*. Un tipo numérico o string marcado con una unidad simbólica (`Int<UserId>`) para que el sistema de tipos lo distinga de otros tipos con la misma representación. Caso particular del sistema de unidades de medida.

C

Cancellation *(cancelación)*. El acto de pedirle a una fibra que termine antes de tiempo. En kaikai es un efecto: `Cancel.raise()`. La fibra puede instalar un handler para hacer limpieza antes de desenrollarse.

Capability *(capacidad)*. El binding que un `handle ... with Effect` le da al body para invocar las operaciones del efecto. Por defecto, el nombre de la capacidad es el del efecto: dentro de un handler de `Log`, se llama `Log.log(...)`. La sintaxis `with X as nombre` permite renombrarla.

Closure *(clausura)*. Un valor de función que captura variables del scope donde se creó.

Continuation *(continuación)*. El "resto de lo que falta hacer" en un punto del programa. En kaikai aparece como el argumento `resume` que reciben los handlers de efectos: llamar a `resume(v)` continúa el cómputo del body con `v`.

Contracts *(contratos)*. `requires` (precondición) y `ensures` (postcondición) en la firma de una función, verificados estáticamente cuando es posible y dinámicamente cuando no. Cap. 11.

D

Default handler *(handler por defecto)*. El handler que el runtime instala automáticamente alrededor de `main` para ciertos efectos (`Console`, `File`, `Spawn`, etc.). Los handlers del usuario instalados con `handle ... with X` se imponen sobre el handler por defecto mientras están en scope.

Doble entrada *(double-entry, en contabilidad)*. Sistema contable donde cada transacción tiene débitos que igualan los créditos. Aparece en el caso de estudio del cap. 18.

Drop *(liberar)*. La operación que Perceus inserta en el último uso de cada valor para decrementar su contador de referencia. Si llega a cero, la memoria se libera.

E

Effect *(efecto)*. Ver *Algebraic effect*.

Effect row *(fila de efectos)*. La lista de efectos en una firma: `Int / Log + Fail + State[Int]`. Se construye con `+` y se trata como un conjunto, no como una secuencia.

Event sourcing. Patrón arquitectónico donde el estado del sistema es la suma de los eventos ocurridos; el log de eventos es la fuente de verdad y el estado en memoria se reconstruye replicándolo. Aparece en el cap. 18.

Exhaustiveness *(exhaustividad)*. Propiedad que el compilador verifica en `match`: que todos los habitantes del tipo del escrutinio estén cubiertos. Cap. 5.

F

Fiber *(fibra)*. Unidad de ejecución cooperativa. En kaikai una fibra es liviana (cientos de bytes), tiene su propio heap, no comparte memoria con otras fibras, y solo cede el control en puntos de yield explícitos. Cap. 13.

Function coloring problem. El problema donde una característica del lenguaje (típicamente `async/await`) parte las funciones en dos colores incompatibles, y cambiar una función contagia a todas las que la llaman. Aparece en el ensayo *"What Color is Your Function?"* de Bob Nystrom (2015), y kaikai lo resuelve metiendo todo en filas de efectos.

H

Handler. El bloque `with Effect { ... }` que decide qué hacer con las invocaciones a un efecto. Para cada operación recibe los argumentos y un `resume`, y decide si continuar el body o no.

Hole *(agujero, hueco)*. Una expresión `?` o `?nombre` que compila pero aborta en runtime si la ejecución llega a ella. Sirve para diseño top-down (humano) y como interfaz para agentes IA. Cap. 15.

I

Inmutabilidad por defecto. Los valores en kaikai son inmutables por construcción: `let x = ...` declara un binding que no cambia. Las construcciones mutables (`var`, `Ref[T]`, `Array[T]`) son la excepción explícita.

L

Last-use analysis *(análisis de último uso)*. La fase del compilador donde se identifica, para cada variable, el punto exacto en el que se usa por última vez. Perceus usa este análisis para insertar `drop`s en el lugar correcto.

Linked *(enlazado, en actores)*. Dos actores enlazados con `Link.link(pid)` se enteran mutuamente cuando uno termina: si uno cae, el otro recibe `Cancel.raise()`. Cap. 14.

M

Mailbox *(buzón)*. La cola de mensajes asociada a un actor. El actor procesa los mensajes en orden FIFO. La **policy** del mailbox decide qué hacer cuando se llena (unbounded, drop-oldest, drop-newest, block-sender).

Match. Expresión de pattern matching. Cubre todos los constructores de un sum type (exhaustivo) y permite extraer componentes de records y listas. Cap. 5.

Monitor *(monitor, en actores)*. Un actor que monitorea a otro recibe un mensaje `MonitorDown` cuando el monitoreado termina, sin acoplar su propia vida a la del observado. Cap. 14.

MVS *(minimum-version selection)*. Algoritmo de resolución de dependencias del gestor de paquetes. Cuando un proyecto declara `manutara@v0.1.0` y otro `manutara@v0.2.0`, MVS elige la **máxima** de las versiones declaradas. Cap. 8.

N

Nothing. Ver *Bottom type*.

Nursery *(guardería).** Un scope léxico que contiene fibras hijas y garantiza que ninguna sobreviva al bloque. Se construye con `nursery { n -> ... }`, que es azúcar sobre `handle ... with Spawn as n { ... }`. Cap. 13.

O

Operation *(operación)*. La declaración nombrada dentro de un `effect`: un nombre, parámetros, tipo de retorno. Las operaciones se invocan con `Effect.op(args)`.

P

Pattern matching. La construcción `match` y los patrones en `let`. Permite descomponer valores estructurados (sum types, records, listas) en componentes. Cap. 5.

Perceus. El sistema de reference counting estático que kaikai usa para liberar memoria sin GC ni borrow checker. Inventado por Reinking, Xie, de Moura y Leijen (PLDI 2021). Apéndice B y cap. 13 §13.2.

Pid *(process id, identificador de proceso)*. Handle tipado de un actor: `Pid[Msg]`. Identifica un mailbox y también garantiza que solo se pueden enviar mensajes de tipo `Msg` a ese mailbox.

Pipe *(pipe, tubería)*. Operadores `|>`, `|`, `||`, `|?`. Encadenan transformaciones. Cap. 6.

Polymorphism *(polimorfismo)*. La capacidad de una función de operar sobre múltiples tipos. En kaikai aparece como generics (`fn map[a, b](xs: [a], f: (a) -> b) : [b]`) y como filas polimórficas (`/ e` donde `e` es una variable de fila).

Protocol *(protocolo)*. Una interfaz declarada con `protocol`, implementada por tipos vía `impl Protocol for T`. Es single-dispatch (resuelve por un solo tipo). Cap. 9.

Pure *(puro)*. Una función es pura si no produce efectos (su fila está vacía). Las funciones puras son fáciles de testear, paralelizar y razonar.

R

Ref *(referencia)*. Una celda mutable con sobrevida arbitraria, accesible vía `Mutable.ref_make` / `Mutable.ref_get` / `Mutable.ref_set`. Cap. 12 §12.7.

Refinement type *(tipo refinado)*. Un tipo con un predicado restrictivo: `Int where self >= 0`. Cap. 11.

Resume. Ver *Continuation*.

Reuse in place *(reutilización en sitio)*. Optimización de Perceus: cuando un valor único se va a liberar y se va a crear otro del mismo shape inmediatamente después, se reusa la misma memoria sin tocar contadores. Apéndice B.

Row variable *(variable de fila)*. Una variable polimórfica que representa "el resto de los efectos" en una firma: `fn map[A, B, e](xs: [A], f: (A) -> B / e) : [B] / e`.

S

Self-hosting. Estado en el que un compilador está escrito en el mismo lenguaje que compila. El compilador `kaic2` de kaikai está escrito en kaikai. Apéndice A.

Span. El rango de bytes en el archivo fuente donde vive una construcción. Los mensajes del compilador usan spans para señalar dónde ocurrió un error.

Spawn. Crear una fibra o un actor nuevo. Operación del efecto `Spawn`.

Stage 0/1/2. Los tres compiladores que forman el bootstrap de kaikai. Stage 0 en C, stage 1 en kaikai-minimal, stage 2 en kaikai completo. Apéndice A.

State[T]. Efecto del stdlib para llevar estado mutable encapsulado. La forma azucarada `var nombre = init` desazucara a `handle ... with State[T](init)` (cap. 12 §12.7).

Stdout, Stderr, Stdin. Salida estándar, salida de error estándar, entrada estándar. En kaikai estos están bajo el efecto `Console` (para los dos primeros) y `Stdin` (para el último).

Sum type *(tipo suma, tipo algebraico de datos)*. Un tipo con varios constructores, cada uno cargando datos distintos. La construcción `type Forma = Circulo(Real) | Cuadrado(Real)`. Cap. 5.

T

Tail call *(llamada de cola)*. Una llamada a función que está en la posición de "lo último que hace la función actual". El compilador la compila a un salto, no a una llamada con push de stack, así que la recursión por cola corre sin consumir memoria de pila. Cap. 6.

Top-down design *(diseño descendente)*. Estilo de diseño en el que se empieza por la firma de las funciones de nivel superior, dejando holes en los cuerpos, y se completan las piezas internas después. Cap. 15.

Trap exit. Capacidad de un actor de **no** propagarse automáticamente cuando una fibra hermana cae. Se activa con `fiber_set_trap_exit(true)`; el actor recibe un mensaje informativo en vez de un `Cancel.raise()`. Cap. 14.

U

Unidad de medida. Una unidad simbólica declarada con `unit` que anota a un valor numérico (`Real<USD>`, `Int<Seconds>`). El sistema de tipos rechaza operaciones que mezclan unidades incompatibles. Cap. 10.

V

Var. Construcción de declaración de celda mutable local. Es azúcar sobre `State[T]`. Cap. 12 §12.7.

Y

Yield *(ceder)*. El acto en que una fibra le pasa el control al scheduler para que otra fibra pueda correr. `fiber_yield()` lo hace explícitamente; las operaciones de IO y `Spawn.await` lo hacen implícitamente.

Apéndice F · Para seguir

El libro cubrió kaikai pero apenas tocó la familia de ideas sobre las que se construye. Si te interesa profundizar, esta es una lista corta de fuentes que vale la pena leer, agrupadas por tema. No pretende ser exhaustiva; pretende ser útil.

F.1 Efectos algebraicos

La pieza del lenguaje que más merece lectura externa es la de efectos algebraicos. La literatura es accesible y vale la pena ir al original.

- **Plotkin, Pretnar, *"Handlers of Algebraic Effects"*** (ESOP 2009). El paper que introduce los handlers como los conocemos. Técnico pero corto.
- **Bauer, Pretnar, *"Programming with Algebraic Effects and Handlers"*** (J. Logical and Algebraic Methods in Programming, 2015). Más pedagógico que el anterior; bueno como segunda lectura.
- **Lenguaje Koka** (Daan Leijen, Microsoft Research). `koka-lang.github.io`. Probablemente el lenguaje con la mejor implementación de efectos algebraicos hoy. Mucha de la inspiración sintáctica de kaikai viene de ahí.
- **Lenguaje Effekt.** `effekt-lang.org`. Tiene un sistema de efectos basado en capabilities. Comparable a Koka pero con otra estética.
- **Eduardo Díaz, *Revelaciones*** (lnds.net, 2015, `https://lnds.net/blog/lnds/2015/10/01/revelaciones/`). La previa histórica al camino de kaikai: teoría de categorías y mónadas como puente desde el mundo funcional clásico hacia los efectos algebraicos. Mencionado en el prólogo.

F.2 Perceus y reference counting

- **Reinking, Xie, de Moura, Leijen, *"Perceus: Garbage Free Reference Counting with Reuse"*** (PLDI 2021). El paper que inventa el sistema que kaikai usa para liberar memoria. Es legible.
- **Lorenz, Leijen, *"Reference Counting with Frame Limited Reuse"*** (ICFP 2023). Extensión que mejora reuse cuando el shape no calza exactamente.
- **Lean 4** (`leanprover.github.io`). Sistema de pruebas que usa una variante de Perceus en su runtime. Es más complejo que Koka pero también está bien documentado.

F.3 Modelo de actores y BEAM

- **Joe Armstrong, *Programming Erlang*** (Pragmatic Bookshelf, 2007/2013). El libro canónico de Erlang escrito por su creador. Cubre la filosofía de "let it crash" con más profundidad que cualquier introducción reciente.
- **Saša Jurić, *Elixir in Action*** (Manning, 2019). El modelo de actores explicado para programadores modernos, con código Elixir. La parte de OTP es excelente.
- **Cesarini, Vinoski, *Designing for Scalability with Erlang/OTP*** (O'Reilly, 2016). Para cuando quieras pensar en sistemas distribuidos serios.

F.4 Concurrencia estructurada

- **Nathaniel J. Smith, *"Notes on structured concurrency, or: Go statement considered harmful"*** (vorpus.org, 2018). El ensayo seminal sobre concurrencia estructurada. Lectura obligatoria si vas a escribir cualquier código concurrente en cualquier lenguaje moderno.
- **Bob Nystrom, *"What Color is Your Function?"*** (journal.stuffwithstuff.com, 2015). El ensayo sobre el problema de los colores que motivó la apuesta de kaikai por efectos en vez de `async`/`await`. Corto, divertido, con tirón.
- **Trio (Python)**, **Kotlin coroutines**, **Swift structured concurrency**, **OCaml 5 Eio**. Cuatro implementaciones concretas del mismo modelo. Comparar cómo lo expresa cada uno ayuda a fijar la idea.

F.5 Diseño de lenguajes

- **Fred Brooks, *"No Silver Bullet"*** (IEEE Computer, 1986). El ensayo clásico sobre la complejidad esencial vs accidental en software. Sigue vigente cuarenta años después porque acertó.
- **Rich Hickey, *"Simple Made Easy"*** (Strange Loop, 2011, video en infoq.com). Una hora de Rich Hickey distinguiendo *simple* de *easy*. Cambia la forma de evaluar APIs y lenguajes.
- **Marlow, Goldsmith et al., *"Out of the Tar Pit"*** (paper 2006, accesible online). Diagnóstico de por qué el software se vuelve complicado y propuesta de cómo evitarlo. Muy influyente en el pensamiento funcional moderno.
- **Steele, Sussman, *"Lambda: The Ultimate Imperative"*** (MIT AI Memo, 1976). Uno de los papers fundadores que muestra que la programación funcional con clausuras y recursión cubre todo lo que los lenguajes imperativos hacen.

F.6 Sistemas de tipos

- **Benjamin Pierce, *Types and Programming Languages*** (MIT Press, 2002). El libro de referencia para sistemas de tipos. No se lee de una sentada, se consulta capítulo por capítulo.
- **Robert Harper, *Practical Foundations for Programming Languages*** (Cambridge University Press, 2016, 2da ed.). Más moderno que Pierce. Cubre efectos, polimorfismo de filas, cosas que no estaban en el de Pierce.

- **Pierce et al., *Software Foundations*** (volúmenes online en `softwarefoundations.cis.upenn.edu`). Curso interactivo sobre sistemas de tipos verificados en Coq. Para quien quiera el rigor pleno.

F.7 Contratos y diseño por contrato

- **Bertrand Meyer, *Object-Oriented Software Construction*** (Prentice Hall, 1997, 2da ed.). El libro original de Eiffel y de design by contract. Aunque el contexto OO no es el de kaikai, los argumentos sobre por qué los contratos importan son los mismos.
- **John Barnes, *Programming in Ada 2012 with a Preview of Ada 2022*** (Cambridge University Press, 2014). Ada es el otro gran exponente de contratos en un lenguaje de uso industrial. Para entender cómo se ven en producción.

F.8 La comunidad y el código

- **Repositorio oficial**: `github.com/kaikailang-org/kaikai`. El compilador, el stdlib, los documentos de diseño. Issues abiertos para reportes y propuestas.
- **Este libro**: `github.com/kaikailang-org/kaikai-book`. PRs con correcciones son bienvenidos. El libro está en español e inglés; ambas ediciones se mantienen en paralelo.
- **Blog del autor**: `lnds.net`. Donde aparecen ideas antes que en el libro, con menos disciplina y más juicio personal.

F.9 Cierre

Si el libro te dejó con ganas de probar algo, la mejor manera de aprender es escribir código. Toma cualquier programa que ya hayas escrito en otro lenguaje (cualquier lenguaje), y pruébalo a portarlo a kaikai. Vas a tropezar con cosas que el libro no cubrió, vas a abrir issues, vas a aprender lo que ningún libro puede enseñar.

El compilador está vivo, el lenguaje está evolucionando, y la comunidad es pequeña pero atenta. Hay lugar para más.