eBook Gratuit

APPRENEZ

Elm Language

eBook gratuit non affilié créé à partir des **contributeurs de Stack Overflow.**

#elm

Table des matières

À propos

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

Chapitre 1: Démarrer avec Elm Language

Remarques

[Elm] [1] est un langage de programmation fonctionnel convivial compilant JavaScript. Elm se concentre sur les interfaces graphiques basées sur un navigateur, les applications à une seule page.

Les utilisateurs le louent généralement pour:

- Aucune exception d'exécution.
- Les meilleures erreurs de compilation jamais
- La facilité de refactorisation.
- Système de type expressif
- The Elm Architecture , dont Redux s'inspire.

Versions

Version	Date de sortie
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

Installation

Pour démarrer le développement avec Elm, vous devez installer un ensemble d'outils appelé elm-platform .

Il comprend: elm-make , elm-réacteur , elm-repl et elm-package .

Tous ces outils sont disponibles via l'interface de ligne de commande, en d'autres termes, vous pouvez les utiliser depuis votre terminal.

Choisissez l'une des méthodes suivantes pour installer Elm:

Utiliser l'installateur

Téléchargez le programme d'installation à partir du site officiel et suivez l'assistant d'installation.

Utiliser npm

Vous pouvez utiliser Node Package Manager pour installer la plateforme Elm.

Installation globale:

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

Installation locale:

```
$ npm install elm
```

Les outils de plate-forme Elm installés localement sont accessibles via:

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

En utilisant homebrew

```
$ brew install elm
```

Basculer entre les versions avec elm-use

Installez elm-use

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

Passer à une version orme plus ancienne ou plus récente

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

Lectures complémentaires

Apprenez à initialiser et à construire votre premier projet.

Bonjour le monde

Voir comment compiler ce code dans Initialize et build

```
import Html

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

Rédacteurs

Atome

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

Table lumineuse

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

Texte sublime

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

Vim

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

Emacs

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

IntelliJ IDEA

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

Supports

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

Code VS

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

Initialiser et construire

La plate-forme Elm doit être installée sur votre ordinateur. Le tutoriel suivant est écrit avec l'hypothèse que vous connaissez bien le terminal.

Initialisation

Créez un dossier et accédez-y avec votre terminal:

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

Initialiser le projet Elm et installer les dépendances de base:

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

`elm-package.json` et le dossier `elm-stuff` devraient apparaître dans votre projet.

Créez le point d'entrée de votre application `Main.elm` et collez l'exemple Hello World dans celui-ci.

Construire le projet

Pour construire votre premier projet, exécutez:

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

Cela produira `index.html` avec le fichier `Main.elm` (et toutes les dépendances) compilé en JavaScript et intégré au HTML. **Essayez de l'ouvrir dans votre navigateur!**

Si cela échoue avec l'erreur, `I cannot find module 'Html'.` Cela signifie que vous n'utilisez pas la dernière version d'Elm. Vous pouvez résoudre le problème en mettant à niveau Elm et en reprenant la première étape, ou en exécutant la commande suivante:

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

Si vous avez votre propre fichier `index.html` (par exemple, lorsque vous travaillez avec des ports), vous pouvez également compiler vos fichiers Elm dans un fichier JavaScript:

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

Plus d'infos dans l'exemple Intégration dans HTML .

Guide de style et format orme

Le guide de style officiel se trouve sur la page d'accueil et concerne généralement:

- lisibilité (au lieu de compacité)
- facilité de modification
- diffs propres

Cela signifie que, par exemple, ceci:

```
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'
```

est considéré **meilleur** 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

L'outil elm-format vous aide à formater **automatiquement** votre code source (généralement lors de la sauvegarde), dans la même veine que le langage gofmt de Go language. Encore une fois, la valeur sous-jacente consiste à avoir **un style cohérent** et à enregistrer des arguments et des guerres de flamme sur diverses questions telles que les *onglets, les espaces* ou la *longueur d'indentation* .

Vous pouvez installer `elm-format` suivant les instructions du dépôt Github . Configurez ensuite votre éditeur pour formater automatiquement les fichiers Elm ou exécutez manuellement les `elm-format FILE_OR_DIR --yes` .

Intégration dans HTML

Il existe trois possibilités pour insérer du code Elm dans une page HTML existante.

Intégrer dans la balise body

En supposant que vous ayez compilé l'exemple Hello World dans le fichier `elm.js` , vous pouvez laisser Elm reprendre la `<body>` comme `elm.js` :

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

AVERTISSEMENT : Parfois, certaines extensions de chrome gâchent avec `<body>` ce qui peut entraîner la rupture de la production de votre application. Il est recommandé de toujours intégrer un div spécifique. Plus d'infos ici .

Intégrer dans une div (ou un autre nœud DOM)

Alternativement, en fournissant un élément HTML concret, le code Elm peut être exécuté dans cet élément de page spécifique:

```
<!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>
```

Intégrer en tant qu'agent Web (pas d'interface utilisateur)

Le code Elm peut également être démarré en tant que travailleur et communiquer via les ports :

```
<!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

Une bonne manière d'apprendre à connaître Elm est d'essayer d'écrire certaines expressions dans REPL (Read-Eval-Print Loop). Ouvrez une console dans votre dossier `elm-app` (que vous avez créé lors de la phase d' initialisation et de génération) et procédez comme suit:

```
$ 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` est en fait un outil très puissant. Disons que vous créez un fichier `Test.elm` dans votre dossier `elm-app` avec le code suivant:

```
module Test exposing (..)


a = 1


b = "Hello"
```

Maintenant, vous revenez à votre REPL (qui est resté ouvert) et tapez:

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

Encore plus impressionnant, si vous ajoutez une nouvelle définition à votre fichier `Test.elm` , par exemple

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

Sauvegardez votre fichier, revenez à votre REPL, et sans importer à nouveau `Test` , la nouvelle définition est disponible immédiatement:

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

C'est très pratique lorsque vous voulez écrire des expressions couvrant plusieurs lignes. Il est également très utile de tester rapidement les fonctions que vous venez de définir. Ajoutez ce qui suit à votre fichier:

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

Enregistrez et revenez au REPL:

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

Chaque fois que vous modifiez et enregistrez un fichier que vous avez importé, et que vous revenez à REPL et essayez de faire quoi que ce soit, le fichier complet est recompilé. Par conséquent, il vous indiquera toute erreur dans votre code. Ajoute ça:

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

Essayez cela:

```
> 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.


>
```

Pour conclure cette introduction à la REPL, ajoutons `elm-repl` connaît également les packages que vous avez installés avec l' `elm package install` . Par exemple:

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

Serveur de construction local (réacteur Elm)

Elm Reactor est l'outil essentiel pour le prototypage de votre application.

Veuillez noter que vous ne pourrez pas compiler `Main.elm` avec Elm Reactor si vous utilisez Http.App.programWithFlags ou Ports

Lancer elm-réacteur dans un répertoire de projets démarrera un serveur Web avec un explorateur de projet, ce qui vous permettra de compiler chaque composant séparé.

Toutes les modifications apportées à votre code sont mises à jour lorsque vous rechargez la page.

```
$ 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
```

Lire Démarrer avec Elm Language en ligne: https://riptutorial.com/fr/elm/topic/1011/demarrer-avec-elm-language

Chapitre 2: Abonnements

Remarques

Les abonnements sont des moyens d'écouter les entrées. Les ports entrants , les événements clavier ou souris, les messages WebSocket, la géolocalisation et les changements de visibilité de la page peuvent tous servir d'entrées.

Examples

Abonnement de base à l'événement Time.every avec 'unsubscribe'

0.18.0

Le modèle est transmis aux abonnements, ce qui signifie que chaque changement d'état peut modifier les abonnements.

```
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" ) ]
        ]
```

Lire Abonnements en ligne: https://riptutorial.com/fr/elm/topic/4279/abonnements

Chapitre 3: Collecte de données: tuples, enregistrements et dictionnaires

Examples

Tuples

Les tuples sont des listes ordonnées de valeurs de tout type.

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

Il est impossible de modifier la structure d'un tuple ou de mettre à jour la valeur.

Les tuples dans Elm sont considérés comme un type de données primitif, ce qui signifie que vous n'avez pas besoin d'importer de modules pour utiliser Tuples.

Accéder aux valeurs

Le module Basics a deux fonctions d'aide pour accéder aux valeurs d'un Tuple d'une longueur de deux `( a, b )` sans utiliser de correspondance de motif:

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

Les valeurs d'accès des tuples de plus grande longueur se font par correspondance de motif.

Correspondance de motif

Les tuples sont extrêmement utiles en combinaison avec la correspondance de motif:

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

Remarques sur les tuples

Les tuples contiennent moins de 7 valeurs de type de données `comparable`

Dictionnaires

Les dictionnaires sont implémentés dans une bibliothèque principale de Dict .

Un dictionnaire mappant des clés uniques sur des valeurs. Les clés peuvent être de type comparable. Cela inclut Int, Float, Time, Char, String et des tuples ou des listes de types comparables.

Les opérations d'insertion, de suppression et de requête prennent toutes l'heure O (log n).

Contrairement aux tuples et aux enregistrements, les dictionnaires peuvent modifier leur structure. En d'autres termes, il est possible d'ajouter et de supprimer des clés.

Il est possible de mettre à jour une valeur par une clé.

Il est également possible d'accéder ou de mettre à jour une valeur à l'aide de clés dynamiques.

Accéder aux valeurs

Vous pouvez extraire une valeur d'un dictionnaire en utilisant une fonction `Dict.get` .

Définition de type de `Dict.get` :

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

Il renverra toujours `Maybe v` , car il est possible d'essayer d'obtenir une valeur par une clé inexistante.

```
import Dict

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

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

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

Mise à jour des valeurs

L'opération de mise à jour sur un dictionnaire est effectuée à l'aide de `Maybe.map` , car la clé demandée peut être absente.

```
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"
```

Des dossiers

Record est un ensemble de paires clé-valeur.

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

Il est impossible d'accéder à une valeur par une clé inexistante.

Il est impossible de modifier dynamiquement la structure de Record.

Les enregistrements vous permettent uniquement de mettre à jour les valeurs par des clés constantes.

Accéder aux valeurs

Il est impossible d'accéder aux valeurs à l'aide d'une clé dynamique pour éviter d'éventuelles erreurs d'exécution:

```
isMorningKeyName =
    "isMorning "


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

La syntaxe alternative pour accéder à la valeur vous permet d'extraire la valeur, tout en parcourant l'enregistrement:

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

Extension de types

Parfois, vous souhaitez que la signature d'un paramètre contraigne les types d'enregistrement que vous transmettez à des fonctions. L'extension des types d'enregistrement rend l'idée de supertypes inutile. L'exemple suivant montre comment ce concept peut être implémenté:

```
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
```

Nous pourrions même aller plus loin dans les enregistrements et faire quelque chose comme:

```
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
  }
```

Nous avons maintenant des moyens de faire passer ces types partiels dans des fonctions:

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

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

Mise à jour des valeurs

Elm a une syntaxe spéciale pour les mises à jour des enregistrements:

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

```
    }


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

Lire Collecte de données: tuples, enregistrements et dictionnaires en ligne: https://riptutorial.com/fr/elm/topic/2166/collecte-de-donnees--tuples--enregistrements-et-dictionnaires

Chapitre 4: Correspondance de motif

Examples

Arguments de fonction

```
type Dog = Dog String

dogName1 dog =
    case dog of
      Dog name ->
         name

dogName2 (Dog name) ->
     name
```

`dogName1` et `dogName2` sont équivalents. Notez que cela ne fonctionne que pour les ADT qui ont un seul constructeur.

```
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"
```

Ici, nous déconstruisons un enregistrement et obtenons également une référence à l'enregistrement non reconstruit.

Argument déconstruit de type unique

```
type ProjectIdType = ProjectId String

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

Lire Correspondance de motif en ligne: https://riptutorial.com/fr/elm/topic/7168/correspondance-de-motif

Chapitre 5: Création de fonctions de mise à jour complexes avec ccapndave / elm-update-extra

Introduction

ccapndave / elm-update-extra est un package fantastique qui vous aide à gérer des fonctions de mise à jour plus complexes et peut être très utile.

Examples

Message qui appelle une liste de messages

En utilisant la fonction de `sequence` , vous pouvez facilement décrire un message qui appelle une liste d'autres messages. C'est utile lorsque vous traitez la sémantique de vos messages.

Exemple 1: vous créez un moteur de jeu et vous devez actualiser l'écran à chaque image.

```
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]
```

Enchaîner des messages avec et puis

La fonction `andThen` permet de mettre à jour la composition de l'appel. Peut être utilisé avec

l'opérateur de pipeline (`|>`) pour enchaîner les mises à jour.

Exemple: Vous créez un éditeur de document et vous souhaitez que chaque message de modification que vous envoyez à votre document, vous l'enregistrez également:

```
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 vous importez également `Update.Extra.Infix exposing (..)` vous pourrez peut-être utiliser l'opérateur infixe:

```
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
```

Lire Création de fonctions de mise à jour complexes avec ccapndave / elm-update-extra en ligne: https://riptutorial.com/fr/elm/topic/9737/creation-de-fonctions-de-mise-a-jour-complexes-avec-ccapndave---elm-update-extra

Chapitre 6: Décodeurs JSON personnalisés

Introduction

Comment utiliser Json.Decode pour créer des décodeurs personnalisés, par exemple, décodage en types d'union et types de données définis par l'utilisateur

Examples

Décodage en type d'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
```

Lire Décodeurs JSON personnalisés en ligne: https://riptutorial.com/fr/elm/topic/9927/decodeurs-json-personnalises

Chapitre 7: Fonctions et application partielle

Syntaxe

- - définir une fonction sans argument revient à définir simplement une valeur
  language = "Elm"
- - appeler une fonction sans argument en indiquant son nom
  la langue
- - les paramètres sont séparés par des espaces et suivent le nom de la fonction
  ajouter xy = x + y
- - appeler une fonction de la même manière
  ajouter 5 2
- - appliquer partiellement une fonction en ne fournissant que certains de ses paramètres
  incrémenter = ajouter 1
- - utilisez l'opérateur |> pour passer l'expression de gauche à la fonction de droite
  dix = 9 |> incrément
- - le <| l'opérateur passe l'expression à droite à la fonction de gauche
  incrémenter <| ajouter 5 4
- - enchaîner / composer deux fonctions avec l'opérateur >>
  backwardsYell = String.reverse >> String.toUpper
- - le << fonctionne de la même manière en sens inverse
  backwardsYell = String.toUpper << String.reverse
- - une fonction avec un nom non alphanumérique entre parenthèses crée un nouvel opérateur
  (#) xy = x * y
  dix = 5 # 2
- - tout opérateur infixe devient une fonction normale lorsque vous l'enveloppez entre parenthèses
  dix = (+) 5 5
- - Les annotations de type facultatives apparaissent au-dessus des déclarations de fonction
  isTen: Int -> Bool
  isTen n = si n == 10 alors vrai sinon faux

Examples

Vue d'ensemble

La syntaxe de l'application de fonction dans Elm n'utilise pas de parenthèse ni de virgule et est plutôt sensible aux espaces.

Pour définir une fonction, spécifiez son nom `multiplyByTwo` et les arguments `x` , toute opération après un signe égal `=` est ce qui a été renvoyé par une fonction.

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

Pour appeler une fonction, spécifiez son nom et ses arguments:

```
multiplyByTwo 2  -- 4
```

Notez que la syntaxe comme `multiplyByTwo(2)` n'est pas nécessaire (même si le compilateur ne se plaint pas). Les parenthèses servent uniquement à résoudre la priorité:

```
> 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
```

Expressions lambda

Elm a une syntaxe spéciale pour les expressions lambda ou les fonctions anonymes:

```
\arguments -> returnedValue
```

Par exemple, comme vu dans `List.filter` :

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

Plus loin à la profondeur, une barre oblique inversée, `\` , est utilisée pour marquer le début de l'expression lambda, et la flèche, `->` , permet de délimiter les arguments du corps de la fonction. S'il y a plus d'arguments, ils sont séparés par un espace:

```
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

Il est possible de définir des variables locales dans une fonction pour

- réduire la répétition du code
- donner un nom aux sous-expressions

- réduire la quantité d'arguments passés.

La construction de ceci est `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!
```

L'ordre des définitions dans la première partie de `let` n'a pas d'importance!

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

        y =
            100
    in
        x + y

> outOfOrder
201 : number
```

Application partielle

Une application partielle consiste à appeler une fonction avec moins d'arguments et à enregistrer le résultat sous une autre fonction (qui attend le reste des arguments).

```
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
```

En tant que sidenote académique, Elm peut le faire en raison du curry derrière les coulisses.

Évaluation stricte et différée

Dans elm, la valeur d'une fonction est calculée lorsque le dernier argument est appliqué. Dans l'exemple ci-dessous, le diagnostic du `log` sera imprimé lorsque f est appelé avec 3 arguments ou qu'une forme curry de f est appliquée avec le dernier argument.

```
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
```

Parfois, vous voudrez empêcher une fonction d'être appliquée immédiatement. Une utilisation typique dans elm est `Lazy.lazy` qui fournit une abstraction pour contrôler quand les fonctions sont appliquées.

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

Les calculs paresseux prennent une fonction d'un argument de type `()` ou d' `Unit` . Le type d'unité est classiquement le type d'un argument d'espace réservé. Dans une liste d'arguments, l'argument correspondant est spécifié sous la forme _ , indiquant que la valeur n'est pas utilisée. La valeur unitaire dans elm est spécifiée par le symbole spécial `()` qui peut représenter conceptuellement un tuple vide ou un trou. Il ressemble à la liste d'arguments vide en C, Javascript et autres langages qui utilisent des parenthèses pour les appels de fonctions, mais c'est une valeur ordinaire.

Dans notre exemple, `f` peut être protégé contre une évaluation immédiate avec 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
```

L'évaluation de la fonction est retardée à chaque fois qu'une fonction est partiellement appliquée.

```
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 a une fonction `always` active, qui ne peut pas être utilisée pour retarder l'évaluation. Parce que l' orme évalue tous les arguments de la fonction , peu importe si et lorsque le résultat de l'application de fonction est utilisée, enroulant une application de fonction `always` ne causera pas un retard, parce que `f` est pleinement appliquée en tant que paramètre `always` .

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

Opérateurs Infix et notation infixe

Elm permet la définition d'opérateurs d'infix personnalisés.

Les opérateurs Infix sont définis à l'aide de parenthèses autour du nom d'une fonction.

Considérons cet exemple d'opérateur infixe pour construction Tuples `1 => True -- (1, True)` :

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

La plupart des fonctions dans Elm sont définies dans la notation de préfixe.

Appliquez une fonction en utilisant la notation infixe en spécifiant le premier argument avant le nom de la fonction entouré d'un caractère 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]
```

Lire Fonctions et application partielle en ligne: https://riptutorial.com/fr/elm/topic/2051/fonctions-et-application-partielle

Chapitre 8: Intégration Backend

Examples

Elm Basic Http.post demande json sur le serveur express node.js

Serveur dynamique en direct qui renvoie une erreur lorsque la chaîne d'entrée dépasse 10 caractères.

Serveur:

```
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');
});
```

Client:

```
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
```

Lire Intégration Backend en ligne: https://riptutorial.com/fr/elm/topic/8087/integration-backend

Chapitre 9: Json.Decode

Remarques

`Json.Decode` expose deux fonctions pour décoder une charge utile, la première est `decodeValue` qui essaie de décoder une `Json.Encode.Value` , la seconde est `decodeString` qui tente de décoder une chaîne JSON. Les deux fonctions prennent 2 paramètres, un décodeur et une `Json.Encode.Value` ou Json.

Examples

Décoder une liste

L'exemple suivant peut être testé sur 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
```

Pré-décoder un champ et décoder le reste en fonction de cette valeur décodée

Les exemples suivants peuvent être testés sur 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
```

Décryptage de JSON de Rust enum

Ceci est utile si vous utilisez de la rouille dans le backend et de l'orme sur le front

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

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

Le Json encodé de rouille sera:

```
c1:
    {"variant": "Message",
     "fields": ["hi"]
```

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

Le décodeur en orme

```
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"
```

Utilisation: les données sont demandées à http rest api et le décodage de la charge utile sera

```
    Http.fromJson complexDecoder payload
```

Le décodage de la chaîne sera

```
    Decode.decodeString complexDecoder payload
```

Décoder une liste d'enregistrements

Le code suivant peut être trouvé dans une démo ici: 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
```

Décoder une date

Dans le cas où vous avez json avec une chaîne de date ISO comme ceci

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

Vous pouvez le mapper sur elm `Date` type:

```
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
```

Décoder une liste d'objets contenant des listes d'objets

Voir Ellie pour un exemple de travail. Cet exemple utilise le module NoRedInk / elm-decode-pipeline .

Étant donné une liste d'objets JSON, qui contiennent eux-mêmes des listes d'objets 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 chaîne ci-dessus se trouve dans la chaîne de `payload` , elle peut être décodée en utilisant les éléments suivants:

```
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
```

Lire Json.Decode en ligne: https://riptutorial.com/fr/elm/topic/2849/json-decode

Chapitre 10: L'architecture des ormes

Introduction

La méthode recommandée pour structurer vos applications s'appelle «l'architecture Elm».

Le programme le plus simple est constitué d'un `model` enregistrement stockant toutes les données qui pourraient être mises à jour, un type d'union `Msg` qui définit les moyens de votre programme met à jour ces données, une fonction `update` à `Msg view update` qui prend le modèle et un `Msg` et retourne un nouveau modèle, et une fonction `view` qui prend un modèle et renvoie le code HTML que votre page affichera. Chaque fois qu'une fonction retourne un `Msg` , le moteur d'exécution Elm l'utilise pour mettre à jour la page.

Examples

Programme débutant

Html a `beginnerProgram` principalement à des fins d'apprentissage.

`beginnerProgram` n'est pas capable de gérer des abonnements ou des commandes en cours d'exécution.

Il est uniquement capable de gérer les entrées utilisateur des événements DOM.

Il ne nécessite qu'une `view` de rendre le `model` et une `update` à `update` fonction pour gérer les changements d'état.

Exemple

Considérons cet exemple minimal de `beginnerProgram` .

Le `model` dans cet exemple consiste en une seule valeur `Int` .

La fonction de `update` n'a qu'une seule branche, qui incrémente l' `Int` , stockée dans le `model` .

La `view` le modèle et attache un clic sur l'événement DOM.

Voir comment construire l'exemple dans Initialize et build

```
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)) ]
```

Programme

`program` est un bon choix, lorsque votre application ne nécessite aucune donnée externe pour l'initialisation.

Il est capable de gérer les abonnements et les commandes, ce qui offre beaucoup plus d'opportunités pour gérer les E / S, telles que la communication HTTP ou l'interopérabilité avec JavaScript.

L'état initial est requis pour renvoyer les commandes de démarrage avec le modèle.

L'initialisation du `program` nécessitera des `subscriptions` , ainsi que le `model` , la `view` et la `update` .

Voir la définition du type:

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

Exemple

Le moyen le plus simple d'illustrer comment vous pouvez utiliser les abonnements consiste à configurer une simple communication de port avec JavaScript.

Voir comment créer l'exemple dans Initialize and build / Embedding dans 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>
```

Programme avec des drapeaux

`programWithFlags` n'a qu'une seule différence avec le `program` .

Il peut accepter les données lors de l'initialisation à partir de JavaScript:

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

Les données transmises depuis JavaScript s'appellent Flags.

Dans cet exemple, nous transmettons un objet JavaScript à Elm avec des informations utilisateur, il est conseillé de spécifier un alias de type pour les indicateurs.

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

Les indicateurs sont transmis à la fonction `init` , produisant l'état initial:

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

Vous remarquerez peut-être la différence par rapport à la signature de type:

```
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
```

Le code d'initialisation est presque identique, car seule la fonction d' `init` est différente.

```
main =
```

```
    programWithFlags
        { init = init
        , update = update
        , view = view
        , subscriptions = subscriptions
        }
```

Communication parent-enfant à sens unique

L'exemple montre la composition du composant et le message unidirectionnel passant du parent aux enfants.

0.18.0

La composition des composants repose sur le balisage des messages avec `Html.App.map`

0.18.0

En `0.18.0` `HTML.App` été réduit en `HTML`

La composition du composant repose sur le balisage des messages avec `Html.map`

Exemple

Voir comment construire l'exemple dans Initialiser et construire

```
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
```

Balisage de message avec Html.App.map

Les composants définissent leurs propres messages, envoyés après les événements DOM émis, par exemple. `CounterMsg` de la communication parent-enfant

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

La vue de ce composant enverra des messages de type `CounterMsg` , par conséquent la signature de type de vue est `Html CounterMsg` .

Pour pouvoir réutiliser `counterView` dans la vue du composant parent, nous devons transmettre chaque message `CounterMsg` via `Msg` du parent.

Cette technique s'appelle le ***marquage de message*** .

Le composant parent doit définir des messages pour la transmission des messages enfants:

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

`FirstCounterMsg Increment` est un message balisé.

0.18.0

Pour obtenir un `counterView` pour envoyer des messages marqués, nous devons utiliser la fonction `Html.App.map` :

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

0.18.0

Le package `HTML.App` été réduit dans le package `HTML` en `v0.18.0`

Pour qu'un `counterView` envoie des messages marqués, nous devons utiliser la fonction `Html.map` :

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

Cela modifie le type signature `Html CounterMsg -> Html Msg` , il est donc possible d'utiliser le compteur dans la vue parent et de gérer les mises à jour d'état avec la fonction de mise à jour du parent.

Lire L'architecture des ormes en ligne: https://riptutorial.com/fr/elm/topic/3771/l-architecture-des-

ormes

Chapitre 11: Le débogage

Syntaxe

- Debug.log "tag" anyValue

Remarques

`Debug.log` prend deux paramètres, une `String` pour marquer la sortie de débogage dans la console (afin de savoir d'où elle provient / à quoi correspond le message) et une valeur de n'importe quel type. `Debug.log` exécute l'effet secondaire de la journalisation de la balise et de la valeur sur la console JavaScript, puis renvoie la valeur. L'implémentation dans JS pourrait ressembler à:

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

JavaScript a des conversions implicites, donc la `value` ne doit pas être explicitement convertie en `String` pour que le code ci-dessus fonctionne. Toutefois, les types Elm doivent être explicitement convertis en une `String` et le code natif pour `Debug.log` montre en action.

Examples

Enregistrer une valeur sans interrompre les calculs

Le deuxième argument de `Debug.log` est toujours renvoyé, vous pouvez donc écrire du code comme celui-ci et cela fonctionnera *simplement* :

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

Remplacer le `case msg of` par la `case Debug.log "The Message" msg of` entraînera le `case Debug.log "The Message" msg of` actuel de la console à chaque appel de la fonction de mise à jour, mais ne changera rien.

Piping a Debug.log

Au moment de l'exécution, les éléments suivants afficheraient une liste d'url dans votre console et continueraient à calculer

```
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
```

Débogueur itinérant

0,17 0,18,0

Au moment de la rédaction de ce document (juillet 2016), elm-Reactor a été temporairement dépourvu de fonctionnalité de déplacement temporel. Il est possible de l'obtenir en utilisant le `jinjor/elm-time-travel` .

Son utilisation reflète les `Html.App` du `program*` modules `Html.App` ou `Navigation` , par exemple au lieu de:

```
import Html.App

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

tu écrirais:

```
import TimeTravel.Html.App

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

(Bien sûr, après avoir installé le paquet avec `elm-package` .)

L'interface de votre application change en conséquence, voir l'une des démos .

0.18.0

Depuis la version **0.18.0,** vous pouvez simplement compiler votre programme avec l'indicateur `--debug` et obtenir un débogage sans effort supplémentaire.

Debug.Crash

```
case thing of
    Cat ->
```

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

Vous pouvez utiliser `Debug.crash` lorsque vous souhaitez que le programme échoue, généralement utilisé lorsque vous êtes en train d'implémenter une expression de `case` . Il n'est *pas* recommandé d'utiliser `Debug.crash` au lieu d'utiliser un type `Maybe` ou `Result` pour des entrées inattendues, mais généralement uniquement au cours du développement (autrement dit, vous ne publiez généralement pas le code Elm qui utilise `Debug.crash`).

`Debug.crash` prend une valeur de `String` , le message d'erreur à afficher lors d'une panne. Notez que Elm affichera également le nom du module et la ligne du plantage, et si le plantage se trouve dans une expression de `case` , il indiquera la valeur du `case` .

Lire Le débogage en ligne: https://riptutorial.com/fr/elm/topic/2845/le-debogage

Chapitre 12: Listes et itération

Remarques

La `List` (liste liée) brille en **accès séquentiel** :

- accéder au premier élément
- précédant le début de la liste
- effacer de la liste

D'un autre côté, ce n'est pas idéal pour **un accès aléatoire** (c.-à-d. Obtenir le nième élément) et la **traversée dans l'ordre inverse** , et vous pourriez avoir plus de chance (et de performances) avec la structure de données `Array` .

Examples

Créer une liste par plage

0.18.0

Avant **0.18.0,** vous pouvez créer des plages comme ceci:

```
> 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

Dans **0.18.0** La syntaxe `[1..5]` a été supprimée .

```
> 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
```

Les plages créées par cette syntaxe sont toujours **inclusives** et le **pas** est toujours **1** .

Créer une liste

```
> 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
>
```

Sous le capot, `List` (liste chaînée) est construit par la fonction `::` (appelée "cons"), qui prend deux arguments: un élément, appelé tête, et une liste (éventuellement vide) à laquelle la tête est ajoutée.

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

`List` ne peut prendre que des valeurs d'un type, donc quelque chose comme `[1,"abc"]` n'est pas possible. Si vous en avez besoin, utilisez des tuples.

```
> 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.


>
```

Obtenir des éléments

```
> 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)
```

Cet encapsulation dans le type `Maybe` se produit à cause du scénario suivant:

Que devrait retourner `List.head` pour une liste vide? (Rappelez-vous, Elm n'a pas d'exceptions ni de null.)

```
> 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)
```

Transformer chaque élément d'une liste

`List.map : (a -> b) -> List a -> List b` est une fonction d'ordre supérieur qui applique une fonction à un paramètre à chaque élément d'une liste, renvoyant une nouvelle liste avec les valeurs modifiées.

```
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 vous avez besoin de connaître l'index des éléments, vous pouvez utiliser `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"]
```

Filtrer une liste

`List.filter : (a -> Bool) -> List a -> List a` est une fonction d'ordre supérieur qui prend une fonction à un paramètre de n'importe quelle valeur à un booléen et applique cette fonction à chaque élément d'une liste donnée, ne garder que les éléments pour lesquels la fonction renvoie `True` . La fonction que prend `List.filter` comme premier paramètre est souvent appelée prédicat .

```
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
```

Évaluez ceci dans `elm-repl` :

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

Correspondance de motif sur une liste

Nous pouvons faire correspondre les listes comme n'importe quel autre type de données, bien qu'elles soient quelque peu uniques, dans la mesure où le constructeur de la construction des listes est la fonction infixe `::` . (Voir l'exemple Création d'une liste pour en savoir plus sur le fonctionnement)

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

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

Nous pouvons faire correspondre autant d'éléments que nous voulons dans la liste:

```
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
```

Obtenir le nième élément de la liste

`List` ne supporte pas "random access", ce qui signifie qu'il faut plus de travail pour obtenir, disons, le cinquième élément de la liste que le premier élément, et par conséquent, il n'y a pas de fonction `List.get nth list` . Il faut aller depuis le début (`1 -> 2 -> 3 -> 4 -> 5`).

Si vous avez besoin d'un accès aléatoire, vous obtiendrez de meilleurs résultats (et performances) avec des structures de données à accès aléatoire, comme `Array` , où prendre le premier élément nécessite autant de travail que, par exemple, le 1000ème. (complexité O (1)).

Néanmoins, il est possible **(mais découragé)** d'obtenir le nième élément:

```
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
```

Encore une fois, cela prend beaucoup plus de travail, plus le `nth` argument est grand.

Réduire une liste à une valeur unique

Dans Elm, les fonctions de réduction sont appelées "folds", et il existe deux méthodes standard pour "plier" les valeurs: de gauche à `foldl` , et de droite à droite, `foldr` .

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

Les arguments de `foldl` et `foldr` sont les suivants:

- **fonction de réduction** : `newValue -> accumulator -> accumulator`
- valeur de départ de l' **accumulateur**
- **liste** pour réduire

Un autre exemple avec une fonction personnalisée:

```
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
```

Dans le premier exemple ci-dessus, le programme se présente comme suit:

```
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
```

Dans le cas d'une fonction commutative comme `(+)` il n'y a pas vraiment de différence.

Mais voyez ce qui se passe avec `(::)` :

```
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]
```

Créer une liste en répétant une valeur

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

Vous pouvez donner à `List.repeat` n'importe quelle valeur:

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

Trier une liste

Par défaut, `List.sort` trie par ordre croissant.

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

`List.sort` besoin que les éléments de la liste soient `comparable` . Cela signifie: `String` , `Char` , `number` (`Int` et `Float`), `List` de `comparable` ou tuple 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)
```

Vous ne pouvez pas trier les listes de `Bool` ou d'objets avec `List.sort` . Pour cela, voir Trier une liste avec un comparateur personnalisé.

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

Tri d'une liste avec un comparateur personnalisé

`List.sortWith` vous permet de trier les listes avec des données de n'importe quelle forme - vous leur fournissez une fonction de comparaison.

```
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
```

Inverser une liste

Note: ceci n'est pas très efficace en raison de la nature de la `List` (voir Remarques ci-dessous). Il

vaudra mieux **construire la liste de la bonne manière depuis le début** que de la construire et de l'inverser.

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

Tri d'une liste par ordre décroissant

Par défaut, `List.sort` trie par ordre croissant, avec la fonction de `compare` .

Il existe deux manières de trier par ordre décroissant: une efficace et une inefficace.

1. **Le moyen efficace** : `List.sortWith` et une fonction de comparaison décroissante.

```
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. La manière inefficace **(déconseillée!)** : `List.sort` puis `List.reverse` .

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

Trier une liste par une valeur dérivée

`List.sortBy` permet d'utiliser une fonction sur les éléments et d'utiliser son résultat pour la comparaison.

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

Il fonctionne aussi très bien avec les accesseurs de disques:

```
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}
```

Lire Listes et itération en ligne: https://riptutorial.com/fr/elm/topic/1635/listes-et-iteration

Chapitre 13: Ports (interop)

Syntaxe

- Elm (reception): port functionName: (valeur -> msg) -> Sub msg
- JS (envoi): app.ports.functionName.send (valeur)
- Elm (Envoi): Port functionName: args -> Cmd msg
- JS (réception): app.ports.functionName.subscribe (fonction (args) {...});

Remarques

Consultez http://guide.elm-lang.org/interop/javascript.html dans *The Elm Guide* pour vous aider à comprendre ces exemples.

Examples

Vue d'ensemble

Un module utilisant les ports doit avoir un mot-clé de `port` dans sa définition de module.

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

Il est impossible d'utiliser les ports avec `Html.App.beginnerProgram` , car il ne permet pas d'utiliser les abonnements ou les commandes.

Les ports sont intégrés pour mettre à jour la boucle de `Html.App.program` ou `Html.App.programWithFlags` .

Remarque

`program` et `programWithFlags` dans elm 0.18 sont à l'intérieur du package `Html` au lieu de `Html.App` .

Sortant

Les ports sortants sont utilisés comme commandes, que vous renvoyez depuis votre fonction de `update` .

Côté orme

Définir le port sortant:

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

Dans cet exemple, nous envoyons un Tuple vide pour déclencher un abonnement côté JavaScript.

Pour ce faire, nous devons appliquer `output` fonction de `output` avec un argument Tuple vide, afin d'obtenir une commande pour l'envoi des données sortantes d'Elm.

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

Côté JavaScript

Initialiser l'application:

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

Abonnez-vous à un port avec un nom correspondant:

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

Remarque

A partir de `0.17.0` , le message sortant immédiat à JavaScript de votre état `initial` n'aura aucun effet.

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

Voir la solution de contournement dans l'exemple ci-dessous.

Entrants

Les données entrantes de JavaScript passent par des abonnements.

Côté orme

Tout d'abord, nous devons définir un port entrant en utilisant la syntaxe suivante:

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

Nous pouvons utiliser `Sub.batch` si nous avons plusieurs abonnements, cet exemple ne contiendra qu'un seul abonnement au `input port`

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

Ensuite, vous devez transmettre les `subscriptions` à votre programme `Html.program` :

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

Côté JavaScript

Initialiser l'application:

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

Envoyer le message à Elm:

```
var counter = 0;

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

Remarque

S'il vous plaît noter que à partir de `0.17.0` le `app.ports.input.send(counter);` immédiat `app.ports.input.send(counter);` après l'application l'initialisation n'aura aucun effet!

Transmettez toutes les données requises pour le démarrage en tant que `Html.programWithFlags` aide de `Html.programWithFlags`

Message sortant immédiat au démarrage en 0.17.0

Pour envoyer un message immédiat contenant des données à JavaScript, vous devez déclencher une action à partir de votre `init` .

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

```
    ( Model 0, send SendOutgoing )


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

Commencer

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
```

Installez le paquet `elm-lang/html` si vous ne l'avez pas encore fait par `elm-package install elm-lang/html --yes` .

Compilez ce code en utilisant `elm-make Main.elm --yes --output elm.js` pour que le fichier HTML le trouve.

Si tout se passe bien, vous devriez pouvoir ouvrir le fichier `index.html` avec le texte "Aucun message" affiché. Après trois secondes, le JS envoie un message, Elm le reçoit, modifie son modèle, envoie une réponse, JS le reçoit et ouvre une alerte.

Lire Ports (interop) en ligne: https://riptutorial.com/fr/elm/topic/2200/ports--interop-

Chapitre 14: Types, variables de type et constructeurs de types

Remarques

S'il vous plaît jouez avec ces concepts vous-même pour vraiment les maîtriser! L' `elm-repl` (voir l' introduction du REPL) est probablement un bon endroit pour jouer avec le code ci-dessus. Vous pouvez également jouer avec `elm-repl` ligne .

Examples

Types de données comparables

Les types comparables sont des types primitifs pouvant être comparés à l'aide d'opérateurs de comparaison du module Basics , tels que: `(<)` , `(>)` , `(<=)` , `(>=)` , `max` , `min` , `compare`

Les types comparables dans Elm sont `Int` , `Float` , `Time` , `Char` , `String` et des tuples ou des listes de types comparables.

Dans la documentation ou les définitions de type, elles sont appelées variables de type spécial `comparable` , par exemple. voir la définition de type pour la fonction `Basics.max` :

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

Signatures de type

Dans Elm, les valeurs sont déclarées en écrivant un nom, un signe égal, puis la valeur réelle:

```
someValue = 42
```

Les fonctions sont également des valeurs, en plus de prendre une valeur ou des valeurs en argument. Ils sont généralement écrits comme suit:

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

Chaque valeur dans Elm a un type. Les types de valeurs ci-dessus seront *déduits* par le compilateur en fonction de leur utilisation. Toutefois, il est recommandé de toujours déclarer explicitement le type de toute valeur de niveau supérieur et, pour ce faire, écrivez une *signature de type* comme suit:

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

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

Comme on peut le voir, `42` peut être définie comme un `Int` ou un `Float` . Cela a un sens intuitif, mais consultez **Variables de type** pour plus d'informations.

Les signatures de type sont particulièrement utiles lorsqu'elles sont utilisées avec des fonctions. Voici la fonction de doublage d'avant:

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

Cette fois, la signature a un `->` , une flèche, et nous prononçons la signature comme "int to int", ou "prend un entier et renvoie un entier". `->` indique qu'en `double` une valeur `Int` comme argument, `double` retournera un `Int` . Par conséquent, il faut un entier pour un entier:

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

> double 3
6 : Int
```

Types de base

Dans `elm-repl` , tapez un morceau de code pour obtenir sa valeur et son type inféré. Essayez ce qui suit pour en savoir plus sur les différents types existants:

```
> 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 }

> ()
() : ()
```

`{}` est le type d'enregistrement vide et `()` est le type vide de Tuple. Ce dernier est souvent utilisé à des fins d'évaluation paresseuse. Voir l'exemple correspondant dans Fonctions et application partielle .

Notez comment le `number` semble non capitalisé. Cela indique qu'il s'agit d'une **variable de type** , et de plus, le `number` mot particulier fait référence à une **variable de type spécial** qui peut être un `Int` ou un `Float` (voir les sections correspondantes pour plus d'informations). Les types, cependant, sont toujours en majuscules, tels que `Char` , `Float` , `List String` , et cetera.

Variables de type

Les variables de type sont des noms non capitalisés dans les signatures de type. Contrairement à leurs homologues en majuscules, tels que `Int` et `String` , ils ne représentent pas un type unique, mais plutôt un type quelconque. Ils sont utilisés pour écrire des fonctions génériques pouvant fonctionner sur *tout* type ou type et sont particulièrement utiles pour écrire des opérations sur des conteneurs tels que `List` ou `Dict` . La fonction `List.reverse` , par exemple, a la signature suivante:

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

Ce qui signifie qu'il peut fonctionner sur une liste de *n'importe quelle valeur de type* , de sorte que `List Int` , `List (List String)` , à la fois ceux-là et tous les autres, peuvent être `reversed` . Par conséquent, `a` est une variable de type qui peut figurer dans n'importe quel type.

La fonction `reverse` aurait pu utiliser *n'importe* quel nom de variable non capitalisé dans sa signature de type, à l'exception de quelques noms de **variables de type spécial** , tels que `number` (voir l'exemple correspondant pour plus d'informations):

```
reverse : List lol -> List lol

reverse : List wakaFlaka -> List wakaFlaka
```

Les noms des variables de type ne sont significatifs que lorsqu'il existe *différentes* variables de type dans une même signature, illustrées par la fonction de `map` sur les listes:

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

`map`

prend une fonction de n'importe quel type `a` à n'importe quel type `b` , avec une liste d'éléments de type `a` , et renvoie une liste d'éléments de type `b` , obtenus en appliquant la fonction donnée à chaque élément de la liste.

Faisons la signature concrète pour mieux voir ceci:

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

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

Comme on peut le voir, à la fois `a = Int` et `b = Int` dans ce cas. Mais si `map` avait une signature de type comme `map : (a -> a) -> List a -> List a` , alors cela *ne* fonctionnerait que sur les fonctions qui fonctionnent sur un seul type, et vous ne pourriez jamais changer la type d'une liste en utilisant la fonction de `map` . Mais comme la signature de type de `map` a plusieurs variables de type différentes, `a` et `b` , nous pouvons utiliser `map` pour changer le type d'une liste:

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

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

Dans ce cas, `a = Int` et `b = Bool` . Par conséquent, pour pouvoir utiliser des fonctions pouvant prendre et renvoyer *des* types *différents* , vous devez utiliser des variables de type différentes.

Alias de type

Parfois, nous voulons donner à un type un nom plus descriptif. Disons que notre application a un type de données représentant les utilisateurs:

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

Et nos fonctions sur les utilisateurs ont des signatures de type dans le sens de:

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

Cela pourrait devenir assez compliqué avec un type d'enregistrement plus important pour un utilisateur, alors utilisons un *alias de type* pour réduire la taille et donner un nom plus significatif à cette structure de données:

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

```
prettyPrintUser : User -> String
```

Les alias de type simplifient la définition et l'utilisation d'un modèle pour une application:

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

Utiliser l' `type alias` littéralement ne fait qu'aliaser un type avec le nom que vous lui donnez. L'utilisation du type de `Model` ci-dessus est identique à l'utilisation de `{ count : Int, lastEditMade : Time }` . Voici un exemple montrant comment les alias ne sont pas différents des types sous-jacents:

```
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 type pour un type d'enregistrement définit une fonction constructeur avec un argument pour chaque champ dans l'ordre de déclaration.

```
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
```

Chaque alias de type d'enregistrement a son propre ordre de champ, même pour un type 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
```

Amélioration de la sécurité des types en utilisant de nouveaux types

Les types d'alias réduisent les problèmes et améliorent la lisibilité, mais ce n'est pas plus sûr que le type alias lui-même. Considérer ce qui suit:

```
type alias Email = String

type alias Name = String

someEmail = "holmes@private.com"

someName = "Benedict"

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

En utilisant le code ci-dessus, nous pouvons écrire `sendEmail someName` , et il compilera, même si cela ne devrait vraiment pas être le cas, car malgré les noms et les emails étant tous deux des `String` , ils sont complètement différents.

Nous pouvons vraiment distinguer une `String` d'une autre `String` au niveau du type en créant un nouveau **type** . Voici un exemple qui réécrit `Email` comme un `type` plutôt qu'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) = ...
```

Notre fonction `isValid` fait quelque chose pour déterminer si une chaîne est une adresse électronique valide. La fonction `create` vérifie si une `String` donnée est un email valide, renvoyant un `Email Maybe` -wrapped pour s'assurer que nous ne renvoyons que des adresses validées. Bien que nous puissions contourner le contrôle de validation en construisant un `Email` directement en écrivant `EmailAddress "somestring"` , si notre déclaration de module n'expose pas le constructeur `EmailAddress` , comme indiqué ici

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

alors aucun autre module n'aura accès au constructeur `EmailAddress` , bien qu'ils puissent toujours utiliser le type `Email` dans les annotations. La **seule** façon de créer un nouvel `Email` dehors de ce module consiste à utiliser la fonction `create` qu'elle fournit et cette fonction garantit qu'elle ne renverra que des adresses e-mail valides. Par conséquent, cette API guide automatiquement l'utilisateur sur le chemin correct via son type de sécurité: `send` ne fonctionne qu'avec les valeurs

construites par `create` , qui effectue une validation et applique le traitement des emails non valides car il retourne un `Maybe Email` - `Maybe Email` .

Si vous souhaitez exporter le constructeur `Email` , vous pouvez écrire

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

Désormais, tout fichier importé `Email` peut également importer son constructeur. Dans ce cas, cela permettrait aux utilisateurs de contourner la validation et d' `send` des e-mails non valides, mais vous ne construisez pas toujours une API comme celle-ci, donc les constructeurs exportateurs peuvent être utiles. Avec un type qui a plusieurs constructeurs, vous pouvez également ne vouloir exporter que certains d'entre eux.

Construire des types

La combinaison de mots-clés `type alias` donne un nouveau nom à un type, mais le mot-clé `type` déclare un nouveau type. Examinons l'un des plus fondamentaux de ces types: `Maybe` - `Maybe`

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

La première chose à noter est que le type `Maybe` est déclaré avec une **variable** de `a` . La deuxième chose à noter est le caractère de pipe, `|` , qui signifie "ou". En d'autres termes, quelque chose de type `Maybe a` est *soit* `Just a` *ou* `Nothing` .

Lorsque vous écrivez le code ci-dessus, `Just` et `Nothing` entrent en ligne de compte en tant que *constructeurs* de *valeurs* , et `Maybe` en tant que *constructeur de type* . Ce sont leurs signatures:

```
Just : a -> Maybe a

Nothing : Maybe a

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

En raison de la *variable de type* `a` , tout type peut être "enveloppé" du type `Maybe` . Donc, `Maybe Int` , `Maybe (List String)` ou `Maybe (Maybe (List Html))` sont tous des types valides. Lors de la déstructuration de toute valeur de `type` avec une expression de `case` , vous devez tenir compte de chaque instanciation possible de ce type. Dans le cas d'une valeur de type `Maybe a` , vous devez tenir compte à la fois de l'affaire `Just a` case et `Nothing` :

```
thing : Maybe Int
thing =
    Just 3

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

```
        Nothing ->
            42

-- blah = 3
```

Essayez d'écrire le code ci-dessus sans la clause `Nothing` dans l'expression de `case` : cela ne compilera pas. C'est ce qui fait du constructeur de type `Maybe` un excellent modèle pour exprimer des valeurs qui peuvent ne pas exister, car il vous oblige à gérer la logique de la valeur `Nothing` .

Le type jamais

Le type `Never` ne peut pas être construit (le module `Basics` n'a pas exporté son **constructeur de valeur** et ne vous a pas donné d'autre fonction qui renvoie `Never` non plus). Il n'y a pas de valeur `never : Never` ou une fonction `createNever : ?? -> Never` .

Cela a ses avantages: vous pouvez encoder dans un système de type une possibilité qui ne peut pas arriver. Cela peut être vu dans des types comme `Task Never Int` qui garantit qu'il réussira avec un `Int` ; ou `Program Never` qui ne prendra aucun paramètre lors de l'initialisation du code Elm à partir de JavaScript.

Variables de type spécial

Elm définit les variables de type spéciales suivantes qui ont une signification particulière pour le compilateur:

- **`comparable`** : composé de `Int` , `Float` , `Char` , `String` et tuples. Cela permet l'utilisation des opérateurs `<` et `>` .

  Exemple: Vous pouvez définir une fonction pour trouver les éléments les plus petits et les plus grands d'une liste (`extent`). Vous pensez quel type de signature écrire. D'une part, vous pouvez écrire `extentInt : List Int -> Maybe (Int, Int)` et `extentChar : List Char -> Maybe (Char, Char)` et un autre pour `Float` et `String` . La mise en œuvre de ces serait la même:

  ```
  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
  ```

  Vous pourriez être tenté d'écrire simplement `extent : List a -> Maybe (a, a)` , mais le compilateur ne vous laissera pas faire, car les fonctions `min` et `max` ne sont pas définies pour ces types (NB: ce ne sont que de simples wrappers) autour de l'opérateur `<` mentionné ci-dessus). Vous pouvez résoudre ce problème en définissant l' `extent : List comparable -> Maybe (comparable, comparable)`

. Cela permet à votre solution d'être *polymorphe* , ce qui signifie simplement qu'elle fonctionnera pour plusieurs types.

- **`number`** : composé de `Int` et `Float` . Permet l'utilisation d'opérateurs arithmétiques sauf division. Vous pouvez ensuite définir par exemple la `sum : List number -> number` et le faire fonctionner à la fois pour les ints et les flottants.

- **`appendable`** : composé de `String` , `List` . Permet l'utilisation de l'opérateur `++` .

- **`compappend`** : Ceci apparaît parfois, mais est un détail d'implémentation du compilateur. Actuellement, cela ne peut pas être utilisé dans vos propres programmes, mais est parfois mentionné.

Notez que dans une annotation de type comme celle-ci: `number -> number -> number` ils se réfèrent tous au même type, donc passer `Int -> Float -> Int` serait une erreur de type. Vous pouvez résoudre ce problème en ajoutant un suffixe au type nom de la variable: `number -> number' -> number''` compilerait alors bien.

Il n'y a pas de nom officiel pour ceux-ci, ils sont parfois appelés:

- Variables de type spécial
- Variables de type typeclass-like
- Pseudo-typeclasses

C'est parce qu'ils fonctionnent comme les classes de type de Haskell, mais sans la possibilité pour l'utilisateur de les définir.

Lire Types, variables de type et constructeurs de types en ligne:
https://riptutorial.com/fr/elm/topic/2648/types--variables-de-type-et-constructeurs-de-types

Crédits

S. No	Chapitres	Contributeurs
1	Démarrer avec Elm Language	2426021684, alejosocorro, AnimiVulpis, Community, Douglas Correa, gabrielperales, gar, halfzebra, Jakub Hampl, jmite, JustGage, lonelyelk, Martin Janiczek, mrkovec, thSoft, Zimm i48
2	Abonnements	lonelyelk, mrkovec, Tosh
3	Collecte de données: tuples, enregistrements et dictionnaires	halfzebra, Martin Janiczek, Mr. Baudin
4	Correspondance de motif	Gerald Kaszuba, Jakub Hampl, Tosh
5	Création de fonctions de mise à jour complexes avec ccapndave / elm-update-extra	Mateus Felipe
6	Décodeurs JSON personnalisés	Khaled Jouda
7	Fonctions et application partielle	Art Yerkes, halfzebra, lonelyelk, Martin Janiczek, Nicholas Montaño, Ryan Plant, Will White
8	Intégration Backend	lonelyelk
9	Json.Decode	ivanceras, Jonathan de M., lonelyelk, Matthew Rankin
10	L'architecture des ormes	AnimiVulpis, halfzebra, mrkovec, Ryan Plant, vlad_o, Zimm i48
11	Le débogage	AnimiVulpis, bdukes, Jonathan de M., Martin Janiczek, Nicholas Montaño
12	Listes et itération	2426021684, AnimiVulpis, jmite, lonelyelk, Martin Janiczek, Nicholas Montaño, Zimm i48
13	Ports (interop)	Adam Bowen, gabrielperales, halfzebra, Martin Janiczek, Nicholas Montaño
14	Types, variables de	Art Yerkes, bright-star, halfzebra, Jakub Hampl, Joseph

	type et constructeurs de types	Weissman, Martin Janiczek, Nicholas Montaño, Zimm i48