EBook Gratis

APRENDIZAJE

Elm Language

Free unaffiliated eBook created from
Stack Overflow contributors.

#elm

Tabla de contenido

Acerca de

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: elm-language

It is an unofficial and free Elm Language ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official Elm Language.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

Capítulo 1: Empezando con Elm Language

Observaciones

[Elm] [1] es un lenguaje de programación funcional amigable que compila a JavaScript. Elm se enfoca en GUIs basadas en navegador, aplicaciones de página única.

Los usuarios generalmente lo elogian por:

- No hay excepciones de tiempo de ejecución.
- Los mejores errores del compilador
- La facilidad de refactorización.
- Sistema de tipo expresivo
- The Elm Architecture , de la que se inspira Redux.

Versiones

Versión	Fecha de lanzamiento
0.18.0	2016-11-14
0.17.1	2016-06-27
0.17	2016-05-10
0.16	2015-11-19
0.15.1	2015-06-30
0.15	2015-04-20

Examples

Instalación

Para iniciar el desarrollo con Elm, necesita instalar un conjunto de herramientas llamadas elm-platform .

Incluye: elm-make , elm-reactor , elm-repl y elm-package .

Todas estas herramientas están disponibles a través de CLI, en otras palabras, puede usarlas desde su terminal.

Elija uno de los siguientes métodos para instalar Elm:

Usando el instalador

Descargue el instalador desde el sitio web oficial y siga el asistente de instalación.

Usando npm

Puede usar Node Package Manager para instalar la plataforma Elm.

Instalación global:

```
$ npm install elm -g
```

Instalación local:

```
$ npm install elm
```

Las herramientas de la plataforma Elm instaladas localmente son accesibles a través de:

```
$ ./node_modules/.bin/elm-repl  # launch elm-repl from local node_modules/
```

Usando Homebrew

```
$ brew install elm
```

Cambiar entre versiones con elm-use

Instalar elm-use

```
$ npm install -g elm-use
```

Cambiar a una versión de olmo más antigua o más nueva

```
$ elm-use 0.18  // or whatever version you want to use
```

Otras lecturas

Aprende a inicializar y construir tu primer proyecto.

Hola Mundo

Vea cómo compilar este código en Inicializar y compilar

```
import Html

main = Html.text "Hello World!"
```

Editores

Átomo

- https://atom.io/packages/language-elm
- https://atom.io/packages/elmjutsu

Mesa ligera

- https://github.com/rundis/elm-light

Texto sublime

- https://packagecontrol.io/packages/Elm%20Language%20Support

Empuje

- https://github.com/ElmCast/elm-vim

Emacs

- https://github.com/jcollard/elm-mode

IntelliJ IDEA

- https://plugins.jetbrains.com/plugin/8192

Soportes

- https://github.com/tommot348/elm-brackets

Código VS

- https://marketplace.visualstudio.com/items?itemName=sbrink.elm

Inicializar y construir

Debe tener la plataforma Elm instalada en su computadora; el siguiente tutorial está escrito con el supuesto de que está familiarizado con el terminal.

Inicialización

Crea una carpeta y navega hacia ella con tu terminal:

```
$ mkdir elm-app
$ cd elm-app/
```

Inicialice el proyecto de Elm e instale dependencias principales:

```
$ elm-package install -y
```

`elm-package.json` y `elm-stuff` carpeta `elm-stuff` deberían aparecer en su proyecto.

Cree el punto de entrada para su aplicación `Main.elm` y pegue el ejemplo de Hello World en él.

Construyendo el proyecto

Para construir su primer proyecto, ejecute:

```
$ elm-make Main.elm
```

Esto producirá `index.html` con el archivo `Main.elm` (y todas las dependencias) compilado en JavaScript e insertado en el HTML. **Intenta abrirlo en tu navegador!**

Si esto falla con el error, `I cannot find module 'Html'.` significa que no está utilizando la última versión de Elm. Puede resolver el problema actualizando Elm y rehaciendo el primer paso, o con el siguiente comando:

```
$ elm-package install elm-lang/html -y
```

En caso de que tenga su propio archivo `index.html` (por ejemplo, cuando trabaje con puertos), también puede compilar sus archivos Elm en un archivo JavaScript:

```
$ elm-make Main.elm --output=elm.js
```

Más información en el ejemplo Incrustación en HTML .

Guía de estilo y formato olmo.

La guía de estilo oficial se encuentra en la página de inicio y generalmente incluye:

- legibilidad (en lugar de compacidad)
- facilidad de modificación
- limpiar los dif

Esto significa que, por ejemplo, esto:

```
homeDirectory : String
homeDirectory =
  "/root/files"


evaluate : Boolean -> Bool
evaluate boolean =
  case boolean of
    Literal bool ->
        bool

    Not b ->
        not (evaluate b)

    And b b' ->
        evaluate b && evaluate b'

    Or b b' ->
        evaluate b || evaluate b'
```

Se considera **mejor** que:

```
homeDirectory = "/root/files"

eval boolean = case boolean of
    Literal bool -> bool
    Not b        -> not (eval b)
    And b b'     -> eval b && eval b'
    Or b b'      -> eval b || eval b'
```

0.16

La herramienta elm-format ayuda al formatear su código fuente **automáticamente** (normalmente en guardar), en una línea similar al gofmt de Go language. Una vez más, el valor subyacente es tener **un estilo coherente** y guardar argumentos y flamewars sobre diversos temas como *pestañas y espacios* o *sangría* .

Puede instalar `elm-format` siguiendo las instrucciones en el repositorio de Github . Luego configure su editor para formatear los archivos Elm automáticamente o ejecute `elm-format FILE_OR_DIR --yes` manualmente.

Incrustar en HTML

Hay tres posibilidades para insertar el código Elm en una página HTML existente.

Incrustar en la etiqueta del cuerpo

Suponiendo que haya compilado el ejemplo de Hello World en el archivo `elm.js` , puede dejar que Elm se haga cargo de la etiqueta `<body>` forma:

```
<!DOCTYPE html>
<html>
    <body>
        <script src="elm.js"></script>
        <script>
          Elm.Main.fullscreen()
        </script>
    </body>
</html>
```

ADVERTENCIA : a veces algunas extensiones de chrome se meten con `<body>` que puede provocar que su aplicación se interrumpa en la producción. Se recomienda siempre incrustar en un div específico. Más información aquí .

Incrustar en una Div (u otro nodo DOM)

Alternativamente, al proporcionar un elemento HTML concreto, el código Elm se puede ejecutar en ese elemento de página específico:

```
<!DOCTYPE html>
<html>
    <head>
        <title>Hello World</title>
    </head>
    <body>
        <div id='app'></div>
        <script src="elm.js"></script>
        <script>
            Elm.Main.embed(document.getElementById('app'))
        </script>
    </body>
</html>
```

Incrustar como trabajador web (sin interfaz de usuario)

El código Elm también se puede iniciar como trabajador y comunicarse a través de los puertos :

```
<!DOCTYPE html>
<html>
    <head>
        <title>Hello Worker</title>
    </head>
```

```
    <body>
        <script src="elm.js"></script>
        <script>
            var app = Elm.Main.worker();
            app.ports.fromElmToJS.subscribe(function(world) {
                console.log(world)
            });
            app.ports.fromJSToElm.send('hello');
        </script>
    </body>
</html>
```

REPL

Una buena manera de aprender sobre Elm es intentar escribir algunas expresiones en el REPL (Read-Eval-Print Loop). Abra una consola en su carpeta de `elm-app` (que ha creado en la fase de Inicialización y compilación) y pruebe lo siguiente:

```
$ elm repl
---- elm-repl 0.17.1 -----------------------------------------------------------
 :help for help, :exit to exit, more at <https://github.com/elm-lang/elm-repl>
--------------------------------------------------------------------------------
> 2 + 2
4 : number
> \x -> x
<function> : a -> a
> (\x -> x + x)
<function> : number -> number
> (\x -> x + x) 2
4 : number
>
```

`elm-repl` es en realidad una herramienta bastante poderosa. Supongamos que crea un archivo `Test.elm` dentro de su carpeta de `elm-app` con el siguiente código:

```
module Test exposing (..)


a = 1


b = "Hello"
```

Ahora, vuelve a tu REPL (que se ha mantenido abierto) y escribe:

```
import Test exposing (..)
> a
1 : number
> b
"Hello" : String
>
```

Aún más impresionante, si agrega una nueva definición a su archivo `Test.elm` , como

```
s = """
Hello,
Goodbye.
"""
```

Guarde su archivo, regrese una vez más a su REPL y, sin importar nuevamente la `Test` , la nueva definición estará disponible de inmediato:

```
> s
"\nHello,\nGoodbye.\n" : String
>
```

Es realmente conveniente cuando desea escribir expresiones que abarquen muchas líneas. También es muy útil para probar rápidamente las funciones que acaba de definir. Agregue lo siguiente a su archivo:

```
f x =
  x + x * x
```

Guarda y vuelve al REPL:

```
> f
<function> : number -> number
> f 2
6 : number
> f 4
20 : number
>
```

Cada vez que modifica y guarda un archivo que ha importado, y vuelve al REPL e intenta hacer cualquier cosa, se recompila el archivo completo. Por lo tanto, le informará sobre cualquier error en su código. Agrega esto:

```
c = 2 ++ 2
```

Trata eso:

```
> 0
-- TYPE MISMATCH ---------------------------------------------------- ./Test.elm

The left argument of (++) is causing a type mismatch.

22|     2 ++ 2
        ^
(++) is expecting the left argument to be a:

    appendable

But the left argument is:

    number

Hint: Only strings, text, and lists are appendable.
```

```
>
```

Para concluir esta introducción a la REPL, agreguemos que `elm-repl` también conoce los paquetes que ha instalado con `elm package install` . Por ejemplo:

```
> import Html.App
> Html.App.beginnerProgram
<function>
    : { model : a, update : b -> a -> a, view : a -> Html.Html b }
      -> Platform.Program Basics.Never
>
```

Servidor de compilación local (Elm Reactor)

Elm Reactor es la herramienta esencial para crear prototipos de su aplicación.

Tenga en cuenta que no podrá compilar `Main.elm` con Elm Reactor, si está utilizando Http.App.programWithFlags o Ports

La ejecución de elm-reactor en un directorio de proyectos iniciará un servidor web con un explorador de proyectos, que le permite compilar cada componente por separado.

Cualquier cambio que realice en su código se actualizará cuando vuelva a cargar la página.

```
$ elm-reactor                     # launch elm-reactor on localhost:8000
$ elm-reactor -a=0.0.0.0 -p=3000  # launch elm-reactor on 0.0.0.0:3000
```

Lea Empezando con Elm Language en línea: https://riptutorial.com/es/elm/topic/1011/empezando-con-elm-language

Capítulo 2: Decodificadores JSON personalizados

Introducción

Cómo usar Json.Decode para crear decodificadores personalizados, por ejemplo, decodificación en tipos de unión y tipos de datos definidos por el usuario

Examples

Decodificación en tipo union

```
import Json.Decode as JD
import Json.Decode.Pipeline as JP

type PostType = Image | Video

type alias Post = {
    id: Int
    , postType: PostType
}
-- assuming server will send int value of 0 for Image or 1 for Video
decodePostType: JD.Decoder PostType
decodePostType =
    JD.int |> JD.andThen (\postTypeInt ->
        case postTypeInt of
            0 ->
                JD.succeed Image


            1 ->
                JD.succed Video

            _ ->
                JD.fail "invalid posttype"

    )

decodePostMap : JD.Decoder Post
decodePostMap =
    JD.map2 Post
        (JD.field "id" JD.int)
        (JD.field "postType" decodePostType)

decodePostPipline : JD.Decoder Post
decodePostPipline =
    JP.decode Post
        |> JP.required "id" JD.int
        |> JP.required "postType" decodePostType
```

Lea Decodificadores JSON personalizados en línea:
https://riptutorial.com/es/elm/topic/9927/decodificadores-json-personalizados

Capítulo 3: Depuración

Sintaxis

- Debug.log "tag" anyValue

Observaciones

`Debug.log` toma dos parámetros, una `String` para etiquetar la salida de depuración en la consola (para que sepa de dónde viene / a qué corresponde el mensaje) y un valor de cualquier tipo. `Debug.log` ejecuta el efecto secundario de registrar la etiqueta y el valor en la consola de JavaScript, y luego devuelve el valor. La implementación en JS podría verse algo como:

```
function log (tag, value){
    console.log(tag, value);
    return value
}
```

JavaScript tiene conversiones implícitas, por lo que el `value` no tiene que convertirse explícitamente en una `String` para que el código anterior funcione. Sin embargo, los tipos de Elm deben convertirse explícitamente en una `String` , y el código nativo para `Debug.log` muestra en acción.

Examples

Registrar un valor sin interrumpir los cálculos

El segundo argumento de `Debug.log` siempre se devuelve, por lo que podría escribir código como el siguiente y *simplemente funcionaría* :

```
update : Msg -> Model -> (Model, Cmd Msg)
update msg model =
    case Debug.log "The Message" msg of
        Something ->
            ...
```

Sustitución `case msg of` con el `case Debug.log "The Message" msg of` hará que el mensaje actual que estar conectado a la consola cada vez que la función de actualización se llama, pero no cambia nada más.

Canalizando un Debug.log

En el tiempo de ejecución, lo siguiente mostraría una lista de url en su consola y continuaría el cálculo

```
payload =
    [{url:..., title:...}, {url=..., title=...}]

main =
    payload
        |> List.map .url -- only takes the url
        |> Debug.log " My list of URLs" -- pass the url list to Debug.log and return it
        |> doSomething -- do something with the url list
```

Depurador que viaja en el tiempo

0.17 0.18.0

Al momento de escribir (julio de 2016), elm-reactor ha sido despojado temporalmente de su funcionalidad de viaje en el tiempo. Sin embargo, es posible obtenerlo utilizando el `jinjor/elm-time-travel` .

Su uso refleja las `Html.App` `program*` módulos `Html.App` o `Navigation` , por ejemplo, en lugar de:

```
import Html.App

main =
    Html.App.program
        { init = init
        , update = update
        , view = view
        , subscriptions = subscriptions
        }
```

usted escribiría:

```
import TimeTravel.Html.App

main =
    TimeTravel.Html.App.program
        { init = init
        , update = update
        , view = view
        , subscriptions = subscriptions
        }
```

(Por supuesto, después de instalar el paquete con `elm-package` .)

La interfaz de su aplicación cambia como resultado, vea una de las demostraciones .

0.18.0

Desde la versión **0.18.0** , simplemente puede compilar su programa con la `--debug` y obtener la depuración del viaje en el tiempo sin ningún esfuerzo adicional.

Debug.crash

```
case thing of
```

```
    Cat ->
        meow
    Bike ->
        ride
    Sandwich ->
        eat
    _ ->
        Debug.crash "Not yet implemented"
```

Puedes usar `Debug.crash` cuando quieres que el programa falle, normalmente se usa cuando estás implementando una expresión de `case` . *No* se recomienda el uso `Debug.crash` lugar de utilizar un `Maybe` o `Result` tipo para entradas inesperadas, pero normalmente sólo durante el curso del desarrollo (es decir, que por lo general no publicará código que utiliza Elm `Debug.crash`).

`Debug.crash` toma un valor de `String` , el mensaje de error que se muestra cuando falla. Tenga en cuenta que Elm también mostrará el nombre del módulo y la línea del bloqueo, y si el bloqueo se produce en una expresión de `case` , indicará el valor del `case` .

Lea Depuración en línea: https://riptutorial.com/es/elm/topic/2845/depuracion

Capítulo 4: Funciones y aplicación parcial.

Sintaxis

- - Definir una función sin argumentos parece lo mismo que simplemente definir un valor
  language = "Elm"
- - llamar a una función sin argumentos indicando su nombre
  idioma
- - Los parámetros están separados por espacios y siguen el nombre de la función.
  agrega xy = x + y
- - llamar a una función de la misma manera
  añadir 5 2
- - aplicar parcialmente una función proporcionando solo algunos de sus parámetros
  incremento = agregar 1
- - use el operador |> para pasar la expresión de la izquierda a la función de la derecha
  ten = 9 |> incremento
- - el <| el operador pasa la expresión de la derecha a la función de la izquierda
  incremento <| añadir 5 4
- - encadenar / componer dos funciones junto con el >> operador
  backwardsYell = String.reverse >> String.toUpper
- - El << funciona igual en sentido inverso.
  backwardsYell = String.toUpper << String.reverse
- - una función con un nombre no alfanumérico entre paréntesis crea un nuevo operador
  (#) xy = x * y
  diez = 5 # 2
- - cualquier operador de infijo se convierte en una función normal cuando lo envuelve entre paréntesis
  diez = (+) 5 5
- - Las anotaciones de tipo opcionales aparecen arriba de las declaraciones de funciones.
  isTen: Int -> Bool
  isTen n = if n == 10 entonces True más False

Examples

Visión general

La sintaxis de la aplicación de la función en Elm no usa paréntesis ni comas, y en su lugar es sensible al espacio en blanco.

Para definir una función, especifique su nombre `multiplyByTwo` y argumentos `x` , cualquier operación después del signo igual `=` es lo que se devuelve de una función.

```
multiplyByTwo x =
    x * 2
```

Para llamar a una función, especifique su nombre y argumentos:

```
multiplyByTwo 2  -- 4
```

Tenga en cuenta que la sintaxis como `multiplyByTwo(2)` no es necesaria (aunque el compilador no se queja). Los paréntesis solo sirven para resolver la precedencia:

```
> multiplyByTwo multiplyByTwo 2
-- error, thinks it's getting two arguments, but it only needs one

> multiplyByTwo (multiplyByTwo 2)
4 : number

> multiplyByTwo 2 + 2
6 : number
-- same as (multiplyByTwo 2) + 2

> multiplyByTwo (2 + 2)
8 : number
```

Expresiones lambda

Elm tiene una sintaxis especial para expresiones lambda o funciones anónimas:

```
\arguments -> returnedValue
```

Por ejemplo, como se ve en `List.filter` :

```
> List.filter (\num -> num > 1) [1,2,3]
[2,3] : List number
```

Más a la profundidad, se usa una barra invertida, `\` , para marcar el comienzo de la expresión lambda, y la flecha, `->` , se usa para delimitar argumentos del cuerpo de la función. Si hay más argumentos, se separan por un espacio:

```
normalFunction x y = x + y
-- is equivalent to
lambdaFunction = \x y -> x + y

> normalFunction 1 2
3 : number

> lambdaFunction 1 2
3 : number
```

Variables locales

Es posible definir variables locales dentro de una función para

- reducir la repetición de código
- dar nombre a subexpresiones

- Reducir la cantidad de argumentos pasados.

El constructo para esto se `let ... in ...`

```
bigNumbers =
    let
        allNumbers =
            [1..100]

        isBig number =
            number > 95
    in
        List.filter isBig allNumbers

> bigNumbers
[96,97,98,99,100] : List number

> allNumbers
-- error, doesn't know what allNumbers is!
```

El orden de las definiciones en la primera parte de `let` no importa!

```
outOfOrder =
    let
        x =
            y + 1  -- the compiler can handle this

        y =
            100
    in
        x + y

> outOfOrder
201 : number
```

Solicitud parcial

La aplicación parcial significa llamar a una función con menos argumentos de los que tiene y guardar el resultado como otra función (que espera el resto de los argumentos).

```
multiplyBy: Int -> Int -> Int
multiplyBy x y =
    x * y


multiplyByTwo : Int -> Int  -- one Int has disappeared! we now know what x is.
multiplyByTwo =
    multiplyBy 2


> multiplyByTwo 2
4 : Int

> multiplyByTwo 4
8 : Int
```

Como una nota de orientación académica, Elm puede hacer esto debido al curry detrás de escena.

Evaluación estricta y demorada.

En elm, el valor de una función se calcula cuando se aplica el último argumento. En el siguiente ejemplo, el diagnóstico desde el `log` se imprimirá cuando f se invoque con 3 argumentos o se aplique una forma currada de f con el último argumento.

```
import String
import Debug exposing (log)

f a b c = String.join "," (log "Diagnostic" [a,b,c]) -- <function> : String -> String ->
String -> String

f2 = f "a1" "b2" -- <function> : String -> String

f "A" "B" "C"
-- Diagnostic: ["A","B","C"]
"A,B,C" : String

f2 "c3"
-- Diagnostic: ["a1","b2","c3"]
"a1,b2,c3" : String
```

A veces querrá evitar que una función se aplique de inmediato. Un uso típico en elm es `Lazy.lazy` que proporciona una abstracción para controlar cuándo se aplican las funciones.

```
lazy : (() -> a) -> Lazy a
```

Los cálculos perezosos toman una función de uno `()` o argumento de tipo de `Unit` . El tipo de unidad es convencionalmente el tipo de un argumento de marcador de posición. En una lista de argumentos, el argumento correspondiente se especifica como _ , lo que indica que el valor no se usa. El valor de la unidad en olmo se especifica mediante el símbolo especial `()` que puede representar conceptualmente una tupla vacía o un agujero. Se parece a la lista de argumentos vacía en C, Javascript y otros idiomas que usan paréntesis para llamadas a funciones, pero es un valor ordinario.

En nuestro ejemplo, `f` puede ser protegido para que no se evalúe inmediatamente con un lambda:

```
doit f = f () -- <function> : (() -> a) -> a
whatToDo = \_ -> f "a" "b" "c" -- <function> : a -> String
-- f is not evaluated yet

doit whatToDo
-- Diagnostic: ["a","b","c"]
"a,b,c" : String
```

La evaluación de la función se retrasa cada vez que una función se aplica parcialmente.

```
defer a f = \_ -> f a -- <function> : a -> (a -> b) -> c -> b
```

```
delayF = f "a" "b" |> defer "c" -- <function> : a -> String

doit delayF
-- Diagnostic: ["a","b","c"]
"a,b,c" : String
```

Elm tiene una función de `always` , que no puede usarse para retrasar la evaluación. Debido a que elm evalúa todos los argumentos de la función independientemente de si se usa el resultado de la aplicación de función y cuándo, el ajuste de una aplicación de función no `always` causará un retraso, porque `f` se aplica completamente como parámetro para `always` .

```
alwaysF = always (f "a" "b" "c") -- <function> : a -> String
-- Diagnostic: ["a","b","c"] -- Evaluation wasn't delayed.
```

Operadores de infijo y notación de infijo.

Elm permite la definición de operadores de infijo personalizados.

Los operadores de infijo se definen utilizando paréntesis alrededor del nombre de una función.

Considere este ejemplo de operador infijo para construcciones Tuples `1 => True -- (1, True)` :

```
(=>) : a -> b -> ( a, b )
(=>) a b =
    ( a, b )
```

La mayoría de las funciones en Elm se definen en la notación de prefijo.

Aplique cualquier función utilizando la notación de infijo especificando el primer argumento antes del nombre de la función incluido con el carácter de acento grave:

```
import List exposing (append)


append [1,1,2] [3,5,8]   -- [1,1,2,3,5,8]
[1,1,2] `append` [3,5,8] -- [1,1,2,3,5,8]
```

Lea Funciones y aplicación parcial. en línea: https://riptutorial.com/es/elm/topic/2051/funciones-y-aplicacion-parcial-

Capítulo 5: Haciendo funciones de actualización complejas con ccapndave / elm-update-extra

Introducción

ccapndave / elm-update-extra es un paquete fantástico que le ayuda a manejar funciones de actualización más complejas y puede ser muy útil.

Examples

Mensaje que llama a una lista de mensajes.

Usando la función de `sequence` , puede describir fácilmente un mensaje que llame a una lista de otros mensajes. Es útil cuando se trata de la semántica de sus mensajes.

Ejemplo 1: Está creando un motor de juego y necesita actualizar la pantalla en cada fotograma.

```
module Video exposing (..)
type Message = module Video exposing (..)

import Update.Extra exposing (sequence)

-- Model definition [...]

type Message
    = ClearBuffer
    | DrawToBuffer
    | UpdateLogic
    | Update

update : Message -> Model -> (Model, Cmd)
update msg model =
    case msg of
        ClearBuffer ->
            -- do something
        DrawToBuffer ->
            -- do something
        UpdateLogic ->
            -- do something
        Update ->
            model ! []
                |> sequence update [ ClearBuffer
                                   , DrawToBuffer
                                   , UpdateLogic]
```

Encadenando mensajes con y luego

La función `andThen` permite actualizar la composición de la llamada. Se puede utilizar con el

operador de tubería (`|>`) para encadenar actualizaciones.

Ejemplo: está creando un editor de documentos y desea que cada mensaje de modificación que envíe a su documento también lo guarde:

```
import Update.Extra exposing (andThen)
import Update.Extra.Infix exposing (..)

-- type alias Model = [...]

type Message
    = ModifyDocumentWithSomeSettings
    | ModifyDocumentWithOtherSettings
    | SaveDocument

update : Model -> Message -> (Model, Cmd)
update model msg =
    case msg of
        ModifyDocumentWithSomeSettings ->
            -- make the modifications
            (modifiedModel, Cmd.none)
            |> andThen SaveDocument
        ModifyDocumentWithOtherSettings ->
            -- make other modifications
            (modifiedModel, Cmd.none)
            |> andThen SaveDocument
        SaveDocument ->
            -- save document code
```

Si también importa `Update.Extra.Infix exposing (..)` puede usar el operador infijo:

```
update : Model -> Message -> (Model, Cmd)
update model msg =
    case msg of
        ModifyDocumentWithSomeSettings ->
            -- make the modifications
            (modifiedModel, Cmd.none)
            :> andThen SaveDocument
        ModifyDocumentWithOtherSettings ->
            -- make other modifications
            (modifiedModel, Cmd.none)
            :> SaveDocument
        SaveDocument ->
            -- save document code
```

Lea Haciendo funciones de actualización complejas con ccapndave / elm-update-extra en línea: https://riptutorial.com/es/elm/topic/9737/haciendo-funciones-de-actualizacion-complejas-con-ccapndave---elm-update-extra

Capítulo 6: Integración de backend

Examples

Solicitud básica de elm Http.post json al servidor express de node.js

El servidor de mayúsculas activo que devuelve un error cuando la cadena de entrada tiene más de 10 caracteres.

Servidor:

```
const express = require('express'),
    jsonParser = require('body-parser').json(),
    app = express();

// Add headers to work with elm-reactor
app.use((req, res, next) => {
    res.setHeader('Access-Control-Allow-Origin', 'http://localhost:8000');
    res.setHeader('Access-Control-Allow-Methods', 'POST, OPTIONS');
    res.setHeader('Access-Control-Allow-Headers', 'X-Requested-With,content-type');
    res.setHeader('Access-Control-Allow-Credentials', true);
    next();
});

app.post('/upcase', jsonParser, (req, res, next) => {
    // Just an example of possible invalid data for an error message demo
    if (req.body.input && req.body.input.length < 10) {
        res.json({
            output: req.body.input.toUpperCase()
        });
    } else {
        res.status(500).json({
            error: `Bad input: '${req.body.input}'`
        });
    }
});

const server = app.listen(4000, () => {
    console.log('Server is listening at http://localhost:4000/upcase');
});
```

Cliente:

```
import Html exposing (..)
import Html.Attributes exposing (..)
import Html.Events exposing (..)
import Http
import Json.Decode as JD
import Json.Encode as JE

main : Program Never Model Msg
main =
    Html.program
        { init = init
```

```
        , view = view
        , update = update
        , subscriptions = subscriptions
        }

-- MODEL

type alias Model =
    { output: String
    , error: Maybe String
    }

init : (Model, Cmd Msg)
init =
    ( Model "" Nothing
    , Cmd.none
    )

-- UPDATE

type Msg
    = UpcaseRequest ( Result Http.Error String )
    | InputString String

update : Msg -> Model -> (Model, Cmd Msg)
update msg model =
    case msg of
        UpcaseRequest (Ok response) ->
            ( { model | output = response, error = Nothing }, Cmd.none )

        UpcaseRequest (Err err) ->
            let
                errMsg = case err of
                    Http.Timeout ->
                        "Request timeout"

                    Http.NetworkError ->
                        "Network error"

                    Http.BadPayload msg _ ->
                        msg

                    Http.BadStatus response ->
                        case JD.decodeString upcaseErrorDecoder response.body of
                            Ok errStr ->
                                errStr

                            Err _ ->
                                response.status.message

                    Http.BadUrl msg ->
                        "Bad url: " ++ msg
            in
                ( { model | output = "", error = Just errMsg }, Cmd.none )

        InputString str ->
            ( model, upcaseRequest str )

-- VIEW

view : Model -> Html Msg
```

```
view model =
    let
        outDiv = case model.error of
            Nothing ->
                div []
                    [ label [ for "outputUpcase" ] [ text "Output" ]
                    , input [ type_ "text", id "outputUpcase", readonly True, value
model.output ] []
                    ]

            Just err ->
                div []
                    [ label [ for "errorUpcase" ] [ text "Error" ]
                    , input [ type_ "text", id "errorUpcase", readonly True, value err ] []
                    ]
    in
        div []
            [ div []
                [ label [ for "inputToUpcase" ] [ text "Input" ]
                , input [ type_ "text", id "inputToUpcase", onInput InputString ] []
                ]
            , outDiv
            ]

-- SUBSCRIPTIONS

subscriptions : Model -> Sub Msg
subscriptions model =
    Sub.none

-- HELPERS

upcaseSuccessDecoder : JD.Decoder String
upcaseSuccessDecoder = JD.field "output" JD.string

upcaseErrorDecoder : JD.Decoder String
upcaseErrorDecoder = JD.field "error" JD.string

upcaseRequestEncoder : String -> JE.Value
upcaseRequestEncoder str = JE.object [ ( "input", JE.string str ) ]

upcaseRequest : String -> Cmd Msg
upcaseRequest str =
    let
        req = Http.post "http://localhost:4000/upcase" ( Http.jsonBody <| upcaseRequestEncoder
str ) upcaseSuccessDecoder
    in
        Http.send UpcaseRequest req
```

Lea Integración de backend en línea: https://riptutorial.com/es/elm/topic/8087/integracion-de-backend

Capítulo 7: Json.Decodificar

Observaciones

`Json.Decode` expone dos funciones para decodificar una carga útil, la primera es `decodeValue` que intenta descodificar un `Json.Encode.Value` , la segunda es `decodeString` que intenta descodificar una cadena JSON. Ambas funciones toman 2 parámetros, un decodificador y una cadena `Json.Encode.Value` o Json.

Examples

Decodificando una lista

El siguiente ejemplo se puede probar en https://ellie-app.com/m9tk39VpQg/0 .

```
import Html exposing (..)
import Json.Decode

payload =
  """
  ["fu", "bar"]
  """

main =
  Json.Decode.decodeString decoder payload -- Ok ["fu","bar"]
  |> toString
  |> text

decoder =
  Json.Decode.list Json.Decode.string
```

Pre-decodifique un campo y decodifique el resto dependiendo de ese valor decodificado

Los siguientes ejemplos se pueden probar en https://ellie-app.com/m9vmQ8NcMc/0 .

```
import Html exposing (..)
import Json.Decode

payload =
  """
  [ { "bark": true, "tag": "dog", "name": "Zap", "playful": true }
  , { "whiskers": true, "tag" : "cat", "name": "Felix" }
  , {"color": "red", "tag": "tomato"}
  ]
  """

-- OUR MODELS

type alias Dog =
  { bark: Bool
```

```
  , name: String
  , playful: Bool
  }

type alias Cat =
  { whiskers: Bool
  , name: String
  }

-- OUR DIFFERENT ANIMALS

type Animal
  = DogAnimal Dog
  | CatAnimal Cat
  | NoAnimal

main =
  Json.Decode.decodeString decoder payload
  |> toString
  |> text

decoder =
  Json.Decode.field "tag" Json.Decode.string
    |> Json.Decode.andThen animalType
    |> Json.Decode.list

animalType tag =
  case tag of
    "dog" ->
      Json.Decode.map3 Dog
          (Json.Decode.field "bark" Json.Decode.bool)
          (Json.Decode.field "name" Json.Decode.string)
          (Json.Decode.field "playful" Json.Decode.bool)
        |> Json.Decode.map DogAnimal
    "cat" ->
      Json.Decode.map2 Cat
          (Json.Decode.field "whiskers" Json.Decode.bool)
          (Json.Decode.field "name" Json.Decode.string)
        |> Json.Decode.map CatAnimal
    _ ->
      Json.Decode.succeed NoAnimal
```

Decodificación JSON de Rust enum

Esto es útil si usa óxido en la parte posterior y olmo en la parte delantera

```
enum Complex{
    Message(String),
    Size(u64)
}

let c1 = Complex::Message("hi");
let c2 = Complex::Size(1024u64);
```

El Json codificado de la herrumbre será:

```
c1:
    {"variant": "Message",
```

```
     "fields": ["hi"]
    }
c2:
    {"variant": "Size",
     "fields": [1024]
    }
```

El decodificador en olmo.

```
import Json.Decode as Decode exposing (Decoder)

type Complex = Message String
    | Size Int

-- decodes json to Complex type
complexDecoder: Decoder Value
complexDecoder =
    ("variant" := Decode.string `Decode.andThen` variantDecoder)

variantDecoder: String -> Decoder Value
variantDecoder variant =
    case variant of
        "Message" ->
            Decode.map Message
                ("fields" := Decode.tuple1 (\a -> a) Decode.string)
        "Size" ->
            Decode.map Size
                ("fields" := Decode.tuple1 (\a -> a) Decode.int)
        _ ->
            Debug.crash "This can't happen"
```

Uso: los datos se solicitan desde http api api y la decodificación de la carga útil será

```
    Http.fromJson complexDecoder payload
```

La decodificación de la cadena será

```
    Decode.decodeString complexDecoder payload
```

Decodificando una lista de registros

El siguiente código se puede encontrar en una demostración aquí: https://ellie-app.com/mbFwJT9jD3/0

```
import Html exposing (..)
import Json.Decode exposing (Decoder)

payload =
  """
  [{
      "id": 0,
      "name": "Adam Carter",
      "work": "Unilogic",
      "email": "adam.carter@unilogic.com",
      "dob": "24/11/1978",
```

```
      "address": "83 Warner Street",
      "city": "Boston",
      "optedin": true
    },
    {
      "id": 1,
      "name": "Leanne Brier",
      "work": "Connic",
      "email": "leanne.brier@connic.org",
      "dob": "13/05/1987",
      "address": "9 Coleman Avenue",
      "city": "Toronto",
      "optedin": false
    }]
  """

type alias User =
  { name: String
  , work: String
  , email: String
  , dob: String
  , address: String
  , city: String
  , optedin: Bool
  }

main =
  Json.Decode.decodeString decoder payload
  |> toString
  |> text

decoder: Decoder (List User)
decoder =
    Json.Decode.map7 User
    (Json.Decode.field "name" Json.Decode.string)
    (Json.Decode.field "work" Json.Decode.string)
    (Json.Decode.field "email" Json.Decode.string)
    (Json.Decode.field "dob" Json.Decode.string)
    (Json.Decode.field "address" Json.Decode.string)
    (Json.Decode.field "city" Json.Decode.string)
    (Json.Decode.field "optedin" Json.Decode.bool)
    |> Json.Decode.list
```

Decodificar una fecha

En caso de que tenga json con una cadena de fecha ISO como esta

```
JSON.stringify({date: new Date()})
// -> "{"date":"2016-12-12T13:24:34.470Z"}"
```

Puedes mapearlo en elm Tipo de `Date` :

```
import Html exposing (text)
import Json.Decode as JD
import Date

payload = """{"date":"2016-12-12T13:24:34.470Z"}"""
```

```
dateDecoder : JD.Decoder Date.Date
dateDecoder =
  JD.string
    |> JD.andThen ( \str ->
          case Date.fromString str of
            Err err -> JD.fail err
            Ok date -> JD.succeed date )

payloadDecoder : JD.Decoder Date.Date
payloadDecoder =
  JD.field "date" dateDecoder

main =
  JD.decodeString payloadDecoder payload
  |> toString
  |> text
```

Decodificar una lista de objetos que contienen listas de objetos

Ver Ellie para un ejemplo de trabajo. Este ejemplo utiliza el módulo NoRedInk / elm-decode-pipeline .

Dada una lista de objetos JSON, que a su vez contienen listas de objetos JSON:

```
[
  {
    "id": 0,
    "name": "Item 1",
    "transactions": [
      { "id": 0, "amount": 75.00 },
      { "id": 1, "amount": 25.00 }
    ]
  },
  {
    "id": 1,
    "name": "Item 2",
    "transactions": [
      { "id": 0, "amount": 50.00 },
      { "id": 1, "amount": 15.00 }
    ]
  }
]
```

Si la cadena anterior está en la cadena de `payload` , puede decodificarse utilizando lo siguiente:

```
module Main exposing (main)

import Html exposing (..)
import Json.Decode as Decode exposing (Decoder)
import Json.Decode.Pipeline as JP
import String


type alias Item =
    { id : Int
    , name : String
    , transactions : List Transaction
```

```
    }


type alias Transaction =
    { id : Int
    , amount : Float
    }


main =
    Decode.decodeString (Decode.list itemDecoder) payload
        |> toString
        |> String.append "JSON "
        |> text



itemDecoder : Decoder Item
itemDecoder =
    JP.decode Item
        |> JP.required "id" Decode.int
        |> JP.required "name" Decode.string
        |> JP.required "transactions" (Decode.list transactionDecoder)


transactionDecoder : Decoder Transaction
transactionDecoder =
    JP.decode Transaction
        |> JP.required "id" Decode.int
        |> JP.required "amount" Decode.float
```

Lea Json.Decodificar en línea: https://riptutorial.com/es/elm/topic/2849/json-decodificar

Capítulo 8: La arquitectura del olmo

Introducción

La forma recomendada de estructurar sus aplicaciones se denomina 'The Elm Architecture'.

El programa más simple consiste en un registro de `model` almacena todos los datos que pueden actualizarse, un `Msg` tipo de unión que define las formas en que su programa actualiza esos datos, una `update` función que toma el modelo y un `Msg` y devuelve un nuevo modelo, y una `view` función que toma un modelo y devuelve el HTML que mostrará su página. Cada vez que una función devuelve un `Msg` , el tiempo de ejecución de Elm lo utiliza para actualizar la página.

Examples

Programa de principiante

Html tiene un programa para `beginnerProgram` principalmente para fines de aprendizaje.

`beginnerProgram` no es capaz de manejar suscripciones o ejecutar comandos.

Solo es capaz de manejar la entrada del usuario de los eventos DOM.

Solo requiere una `view` para representar el `model` y una función de `update` para manejar los cambios de estado.

Ejemplo

Considere este ejemplo mínimo de `beginnerProgram` .

El `model` en este ejemplo consiste en un solo valor `Int` .

La función de `update` tiene solo una rama, que incrementa el `Int` , almacenado en el `model` .

La `view` representa el modelo y adjunta el evento DOM.

Vea cómo construir el ejemplo en Inicializar y compilar

```
import Html exposing (Html, button, text)
import Html exposing (beginnerProgram)
import Html.Events exposing (onClick)


main : Program Never
main =
    beginnerProgram { model = 0, view = view, update = update }
```

```
-- UPDATE


type Msg
    = Increment

update : Msg -> Int -> Int
update msg model =
    case msg of
        Increment ->
            model + 1


-- VIEW


view : Int -> Html Msg
view model =
    button [ onClick Increment ] [ text ("Increment: " ++ (toString model)) ]
```

Programa

`program` es una buena opción cuando su aplicación no requiere datos externos para la inicialización.

Es capaz de manejar suscripciones y comandos, lo que permite muchas más oportunidades para el manejo de E / S, como la comunicación HTTP o la interoperabilidad con JavaScript.

Se requiere que el estado inicial devuelva los Comandos de inicio junto con el Modelo.

La inicialización del `program` requerirá que se proporcionen `subscriptions` , junto con el `model` , la `view` y la `update` .

Ver la definición de tipo:

```
program :
    { init : ( model, Cmd msg )
    , update : msg -> model -> ( model, Cmd msg )
    , subscriptions : model -> Sub msg
    , view : model -> Html msg
    }
    -> Program Never
```

Ejemplo

La forma más sencilla de ilustrar cómo puede usar las Suscripciones es configurar una comunicación Port simple con JavaScript.

Vea cómo compilar el ejemplo en Inicializar y compilar / incrustar en HTML

```
port module Main exposing (..)
```

```
import Html exposing (Html, text)
import Html exposing (program)


main : Program Never
main =
    program
        { init = init
        , view = view
        , update = update
        , subscriptions = subscriptions
        }


port input : (Int -> msg) -> Sub msg


-- MODEL


type alias Model =
    Int


init : ( Model, Cmd msg )
init =
    ( 0, Cmd.none )


-- UPDATE


type Msg = Incoming Int


update : Msg -> Model -> ( Model, Cmd msg )
update msg model =
    case msg of
        Incoming x ->
          ( x, Cmd.none )


-- SUBSCRIPTIONS


subscriptions : Model -> Sub Msg
subscriptions model =
    input Incoming


-- VIEW


view : Model -> Html msg
view model =
    text (toString model)
```

```
<!DOCTYPE html>
<html>
    <head>
        <script src='elm.js'></script>
```

```
</head>
    <body>
    <div id='app'></div>
    <script>var app = Elm.Main.embed(document.getElementById('app'));</script>
    <button onclick='app.ports.input.send(1);'>send</button>
</body>
</html>
```

Programa con banderas

`programWithFlags` tiene una sola diferencia del `program` .

Puede aceptar los datos en la inicialización de JavaScript:

```
var root = document.body;
var user = { id: 1, name: "Bob" };
var app = Elm.Main.embed( root, user );
```

Los datos, pasados de JavaScript se llaman Flags.

En este ejemplo, estamos pasando un objeto de JavaScript a Elm con información del usuario, es una buena práctica especificar un alias de tipo para las marcas.

```
type alias Flags =
    { id: Int
    , name: String
    }
```

Las banderas se pasan a la función `init` , produciendo el estado inicial:

```
init : Flags -> ( Model, Cmd Msg )
init flags =
    let
        { id, name } =
            flags
    in
        ( Model id name, Cmd.none )
```

Es posible que note la diferencia con su tipo de firma:

```
programWithFlags :
    { init : flags -> ( model, Cmd msg )          -- init now accepts flags
    , update : msg -> model -> ( model, Cmd msg )
    , subscriptions : model -> Sub msg
    , view : model -> Html msg
    }
    -> Program flags
```

El código de inicialización es casi el mismo, ya que solo la función `init` es diferente.

```
main =
    programWithFlags
        { init = init
```

```
        , update = update
        , view = view
        , subscriptions = subscriptions
        }
```

Comunicación unidireccional entre padres e hijos.

El ejemplo muestra la composición de componentes y el mensaje de una sola vía pasando de padres a hijos.

0.18.0

La composición de componentes se basa en el etiquetado de mensajes con `Html.App.map`

0.18.0

En `0.18.0` `HTML.App` fue contraído en `HTML`

La composición de componentes se basa en el etiquetado de mensajes con `Html.map`

Ejemplo

Vea cómo construir el ejemplo en Inicializar y construir

```
module Main exposing (..)

import Html exposing (text, div, button, Html)
import Html.Events exposing (onClick)
import Html.App exposing (beginnerProgram)


main =
    beginnerProgram
        { view = view
        , model = init
        , update = update
        }

{- In v0.18.0 HTML.App was collapsed into HTML
   Use Html.map instead of Html.App.map
-}
view : Model -> Html Msg
view model =
    div []
        [ Html.App.map FirstCounterMsg (counterView model.firstCounter)
        , Html.App.map SecondCounterMsg (counterView model.secondCounter)
        , button [ onClick ResetAll ] [ text "Reset counters" ]
        ]


type alias Model =
    { firstCounter : CounterModel
    , secondCounter : CounterModel
    }
```

```
init : Model
init =
    { firstCounter = 0
    , secondCounter = 0
    }


type Msg
    = FirstCounterMsg CounterMsg
    | SecondCounterMsg CounterMsg
    | ResetAll


update : Msg -> Model -> Model
update msg model =
    case msg of
        FirstCounterMsg childMsg ->
            { model | firstCounter = counterUpdate childMsg model.firstCounter }

        SecondCounterMsg childMsg ->
            { model | secondCounter = counterUpdate childMsg model.secondCounter }

        ResetAll ->
            { model
                | firstCounter = counterUpdate Reset model.firstCounter
                , secondCounter = counterUpdate Reset model.secondCounter
            }


type alias CounterModel =
    Int


counterView : CounterModel -> Html CounterMsg
counterView model =
    div []
        [ button [ onClick Decrement ] [ text "-" ]
        , text (toString model)
        , button [ onClick Increment ] [ text "+" ]
        ]


type CounterMsg
    = Increment
    | Decrement
    | Reset


counterUpdate : CounterMsg -> CounterModel -> CounterModel
counterUpdate msg model =
    case msg of
        Increment ->
            model + 1

        Decrement ->
            model - 1

        Reset ->
            0
```

Etiquetado de mensajes con Html.App.map

Los componentes definen sus propios mensajes, enviados después de los eventos DOM emitidos, por ejemplo. `CounterMsg` de comunicación padre-hijo

```
type CounterMsg
    = Increment
    | Decrement
    | Reset
```

La vista de este componente enviará mensajes de tipo `CounterMsg` , por lo tanto, la firma del tipo de vista es `Html CounterMsg` .

Para poder reutilizar `counterView` vista interior del componente principal, tenemos que pasar todos los `CounterMsg` mensaje a través de los padres `Msg` .

Esta técnica se llama ***etiquetado de mensajes*** .

El componente principal debe definir mensajes para pasar mensajes secundarios:

```
type Msg
    = FirstCounterMsg CounterMsg
    | SecondCounterMsg CounterMsg
    | ResetAll
```

`FirstCounterMsg Increment` es un mensaje etiquetado.

0.18.0

Para obtener un `counterView` para enviar mensajes etiquetados, debemos usar la función `Html.App.map` :

```
Html.map FirstCounterMsg (counterView model.firstCounter)
```

0.18.0

El paquete `Html.App` se `Html.App` en el paquete `Html` en `v0.18.0`

Para obtener un `counterView` para enviar mensajes etiquetados, debemos usar la función `Html.map` :

```
Html.map FirstCounterMsg (counterView model.firstCounter)
```

Eso cambia el tipo de firma `Html CounterMsg -> Html Msg` por lo que es posible usar el contador dentro de la vista principal y manejar las actualizaciones de estado con la función de actualización de los padres.

Lea La arquitectura del olmo en línea: https://riptutorial.com/es/elm/topic/3771/la-arquitectura-del-olmo

Capítulo 9: La coincidencia de patrones

Examples

Argumentos de función

```
type Dog = Dog String

dogName1 dog =
    case dog of
      Dog name ->
         name

dogName2 (Dog name) ->
     name
```

`dogName1` y `dogName2` son equivalentes. Tenga en cuenta que esto solo funciona para los ADT que tienen un único constructor.

```
type alias Pet =
   { name: String
   , weight: Float

   }

render : Pet -> String
render ({name, weight} as pet) =
    (findPetEmoji pet) ++ " " ++ name ++ " weighs " ++ (toString weight)

findPetEmoji : Pet -> String
findPetEmoji pet =
    Debug.crash "Implementation TBD"
```

Aquí deconstruimos un registro y también obtenemos una referencia al registro no construido.

Argumento deconstruido de tipo único

```
type ProjectIdType = ProjectId String

getProject : ProjectIdType -> Cmd Msg
getProject (ProjectId id) =
    Http.get <| "/projects/" ++ id
```

Lea La coincidencia de patrones en línea: https://riptutorial.com/es/elm/topic/7168/la-coincidencia-de-patrones

Capítulo 10: Listas e iteración

Observaciones

La `List` (lista enlazada) brilla en el **acceso secuencial** :

- accediendo al primer elemento
- antepuesto al frente de la lista
- borrar de la parte frontal de la lista

Por otro lado, no es ideal para **el acceso aleatorio** (es decir, obtener el elemento nth) y el **desplazamiento en orden inverso** , y es posible que tenga más suerte (y rendimiento) con la estructura de datos de `Array` .

Examples

Creando una lista por rango

0.18.0

Antes de **0.18.0** puedes crear rangos como este:

```
> range = [1..5]
[1,2,3,4,5] : List number
>
> negative = [-5..3]
[-5,-4,-3,-2,-1,0,1,2,3] : List number
```

0.18.0

En **0.18.0** La sintaxis `[1..5]` ha sido eliminada .

```
> range = List.range 1 5
[1,2,3,4,5] : List number
>
> negative = List.range -5 3
[-5,-4,-3,-2,-1,0,1,2,3] : List number
```

Los rangos creados por esta sintaxis son siempre **inclusivos** y el **paso** es siempre **1** .

Creando una lista

```
> listOfNumbers = [1,4,99]
[1,4,99] : List number
>
> listOfStrings = ["Hello","World"]
["Hello","World"] : List String
>
```

```
> emptyList = []   -- can be anything, we don't know yet
[] : List a
>
```

Bajo el capó, la `List` (lista enlazada) se construye mediante la función `::` (llamada "contras"), que toma dos argumentos: un elemento, conocido como la cabecera, y una lista (posiblemente vacía) a la cual se añade la cabecera.

```
> withoutSyntaxSugar = 1 :: []
[1] : List number
>
> longerOne = 1 :: 2 :: 3 :: []
[1,2,3] : List number
>
> nonemptyTail = 1 :: [2]
[1,2] : List number
>
```

`List` solo puede tomar valores de un tipo, por lo que algo como `[1,"abc"]` no es posible. Si necesitas esto, usa tuplas.

```
> notAllowed = [1,"abc"]
-------------------------------------- ERRORS --------------------------------------

-- TYPE MISMATCH ------------------------------------------------ repl-temp-000.elm

The 1st and 2nd elements are different types of values.

8|              [1,"abc"]
                ^^^^^
The 1st element has this type:

    number

But the 2nd is:

    String

Hint: All elements should be the same type of value so that we can iterate
through the list without running into unexpected values.


>
```

Obteniendo elementos

```
> ourList = [1,2,3,4,5]
[1,2,3,4,5] : List number
>
> firstElement = List.head ourList
Just 1 : Maybe Int
>
> allButFirst = List.tail ourList
Just [2,3,4,5] : Maybe (List Int)
```

Este ajuste en el tipo `Maybe` ocurre debido al siguiente escenario:

¿Qué debería devolver `List.head` por una lista vacía? (Recuerda, Elm no tiene excepciones o nulos).

```
> headOfEmpty = List.head []
Nothing : Maybe Int
>
> tailOfEmpty = List.tail []
Nothing : Maybe (List Int)
>
> tailOfAlmostEmpty = List.tail [1] -- warning ... List is a *linked list* :)
Just [] : Maybe (List Int)
```

Transformando cada elemento de una lista.

`List.map : (a -> b) -> List a -> List b` es una función de orden superior que aplica una función de un parámetro a cada elemento de una lista, devolviendo una nueva lista con los valores modificados.

```
import String

ourList : List String
ourList =
    ["wubba", "lubba", "dub", "dub"]

lengths : List Int
lengths =
    List.map String.length ourList
-- [5,5,3,3]

slices : List String
slices =
    List.map (String.slice 1 3) ourList
-- ["ub", "ub", "ub", "ub"]
```

Si necesita conocer el índice de los elementos, puede usar `List.indexedMap : (Int -> a -> b) -> List a -> List b` :

```
newList : List String
newList =
    List.indexedMap (\index element -> String.concat [toString index, ": ", element]) ourList
-- ["0: wubba","1: lubba","2: dub","3: dub"]
```

Filtrando una lista

`List.filter : (a -> Bool) -> List a -> List a` es una función de orden superior que lleva una función de un parámetro de cualquier valor a un booleano, y aplica esa función a cada elemento de una lista dada, manteniendo solo aquellos elementos para los que la función devuelve `True` . La función que `List.filter` toma como su primer parámetro a menudo se denomina predicado .

```
import String
```

```
catStory : List String
catStory =
    ["a", "crazy", "cat", "walked", "into", "a", "bar"]

-- Any word with more than 3 characters is so long!
isLongWord : String -> Bool
isLongWord string =
    String.length string > 3

longWordsFromCatStory : List String
longWordsFromCatStory =
    List.filter isLongWord catStory
```

Evalúa esto en `elm-repl` :

```
> longWordsFromCatStory
["crazy", "walked", "into"] : List String
>
> List.filter (String.startsWith "w") longWordsFromCatStory
["walked"] : List String
```

Coincidencia de patrones en una lista

Podemos coincidir en listas como cualquier otro tipo de datos, aunque son un tanto únicos, ya que el constructor para construir listas es la función de infijo `::` . (Consulte el ejemplo Creación de una lista para obtener más información sobre cómo funciona).

```
matchMyList : List SomeType -> SomeOtherType
matchMyList myList =
    case myList of
        [] ->
            emptyCase

        (theHead :: theRest) ->
            doSomethingWith theHead theRest
```

Podemos hacer coincidir tantos elementos en la lista como queramos:

```
hasAtLeast2Elems : List a -> Bool
hasAtLeast2Elems myList =
    case myList of
        (e1 :: e2 :: rest) ->
            True

        _ ->
            False

hasAtLeast3Elems : List a -> Bool
hasAtLeast3Elems myList =
    case myList of
        (e1 :: e2 :: e3 :: rest) ->
            True

        _ ->
            False
```

Obteniendo el elemento nth de la lista

`List` no admite el "acceso aleatorio", lo que significa que se necesita más trabajo para obtener, digamos, el quinto elemento de la lista que el primer elemento, y como resultado no hay `List.get nth list` función de `List.get nth list` . Uno tiene que ir desde el principio (`1 -> 2 -> 3 -> 4 -> 5`).

Si necesita acceso aleatorio, puede obtener mejores resultados (y rendimiento) con estructuras de datos de acceso aleatorio, como `Array` , donde tomar el primer elemento requiere la misma cantidad de trabajo que tomar, por ejemplo, el 1000. (complejidad O (1)).

Sin embargo, es posible **(pero desanimado)** obtener el elemento nth:

```
get : Int -> List a -> Maybe a
get nth list =
    list
        |> List.drop (nth - 1)
        |> List.head

fifth : Maybe Int
fifth = get 5 [1..10]
--    = Just 5

nonexistent : Maybe Int
nonexistent = get 5 [1..3]
--          = Nothing
```

De nuevo, esto requiere mucho más trabajo cuanto más grande sea el `nth` argumento.

Reduciendo una lista a un solo valor

En Elm, las funciones de reducción se denominan "pliegues", y existen dos métodos estándar para "plegar" los valores: desde la izquierda, `foldl` , y desde la derecha, `foldr` .

```
> List.foldl (+) 0 [1,2,3]
6 : number
```

Los argumentos para `foldl` y `foldr` son:

- **función reductora** : `newValue -> accumulator -> accumulator`
- valor de arranque del **acumulador**
- **lista** para reducir

Un ejemplo más con función personalizada:

```
type alias Counts =
    { odd : Int
    , even : Int
    }

addCount : Int -> Counts -> Counts
addCount num counts =
```

```
    let
        (incOdd, incEven) =
            if num `rem` 2 == 0
                then (0,1)
                else (1,0)
    in
        { counts
            | odd = counts.odd + incOdd
            , even = counts.even + incEven
        }

> List.foldl
      addCount
      { odd = 0, even = 0 }
      [1,2,3,4,5]
{ odd = 3, even = 2 } : Counts
```

En el primer ejemplo anterior, el programa es el siguiente:

```
List.foldl (+) 0 [1,2,3]
3 + (2 + (1 + 0))
3 + (2 + 1)
3 + 3
6
```

```
List.foldr (+) 0 [1,2,3]
1 + (2 + (3 + 0))
1 + (2 + 3)
1 + 5
6
```

En el caso de una función conmutativa como `(+)` no hay realmente una diferencia.

Pero mira que pasa con `(::)` :

```
List.foldl (::) [] [1,2,3]
3 :: (2 :: (1 :: []))
3 :: (2 :: [1])
3 :: [2,1]
[3,2,1]
```

```
List.foldr (::) [] [1,2,3]
1 :: (2 :: (3 :: []))
1 :: (2 :: [3])
1 :: [2,3]
[1,2,3]
```

Creando una lista repitiendo un valor

```
> List.repeat 3 "abc"
["abc","abc","abc"] : List String
```

Puedes darle a `List.repeat` cualquier valor:

```
> List.repeat 2 {a = 1, b = (2,True)}
[{a = 1, b = (2,True)}, {a = 1, b = (2,True)}]
  : List {a : Int, b : (Int, Bool)}
```

Ordenar una lista

Por defecto, `List.sort` ordena en orden ascendente.

```
> List.sort [3,1,5]
[1,3,5] : List number
```

`List.sort` necesita que los elementos de la lista sean `comparable` . Eso significa: `String` , `Char` , `number` (`Int` y `Float`), `List` de `comparable` o tupla de `comparable` .

```
> List.sort [(5,"ddd"),(4,"zzz"),(5,"aaa")]
[(4,"zzz"),(5,"aaa"),(5,"ddd")] : List ( number, String )

> List.sort [[3,4],[2,3],[4,5],[1,2]]
[[1,2],[2,3],[3,4],[4,5]] : List (List number)
```

No puede ordenar listas de `Bool` u objetos con `List.sort` . Para eso vea Ordenar una lista con un comparador personalizado.

```
> List.sort [True, False]
-- error, can't compare Bools
```

Ordenar una lista con comparador personalizado

`List.sortWith` permite ordenar listas con datos de cualquier forma; usted le proporciona una función de comparación.

```
compareBools : Bool -> Bool -> Order
compareBools a b =
    case (a,b) of
        (False, True) ->
            LT

        (True, False) ->
            GT

        _ ->
            EQ

> List.sortWith compareBools [False, True, False, True]
[False, False, True, True] : List Bool
```

Invertir una lista

Nota: esto no es muy eficiente debido a la naturaleza de la `List` (ver Comentarios a continuación). Será mejor **construir la lista de la manera "correcta" desde el principio** que construirla y luego revertirla.

```
> List.reverse [1,3,5,7,9]
[9,7,5,3,1] : List number
```

Ordenar una lista en orden descendente

Por defecto, `List.sort` ordena en orden ascendente, con la función de `compare` .

Hay dos formas de clasificar en orden descendente: una eficiente y otra ineficiente.

1. **La forma eficiente** : `List.sortWith` y una función de comparación descendente.

```
descending a b =
    case compare a b of
      LT -> GT
      EQ -> EQ
      GT -> LT

> List.sortWith descending [1,5,9,7,3]
[9,7,5,3,1] : List number
```

2. El modo ineficiente **(¡desalentado!)** : `List.sort` y luego `List.reverse` .

```
> List.reverse (List.sort [1,5,9,7,3])
[9,7,5,3,1] : List number
```

Ordenar una lista por un valor derivado

`List.sortBy` permite usar una función en los elementos y usar su resultado para la comparación.

```
> List.sortBy String.length ["longest","short","medium"]
["short","medium","longest"] : List String
-- because the lengths are: [7,5,6]
```

También funciona muy bien con los registros de acceso:

```
people =
    [ { name = "John", age = 43 }
    , { name = "Alice", age = 30 }
    , { name = "Rupert", age = 12 }
    ]

> List.sortBy .age people
[ { name = "Rupert", age = 12 }
, { name = "Alice", age = 30 }
, { name = "John", age = 43 }
] : List {name: String, age: number}

> List.sortBy .name people
[ { name = "Alice", age = 30 }
, { name = "John", age = 43 }
, { name = "Rupert", age = 12 }
] : List {name: String, age: number}
```

Lea Listas e iteración en línea: https://riptutorial.com/es/elm/topic/1635/listas-e-iteracion

Capítulo 11: Puertos (interoperabilidad JS)

Sintaxis

- Elm (recibir): puerto functionName: (value -> msg) -> Sub msg
- JS (envío): app.ports.functionName.send (valor)
- Elm (enviando): puerto functionName: args -> Cmd msg
- JS (recibiendo): app.ports.functionName.subscribe (function (args) {...});

Observaciones

Consulte http://guide.elm-lang.org/interop/javascript.html de *The Elm Guide* para ayudar a entender estos ejemplos.

Examples

Visión general

Un módulo, que utiliza puertos, debe tener una palabra clave de `port` en su definición de módulo.

```
port module Main exposing (..)
```

Es imposible usar puertos con `Html.App.beginnerProgram` , ya que no permite el uso de suscripciones o comandos.

Los puertos están integrados para actualizar el bucle de `Html.App.program` o `Html.App.programWithFlags` .

Nota

`program` y `programWithFlags` en elm 0.18 están dentro del paquete `Html` lugar de `Html.App` .

Saliente

Los puertos salientes se utilizan como Comandos, que devuelve de su función de `update` .

Lado olmo

Definir puerto de salida:

```
port output : () -> Cmd msg
```

En este ejemplo, enviamos un Tuple vacío, solo para activar una suscripción en el lado de JavaScript.

Para hacerlo, tenemos que aplicar `output` función de `output` con un Tuple vacío como argumento, para obtener un comando para enviar los datos salientes de Elm.

```
update msg model =
    case msg of
        TriggerOutgoing data ->
            ( model, output () )
```

Lado de JavaScript

Inicializar la aplicación:

```
var root = document.body;
var app = Elm.Main.embed(root);
```

Suscríbete a un puerto con un nombre correspondiente:

```
app.ports.output.subscribe(function () {
    alert('Outgoing message from Elm!');
});
```

Nota

A partir del `0.17.0` , el mensaje saliente inmediato a JavaScript desde su estado `initial` no tendrá efecto.

```
init : ( Model, Cmd Msg )
init =
    ( Model 0, output () ) -- Nothing will happen
```

Vea la solución en el siguiente ejemplo.

Entrante

Los datos entrantes de JavaScript están pasando por las suscripciones.

Lado olmo

Primero, necesitamos definir un puerto entrante, usando la siguiente sintaxis:

```
port input : (Int -> msg) -> Sub msg
```

Podemos usar `Sub.batch` si tenemos varias suscripciones, este ejemplo solo contendrá una suscripción al `input port`

```
subscriptions : Model -> Sub Msg
subscriptions model =
    input Get
```

Luego tiene que pasar las `subscriptions` a su programa `Html.program` :

```
main =
    Html.program
        { init = init
        , view = view
        , update = update
        , subscriptions = subscriptions
        }
```

Lado de JavaScript

Inicializar la aplicación:

```
var root = document.body;
var app = Elm.Main.embed(root);
```

Envía el mensaje a Elm:

```
var counter = 0;

document.body.addEventListener('click', function () {
    counter++;
    app.ports.input.send(counter);
});
```

Nota

Tenga en cuenta que a partir del `0.17.0` el `app.ports.input.send(counter);` inmediato `app.ports.input.send(counter);` Después de la inicialización de la aplicación no tendrá ningún efecto!

Pase todos los datos necesarios para el inicio como indicadores que utilizan `Html.programWithFlags`

Mensaje saliente inmediato en el arranque en 0.17.0

Para enviar un mensaje inmediato con datos a JavaScript, debe activar una acción desde su `init` .

```
init : ( Model, Cmd Msg )
init =
```

```
    ( Model 0, send SendOutgoing )


send : msg -> Cmd msg
send msg =
    Task.perform identity identity (Task.succeed msg)
```

Empezar

index.html

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title>Trying out ports</title>
  </head>
  <body>
    <div id="app"></div>
    <script src="elm.js"></script>
    <script>

      var node = document.getElementById('app');
      var app = Elm.Main.embed(node);

      // subscribe to messages from Elm
      app.ports.toJs.subscribe(function(messageFromElm) {
        alert(messageFromElm);
        // we could send something back by
        // app.ports.fromJs.send('Hey, got your message! Sincerely, JS');
      });

      // wait three seconds and then send a message from JS to Elm
      setTimeout(function () {
        app.ports.fromJs.send('Hello from JS');
      }, 3000);

    </script>
  </body>
</html>
```

Main.elm

```
port module Main exposing (..)

import Html

port toJs : String -> Cmd msg
port fromJs : (String -> msg) -> Sub msg

main =
   Html.program
        { init = (Nothing, Cmd.none) -- our model will be the latest message from JS (or
Nothing for 'no message yet')
        , update = update
        , view = view
        , subscriptions = subscriptions
        }
```

```
type Msg
    = GotMessageFromJs String

update msg model =
    case msg of
        GotMessageFromJs message ->
            (Just message, toJs "Hello from Elm")

view model =
    case model of
        Nothing ->
            Html.text "No message from JS yet :("
        Just message ->
            Html.text ("Last message from JS: " ++ message)

subscriptions model =
    fromJs GotMessageFromJs
```

Instale el paquete `elm-lang/html` si aún no lo ha hecho con `elm-package install elm-lang/html --yes` .

Compile este código usando `elm-make Main.elm --yes --output elm.js` para que el archivo HTML lo encuentre.

Si todo va bien, debería poder abrir el archivo `index.html` con el texto "Sin mensaje" que se muestra. Después de tres segundos, JS envía un mensaje, Elm lo recibe, cambia su modelo, envía una respuesta, JS lo recibe y abre una alerta.

Lea Puertos (interoperabilidad JS) en línea: https://riptutorial.com/es/elm/topic/2200/puertos--interoperabilidad-js-

Capítulo 12: Recopilación de datos: tuplas, registros y diccionarios

Examples

Tuplas

Las tuplas son listas ordenadas de valores de cualquier tipo.

```
(True, "Hello!", 42)
```

Es imposible cambiar la estructura de un Tuple o actualizar el valor.

Las tuplas en Elm se consideran un tipo de datos primitivo, lo que significa que no es necesario importar ningún módulo para usar tuplas.

Valores de acceso

El módulo básico tiene dos funciones auxiliares para acceder a los valores de una tupla con una longitud de dos `( a, b )` sin utilizar la coincidencia de patrones:

```
fst (True, "Hello!") -- True
snd (True, "Hello!") -- "Hello!"
```

Los valores de acceso de las tuplas con una mayor longitud se realizan mediante la coincidencia de patrones.

La coincidencia de patrones

Las tuplas son extremadamente útiles en combinación con la coincidencia de patrones:

```
toggleFlag: (Sting, Bool) -> (Sting, Bool)
toggleFlag (name, flag) =
    (name, not flag)
```

Observaciones sobre las tuplas

Las tuplas contienen menos de 7 valores de tipo de datos `comparable`

Los diccionarios

Los diccionarios se implementan en una biblioteca principal de Dict .

> Un diccionario mapeando claves únicas a los valores. Las claves pueden ser de cualquier tipo comparable. Esto incluye Int, Float, Time, Char, String y tuplas o listas de tipos comparables.
>
> Las operaciones de inserción, eliminación y consulta llevan tiempo O (log n).

A diferencia de las tuplas y los registros, los diccionarios pueden cambiar su estructura, en otras palabras, es posible agregar y eliminar claves.

Es posible actualizar un valor por una clave.

También es posible acceder o actualizar un valor utilizando teclas dinámicas.

Valores de acceso

Puede recuperar un valor de un Diccionario utilizando una función `Dict.get` .

Escriba la definición de `Dict.get` :

```
get : comparable -> Dict comparable v -> Maybe v
```

Siempre devolverá `Maybe v` , porque es posible intentar obtener un valor con una clave que no existe.

```
import Dict

initialUsers =
      Dict.fromList [ (1, "John"), (2, "Brad") ]

getUserName id =
  initialUsers
  |> Dict.get id
  |> Maybe.withDefault "Anonymous"

getUserName 2 -- "Brad"
getUserName 0 -- "Anonymous"
```

Actualizando valores

La operación de actualización en un Diccionario se realiza utilizando `Maybe.map` , ya que la clave solicitada puede estar ausente.

```
import Dict

initialUsers =
  Dict.fromList [ (1, "John"), (2, "Brad") ]
```

```
updatedUsers =
  Dict.update 1 (Maybe.map (\name -> name ++ " Johnson")) initialUsers

Maybe.withDefault "No user" (Dict.get 1 updatedUsers) -- "John Johnson"
```

Archivos

Registro es un conjunto de pares clave-valor.

```
greeter =
    { isMorning: True
    , greeting: "Good morning!"
    }
```

Es imposible acceder a un valor por una clave inexistente.

Es imposible modificar dinámicamente la estructura del Registro.

Los registros solo le permiten actualizar valores mediante claves constantes.

Valores de acceso

No se puede acceder a los valores con una clave dinámica para evitar posibles errores en el tiempo de ejecución:

```
isMorningKeyName =
    "isMorning "


greeter[isMorningKeyName] -- Compiler error
greeter.isMorning -- True
```

La sintaxis alternativa para acceder al valor le permite extraer el valor, mientras avanza a través del Registro:

```
greeter
    |> .greeting
    |> (++) " Have a nice day!" -- "Good morning! Have a nice day!"
```

Tipos de extensión

A veces, desearía que la firma de un parámetro restringiera los tipos de registro que pasa a las funciones. Extender los tipos de registro hace que la idea de supertipos no sea necesaria. El siguiente ejemplo muestra cómo se puede implementar este concepto:

```
type alias Person =
    { name : String
    }
```

```
type alias Animal =
    { name : String
    }


peter : Person
peter =
    { name = "Peter" }


dog : Animal
dog =
    { name = "Dog" }


getName : { a | name : String } -> String
getName livingThing =
    livingThing.name


bothNames : String
bothNames =
    getName peter ++ " " ++ getName dog
```

Incluso podríamos llevar los registros extendidos un paso más allá y hacer algo como:

```
type alias Named a = { a | name : String }
type alias Totalled a = { a | total : Int }


totallyNamed : Named ( Totalled { age : Int })
totallyNamed =
  { name = "Peter Pan"
  , total = 1337
  , age = 14
  }
```

Ahora tenemos formas de pasar estos tipos parciales en funciones:

```
changeName : Named a -> String -> Named a
changeName a newName =
  { a | name = newName }

cptHook = changeName totallyNamed "Cpt. Hook" |> Debug.log "who?"
```

Actualizando valores

Elm tiene una sintaxis especial para las actualizaciones de registros:

```
model =
    { id: 1
    , score: 0
    , name: "John Doe"
```

```
    }


update model =
    { model
       | score = model.score + 100
       | name = "John Johnson"
    }
```

Lea Recopilación de datos: tuplas, registros y diccionarios en línea: https://riptutorial.com/es/elm/topic/2166/recopilacion-de-datos--tuplas--registros-y-diccionarios

Capítulo 13: Suscripciones

Observaciones

Las suscripciones son medios para escuchar entradas. Puertos entrantes , eventos de teclado o mouse, mensajes WebSocket, geolocalización y cambios de visibilidad de la página, todos pueden servir como entradas.

Examples

Suscripción básica al evento Time.every con 'cancelar suscripción'

0.18.0

El modelo se pasa a las suscripciones, lo que significa que cada cambio de estado puede modificar las suscripciones.

```
import Html exposing ( Html, div, text, button )
import Html.Events exposing ( onClick )
import Time

main : Program Never Model Msg
main =
    Html.program
        { init = init
        , update = update
        , subscriptions = subscriptions
        , view = view
        }

-- MODEL

type alias Model =
    { time: Time.Time
    , suspended: Bool
    }

init : (Model, Cmd Msg)
init =
    ( Model 0 False, Cmd.none )

-- UPDATE

type Msg
    = Tick Time.Time
    | SuspendToggle

update : Msg -> Model -> ( Model, Cmd Msg )
update msg model =
    case msg of
        Tick newTime ->
            ( { model | time = newTime }, Cmd.none )
```

```
        SuspendToggle ->
            ( { model | suspended = not model.suspended }, Cmd.none )

-- SUBSCRIPTIONS

subscriptions : Model -> Sub Msg
subscriptions model =
    if model.suspended then
        Sub.none
    else
        Time.every Time.second Tick

-- VIEW

view : Model -> Html Msg
view model =
    div []
        [ div [] [ text <| toString model ]
        , button [ onClick SuspendToggle ] [ text ( if model.suspended then "Resume" else
"Suspend" ) ]
        ]
```

Lea Suscripciones en línea: https://riptutorial.com/es/elm/topic/4279/suscripciones

Capítulo 14: Tipos, variables de tipo y constructores de tipo

Observaciones

¡Por favor juega con estos conceptos para dominarlos realmente! `elm-repl` (ver la Introducción al REPL) es probablemente un buen lugar para jugar con el código anterior. También puedes jugar con `elm-repl` online .

Examples

Tipos de datos comparables

Los tipos comparables son tipos primitivos que pueden compararse utilizando operadores de comparación del módulo de Conceptos básicos , como: `(<)` , `(>)` , `(<=)` , `(>=)` , `max` , `min` , `compare`

Los tipos comparables en Elm son `Int` , `Float` , `Time` , `Char` , `String` y tuplas o listas de tipos comparables.

En la documentación o en las definiciones de tipo se les conoce como una variable de tipo especial `comparable` , por ejemplo. ver definición de tipo para la función `Basics.max` :

```
max : comparable -> comparable -> comparable
```

Tipo de firmas

En Elm, los valores se declaran escribiendo un nombre, un signo igual y luego el valor real:

```
someValue = 42
```

Las funciones también son valores, con la adición de tomar un valor o valores como argumentos. Por lo general se escriben de la siguiente manera:

```
double n = n * 2
```

Cada valor en Elm tiene un tipo. Los tipos de los valores anteriores se *infiere* por el compilador dependiendo de cómo se utilizan. Pero es una buena práctica siempre declarar explícitamente el tipo de cualquier valor de nivel superior, y para hacerlo, escriba una *firma de tipo de la* siguiente manera:

```
someValue : Int
someValue =
    42
```

```
someOtherValue : Float
someOtherValue =
    42
```

Como podemos ver, `42` se puede definir como *o bien* un `Int` o un `Float` . Esto tiene sentido intuitivo, pero consulte **Variables de tipo** para obtener más información.

Las firmas de tipos son particularmente valiosas cuando se usan con funciones. Aquí está la función de duplicación de antes:

```
double : Int -> Int
double n =
    n * 2
```

Esta vez, la firma tiene una `->` , una flecha, y la pronunciaríamos como "int to int", o "toma un entero y devuelve un entero". `->` indica que al dar un valor `Int double` como argumento, `double` devolverá un `Int` . Por lo tanto, toma un entero a un entero:

```
> double
<function> : Int -> Int

> double 3
6 : Int
```

Tipos basicos

En `elm-repl` , escriba un fragmento de código para obtener su valor y tipo inferido. Intente lo siguiente para aprender sobre los diversos tipos que existen:

```
> 42
42 : number

> 1.987
1.987 : Float

> 42 / 2
21 : Float

> 42 % 2
0 : Int

> 'e'
'e' : Char

> "e"
"e" : String

> "Hello Friend"
"Hello Friend" : String

> ['w', 'o', 'a', 'h']
['w', 'o', 'a', 'h'] : List Char

> ("hey", 42.42, ['n', 'o'])
```

```
("hey", 42.42, ['n', 'o']) : ( String, Float, List Char )

> (1, 2.1, 3, 4.3, 'c')
(1,2.1,3,4.3,'c') : ( number, Float, number', Float, Char )

> {}
{} : {}

> { hey = "Hi", someNumber = 43 }
{ hey = "Hi", someNumber = 43 } : { hey : String, someNumber : number }

> ()
() : ()
```

`{}` es el tipo de registro vacío, y `()` es el tipo de tupla vacío. Este último se utiliza a menudo para fines de evaluación perezosa. Vea el ejemplo correspondiente en Funciones y Aplicación Parcial .

Observe cómo el `number` aparece sin capitalizar. Esto indica que es una **Variable de tipo** y, además, el `number` palabra en particular se refiere a una **Variable de tipo especial** que puede ser `Int` o `Float` (consulte las secciones correspondientes para obtener más información). Sin embargo, los tipos siempre están en mayúsculas, como `Char` , `Float` , `List String` , etcétera.

Variables de tipo

Las variables de tipo son nombres sin capitalizar en las firmas de tipo. A diferencia de sus homólogos en mayúsculas, como `Int` y `String` , no representan un solo tipo, sino cualquier tipo. Se utilizan para escribir funciones genéricas que pueden operar en *cualquier* tipo o tipo, y son particularmente útiles para escribir operaciones en contenedores como `List` o `Dict` . La función `List.reverse` , por ejemplo, tiene la siguiente firma:

```
reverse : List a -> List a
```

Lo que significa que puede funcionar en una lista de *cualquier valor de tipo* , por lo que `List Int` , `List (List String)` , ambos y cualquier otro puede `reversed` la misma manera. Por lo tanto, `a` es una variable de tipo que puede representar cualquier tipo.

La función `reverse` podría haber usado *cualquier* nombre de variable sin capitalizar en su firma de tipo, excepto por un puñado de nombres de **variables de tipo especial** , como el `number` (consulte el ejemplo correspondiente en este enlace para obtener más información):

```
reverse : List lol -> List lol

reverse : List wakaFlaka -> List wakaFlaka
```

Los nombres de las variables de tipo se vuelven significativos solo cuando hay *diferentes* variables de tipo dentro de una única firma, ejemplificada por la función de `map` en las listas:

```
map : (a -> b) -> List a -> List b
```

`map` toma alguna función de cualquier tipo `a` a cualquier tipo `b` , junto con una lista con los

elementos de algún tipo `a` , y devuelve una lista de elementos de algún tipo `b` , que se pone mediante la aplicación de la función dada a cada elemento de la lista.

Hagamos la firma concreta para ver mejor esto:

```
plusOne : Int -> Int
plusOne x =
    x + 1

> List.map plusOne
<function> : List Int -> List Int
```

Como podemos ver, tanto `a = Int` como `b = Int` en este caso. Pero, si el `map` tuviera una firma de tipo como `map : (a -> a) -> List a -> List a` , entonces *solo* funcionaría en funciones que operan en un solo tipo, y nunca sería capaz de cambiar el tipo de una lista mediante la función de `map` . Pero dado que el tipo de firma del `map` tiene múltiples variables de tipo diferente, `a` y `b` , podemos usar `map` para cambiar el tipo de una lista:

```
isOdd : Int -> Bool
isOdd x =
    x % 2 /= 0

> List.map isOdd
<function> : List Int -> List Bool
```

En este caso, `a = Int` y `b = Bool` . Por lo tanto, para poder usar funciones que pueden tomar y devolver *diferentes* tipos, debe usar diferentes variables de tipo.

Tipo de alias

A veces queremos darle a un tipo un nombre más descriptivo. Digamos que nuestra aplicación tiene un tipo de datos que representa a los usuarios:

```
{ name : String, age : Int, email : String }
```

Y nuestras funciones en los usuarios tienen firmas de tipo en la línea de:

```
prettyPrintUser : { name : String, age : Int, email : String } -> String
```

Esto podría volverse bastante difícil de manejar con un tipo de registro más grande para un usuario, así que usemos un *alias de tipo* para reducir el tamaño y darle un nombre más significativo a esa estructura de datos:

```
type alias User =
    { name: String
    , age : Int
    , email : String
    }


prettyPrintUser : User -> String
```

Los alias de tipo hacen que sea mucho más limpio definir y usar un modelo para una aplicación:

```
type alias Model =
    { count : Int
    , lastEditMade : Time
    }
```

Usar el `type alias` literalmente solo alias un tipo con el nombre que le das. Usar el tipo de `Model` anterior es exactamente lo mismo que usar `{ count : Int, lastEditMade : Time }` . Aquí hay un ejemplo que muestra cómo los alias no son diferentes a los tipos subyacentes:

```
type alias Bugatti = Int

type alias Fugazi = Int

unstoppableForceImmovableObject : Bugatti -> Fugazi -> Int
unstoppableForceImmovableObject bug fug =
    bug + fug

> unstoppableForceImmovableObject 09 87
96 : Int
```

Un alias de tipo para un tipo de registro define una función constructora con un argumento para cada campo en el orden de declaración.

```
type alias Point = { x : Int, y : Int }

Point 3 7
{ x = 3, y = 7 } : Point

type alias Person = { last : String, middle : String, first : String }

Person "McNameface" "M" "Namey"
{ last = "McNameface", middle = "M", first = "Namey" } : Person
```

Cada alias de tipo de registro tiene su propio orden de campo incluso para un tipo compatible.

```
type alias Person = { last : String, middle : String, first : String }
type alias Person2 = { first : String, last : String, middle : String }

Person2 "Theodore" "Roosevelt" "-"
{ first = "Theodore", last = "Roosevelt", middle = "-" } : Person2

a = [ Person "Last" "Middle" "First", Person2 "First" "Last" "Middle" ]
[{ last = "Last", middle = "Middle", first = "First" },{ first = "First", last = "Last",
middle = "Middle" }] : List Person2
```

Mejora de la seguridad de tipos utilizando nuevos tipos

Los tipos de alias reducen la repetición y mejoran la legibilidad, pero no son más seguros de lo que son los tipos con alias. Considera lo siguiente:

```
type alias Email = String
```

```
type alias Name = String

someEmail = "holmes@private.com"

someName = "Benedict"

sendEmail : Email -> Cmd msg
sendEmail email = ...
```

Usando el código anterior, podemos escribir `sendEmail someName` , y se compilará, aunque realmente no debería, porque a pesar de que los nombres y los correos electrónicos son `String` , son cosas completamente diferentes.

Podemos distinguir realmente una `String` de otra `String` en el nivel de tipo creando un nuevo **tipo** . Aquí hay un ejemplo que reescribe el `Email` como un `type` lugar de un `type alias` :

```
module Email exposing (Email, create, send)

type Email = EmailAddress String

isValid : String -> Bool
isValid email =
  -- ...validation logic

create : String -> Maybe Email
create email =
    if isValid email then
        Just (EmailAddress email)
    else
        Nothing

send : Email -> Cmd msg
send (EmailAddress email) = ...
```

Nuestra función `isValid` hace algo para determinar si una cadena es una dirección de correo electrónico válida. El `create` función comprueba si un determinado `String` es un correo electrónico válido, devolviendo un `Maybe` -wrapped `Email` para asegurarse de que sólo volvemos direcciones validadas. Si bien podemos esquivar la verificación de validación construyendo un `Email` directamente escribiendo `EmailAddress "somestring"` , si nuestra declaración de módulo no expone el constructor `EmailAddress` , como se muestra aquí

```
module Email exposing (Email, create, send)
```

entonces ningún otro módulo tendrá acceso al constructor `EmailAddress` , aunque aún pueden usar el tipo de `Email` en las anotaciones. La **única** forma de crear un nuevo `Email` fuera de este módulo es mediante la función de `create` que proporciona, y esa función garantiza que solo devolverá direcciones de correo electrónico válidas en primer lugar. Por lo tanto, esta API guía automáticamente al usuario por el camino correcto a través de su seguridad de tipo: `send` solo funciona con los valores construidos por `create` , que realiza una validación, e impone el manejo de correos electrónicos no válidos, ya que devuelve un `Maybe Email` .

Si desea exportar el constructor de `Email` , puede escribir

```
module Email exposing (Email(EmailAddress), create, send)
```

Ahora cualquier archivo que importe `Email` también puede importar su constructor. En este caso, hacerlo permitiría a los usuarios eludir la validación y `send` correos electrónicos no válidos, pero no siempre está creando una API como esta, por lo que exportar constructores puede ser útil. Con un tipo que tiene varios constructores, también es posible que solo desee exportar algunos de ellos.

Construyendo tipos

La combinación de palabras clave de `type alias` da un nuevo nombre para un tipo, pero la palabra clave de `type` en aislamiento declara un nuevo tipo. Examinemos uno de los más fundamentales de estos tipos: `Maybe`

```
type Maybe a
    = Just a
    | Nothing
```

Lo primero que se debe tener en cuenta es que el tipo `Maybe` se declara con una **variable** de **tipo** de `a` . La segunda cosa a tener en cuenta es el carácter de la tubería, `|` , que significa "o". En otras palabras, algo de tipo `Maybe a` *sea* `Just a` *or* `Nothing` .

Cuando escribe el código anterior, `Just` and `Nothing` entra en el alcance como *constructores de valor* , y `Maybe` entra en el alcance como un *constructor de tipo* . Estas son sus firmas:

```
Just : a -> Maybe a

Nothing : Maybe a

Maybe : a -> Maybe a -- this can only be used in type signatures
```

Debido a la *variable de tipo* `a` , cualquier tipo puede "ajustarse dentro" del tipo `Maybe` . Entonces, `Maybe Int` , `Maybe (List String)` , o `Maybe (Maybe (List Html))` , son todos tipos válidos. Al desestructurar cualquier valor de `type` con una expresión de `case` , debe tener en cuenta cada posible instanciación de ese tipo. En el caso de un valor de tipo `Maybe a` , debe tener en cuenta tanto el caso `Just a` case como el caso `Nothing` :

```
thing : Maybe Int
thing =
    Just 3

blah : Int
blah =
    case thing of
        Just n ->
            n

        Nothing ->
            42

-- blah = 3
```

Intente escribir el código anterior sin la cláusula `Nothing` en la expresión del `case` : no se compilará. Esto es lo que hace que el constructor de tipo `Maybe` un gran patrón para expresar valores que pueden no existir, ya que te obliga a manejar la lógica de cuando el valor es `Nothing` .

El nunca escribe

El tipo `Never` se puede construir (el módulo `Basics` no ha exportado su **constructor de valores** y tampoco le ha dado ninguna otra función que devuelva `Never`). No hay ningún valor `never : Never` o una función `createNever : ?? -> Never` .

Esto tiene sus beneficios: puede codificar en un sistema de tipos una posibilidad que no puede ocurrir. Esto se puede ver en tipos como `Task Never Int` que garantiza que tendrá éxito con un `Int` ; o `Program Never` eso no tomará ningún parámetro al inicializar el código Elm desde JavaScript.

Variables Tipo Especial

Elm define las siguientes variables de tipo especial que tienen un significado particular para el compilador:

- **`comparable`** : Compuesto por `Int` , `Float` , `Char` , `String` y tuples. Esto permite el uso de los operadores `<` y `>` .

  Ejemplo: Podría definir una función para encontrar los elementos más pequeños y más grandes en una lista (`extent`). Piensas qué tipo de firma escribir. Por un lado, puede escribir `extentInt : List Int -> Maybe (Int, Int)` y `extentChar : List Char -> Maybe (Char, Char)` y otro para `Float` and `String` . La implementación de estos sería la misma:

  ```
  extentInt list =
    let
      helper x (minimum, maximum) =
        ((min minimum x), (max maximum x))
    in
      case list of
        [] ->
          Nothing
        x :: xs ->
          Just <| List.foldr helper (x, x) xs
  ```

  Es posible que tenga la tentación de simplemente escribir la `extent : List a -> Maybe (a, a)` , pero el compilador no le permitirá hacer esto, porque las funciones `min` y `max` no están definidas para estos tipos (NB: estos son solo envoltorios simples alrededor del `<` operador mencionado anteriormente). Puede resolver esto definiendo la `extent : List comparable -> Maybe (comparable, comparable)` . Esto permite que su solución sea *polimórfica* , lo que significa que funcionará para más de un tipo.

- **`number`** : Compuesto por `Int` y `Float` . Permite el uso de operadores aritméticos excepto división. Luego puede definir, por ejemplo, la `sum : List number -> number` y hacer que funcione tanto para ints como para flotadores.

- **`appendable`** : compuesto por `String` , `List` . Permite el uso del operador `++` .

- **`compappend`** : esto aparece a veces, pero es un detalle de implementación del compilador. Actualmente esto no se puede utilizar en sus propios programas, pero a veces se menciona.

Tenga en cuenta que en una anotación de tipo como esta: `number -> number -> number` todos se refieren al mismo tipo, por lo que pasar `Int -> Float -> Int` sería un error de tipo. Puede resolver esto agregando un sufijo al tipo variable nombre: `number -> number' -> number''` luego compilaría bien.

No hay un nombre oficial para estos, a veces se les llama:

- Variables Tipo Especial
- Variables de tipo tipo clase de clase
- Clases de seudo tipo

Esto se debe a que funcionan como las Clases de tipos de Haskell, pero sin que el usuario pueda definirlas.

Lea Tipos, variables de tipo y constructores de tipo en línea:
https://riptutorial.com/es/elm/topic/2648/tipos--variables-de-tipo-y-constructores-de-tipo

Creditos

S. No	Capítulos	Contributors
1	Empezando con Elm Language	2426021684, alejosocorro, AnimiVulpis, Community, Douglas Correa, gabrielperales, gar, halfzebra, Jakub Hampl, jmite, JustGage, lonelyelk, Martin Janiczek, mrkovec, thSoft, Zimm i48
2	Decodificadores JSON personalizados	Khaled Jouda
3	Depuración	AnimiVulpis, bdukes, Jonathan de M., Martin Janiczek, Nicholas Montaño
4	Funciones y aplicación parcial.	Art Yerkes, halfzebra, lonelyelk, Martin Janiczek, Nicholas Montaño, Ryan Plant, Will White
5	Haciendo funciones de actualización complejas con ccapndave / elm-update-extra	Mateus Felipe
6	Integración de backend	lonelyelk
7	Json.Decodificar	ivanceras, Jonathan de M., lonelyelk, Matthew Rankin
8	La arquitectura del olmo	AnimiVulpis, halfzebra, mrkovec, Ryan Plant, vlad_o, Zimm i48
9	La coincidencia de patrones	Gerald Kaszuba, Jakub Hampl, Tosh
10	Listas e iteración	2426021684, AnimiVulpis, jmite, lonelyelk, Martin Janiczek, Nicholas Montaño, Zimm i48
11	Puertos (interoperabilidad JS)	Adam Bowen, gabrielperales, halfzebra, Martin Janiczek, Nicholas Montaño
12	Recopilación de datos: tuplas, registros y diccionarios	halfzebra, Martin Janiczek, Mr. Baudin

13	Suscripciones	lonelyelk, mrkovec, Tosh
14	Tipos, variables de tipo y constructores de tipo	Art Yerkes, bright-star, halfzebra, Jakub Hampl, Joseph Weissman, Martin Janiczek, Nicholas Montaño, Zimm i48