Instrukcja laboratoryjna ELM

(M.Jurkiewicz, K.Manuszewski)

Poniższa instrukcja laboratoryjna ma na celu krok po kroku przeprowadzić czytelnika przez proces tworzenia prostej strony www w Elmie, zaczynając od przygotowania środowiska deweloperskiego, przez podstawy języka wraz z ćwiczeniami pozwalającymi na lepsze zrozumienie składni, aż po stworzenie prostej listy elementów np. galerii obrazów.

I. Przygotowanie środowiska

Pierwszą rzeczą, którą należy się zająć przed rozpoczęciem nowego projektu jest przygotowanie odpowiedniego środowiska deweloperskiego. Należy upewnić się, że wszystkie narzędzia potrzebne do wykonania pracy są zainstalowane i prawidłowo skonfigurowane. W przypadku pracy z Elm'em zalecane będzie korzystanie przede wszystkim z platformy dostarczanej przez autora, edytora tekstu wspierającego podświetlanie składni, a także innych narzędzi wspomagających proces tworzenia oprogramowania z użyciem tej technologii.

Platforma Elm

Najważniejszą rzeczą, jaka będzie potrzebna podczas pracy z Elm'em, będzie platforma języka zawierająca m.in. takie narzędzia jak kompilator oraz menadżer bibliotek. Poniżej przedstawiam instrukcję instalacji tej platformy na systemach operacyjnych Linux i Windows. Po przejściu tych kroków, w wierszu poleceń należy wykonać instrukcję `elm`. Jeśli wszystko zostało prawidłowo za- instalowane i skonfigurowane, na ekranie powinien pojawić się widok podobny do przedstawionego na nast. rysunku.

```
→ ~ elm
Hi, thank you for trying out Elm 0.19.1. I hope you like it!

-------------------------------------------------------------------------------
I highly recommend working through <https://guide.elm-lang.org> to get started.
It teaches many important concepts, including how to use `elm` in the terminal.
-------------------------------------------------------------------------------

The most common commands are:

    elm repl
        Open up an interactive programming session. Type in Elm expressions like
        (2 + 2) or (String.length "test") and see if they equal four!

    elm init
        Start an Elm project. It creates a starter elm.json file and provides a
        link explaining what to do from there.

    elm reactor
        Compile code with a click. It opens a file viewer in your browser, and
        when you click on an Elm file, it compiles and you see the result.

There are a bunch of other commands as well though. Here is a full list:

    elm repl    --help
    elm init    --help
    elm reactor --help
    elm make    --help
    elm install --help
    elm bump    --help
    elm diff    --help
    elm publish --help

Adding the --help flag gives a bunch of additional details about each one.

Be sure to ask on the Elm slack if you run into trouble! Folks are friendly and
happy to help out. They hang out there because it is fun, so be kind to get the
best results!
```

Rysune: Wyjście instrukcji `elm`

Linux

Najprostszym sposobem instalacji platformy Elm na systemie operacyjnym Linux jest wykorzystanie narzędzia `npm` – powszechnie używanego menadżera pakietów służącego do zarządzania warstwą frontendową aplikacji internetowych. Aby zainstalować `npm` należy skorzystać z systemowego menadżera pakietów.

```
$ sudo apt update
$ sudo apt install npm
```

Do instalacji samego ELMa posłuży polecenie:

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

Zgodnie z dokumentacją `npm` [15], flaga `-g` oznacza, że pakiet zostanie zainstalowany globalnie, dzięki czemu będzie dostępny z każdego miejsca z systemu. Aby sprawdzić, czy rzeczywiście tak się stało, wystarczy w wierszu poleceń uruchomić komendę `elm`.

Windows

Osoby korzystające z systemu operacyjnego Windows mogą skorzystać z npm, tak jak to było opisane w powyższej sekcji dotyczącej Linuxa lub posłużyć się dedykowanym instalatorem (zalecane z uwagi na pot. mniejsze problemy z npm wersjonowaniem bibliotek itd.) Elm'a [https://guide.elm-lang.org/install/elm.html] na system Windows. W tym drugim przypadku wystarczy przejść przez wszystkie kroki zostawiając opcje domyślne i w rezultacie Elm zostanie pomyślnie zainstalowany i będzie gotowy do użytkowania. W celu sprawdzenia czy faktycznie tak się stało, należy uruchomić wiersz poleceń oraz wykonać instrukcję `elm`.

Edytor

Ważnym elementem tworzenia oprogramowania jest wyposażenie się w odpowiedni edytor tek- stowy, który jest w stanie podświetlać składnię języka, z którego aktualnie korzystamy. Żeby osią- gnąć ten cel, w przypadku Elm'a potrzebna będzie instalacja dodatkowej wtyczki do jednego z następujących edytorów:

- Atom
- Emacs
- IntelliJ
- Light Table
- Sublime Text
- Vim
- VS Code

Powyższa lista zawiera odnośniki do wspomnianych wtyczek dla danego edytora, wystarczy kliknąć nazwę swojego ulubionego edytora i pobrać odpowiedni dodatek. Na potrzeby niniejszej instrukcji przedstawię proces instalacji wtyczki dla edytora Visual Studio Code, ponieważ jest to jedno z najbardziej powszechnie używanych narzędzi do pracy z kodem źródłowym.

W tym celu należy otworzyć edytor VS Code, otworzyć konsolę *Quick Open* przy pomocy skrótu

Ctrl+P i wpisać następującą komendę:

```
ext install Elmtooling.elm-ls-vscode
```

Wtyczka powinna zostać automatycznie zainstalowana i dalsza konfiguracja nie jest konieczna, edytor powinien być gotowy do pracy. Po instalacji można przejść do kolejnego kroku.

Tworzenie projektu

Elm jest dostarczany wraz z zestawem bardzo przydatnych narzędzi. Jednym z nich jest `elm init`, które posłuży nam do stworzenia nowego projektu. W tym celu należy otworzyć wiersz poleceń i wykonać następujące instrukcje:

```
$ mkdir lab
$ cd lab
$ elm init
```

Po wypisaniu zawartości katalogu `lab` powinny pojawić się dwa nowe elementy:

- Plik `elm.json` opisujący projekt oraz jego zależności
- Katalog `src/`, w którym będziemy umieszczać przyszłe pliki Elm'a

Następnym krokiem będzie utworzenie nowego pliku np.: `HelloWorldTest.elm` w nowo utworzonym katalogu `src/`. Będzie się tam znajdował kod aplikacji, która zostanie stworzona w kolejnych krokach.

II. Podstawy języka Elm

W celu nauki podstaw języka Elm użyte zostanie narzędzie `elm repl`, pozwalającego na ko- rzystanie z interaktywnej sesji programistycznej. Należy otworzyć wiersz poleceń i wpisać polecenie `elm repl`. Powinien ukazać się widok podobny do przedstawionego na rysunku poniżej.

```
→ ~ elm repl
—— Elm 0.19.1 ——————————————————————————————————————————————————
Say :help for help and :exit to exit! More at <https://elm-lang.org/0.19.1/repl>
————————————————————————————————————————————————————————————————
>
```

Rysunek: Otwarta interaktywna sesja `elm repl`

Elm jest językiem funkcyjnym o składni zbliżonej do F# (por. ML), stąd składnia zostanie omówiona dość pobieżnie tak by wskazać rzucające się w oczy różnice.

Wartości

Najmniejszym budulcem aplikacji w Elmie są **wartości**. Mogą to być liczby, ciągi znaków, czy typy logiczne, np. 10, „Hello", True. Po wpisaniu do okna `elm repl` danej wartości, na ekranie powinna zostać pokazana powtórzona wartość, a po dwukropku jej typ.

Wartości można także łączyć z operatorami. Dla liczb będą to typowe operatory matematyczne, jak `+`, `-`, `*`, `/`, dla ciągów znaków operatorem konkatenacji jest `++`, a dla typów logicznych dostępne są operatory logiczne „`&&`" (AND) oraz „`||`" (OR).

Na rysunku poniżej pokazane zostały przykłady efektów takich wywołań.

```
> 2 + (2 * 2)
6 : number
```

(a) Liczba

```
> "Hello " ++ "World!"
"Hello World!" : String
```

(b) Ciąg znaków

```
> True && False
False : Bool
```

(c) Typ logiczny

Rysunek: Wartości w Elmie

Funkcje

Funkcje w Elmie określają, w jaki sposób wartości mogą zostać przetworzone. Na rysunku pokazana została przykładowa funkcja `hello`, która przyjmuje argument `name` i zwraca nowy ciąg znaków.

```
> hello name =
|   "Hello " ++ name ++ "!"
|
<function> : String → String
> hello "Bob"
"Hello Bob!" : String
> hello "Alice"
"Hello Alice!" : String
>
```

Rysunek: Definicja i użycie funkcji w Elmie

Można zauważyć, że typ argumentu `name` nie został sprecyzowany. Elm sam potrafi określić, czy dana funkcja wykona się poprawnie na podstawie operacji w niej zawartych. W pokazanym przykładzie argument `name` jest wykorzystany jako operand operatora konkatenacji „++", który potrzebuje dwóch operandów typu `String` do prawidłowego działania programu. Jeśli użytkownik zamiast ciągu znaków podałby jako argument inny typ, np. liczbę, to kompilator Elma zwróciłby na to uwagę i wystosował użytkownikowi odpowiedni komunikat.

Przykład takiego komunikatu został przedstawiony na następnym rysunku.

```
> hello 42
-- TYPE MISMATCH ------------------------------------------------

The 1st argument to `hello` is not what I expect:

6|   hello 42
           ^^
This argument is a number of type:

    number

But `hello` needs the 1st argument to be:

    String

Hint: Try using String.fromInt to convert it to a string?

>
```

Rysunek: Komunikat kompilatora o błędzie

Listy

Listy są jednymi z najczęściej używanych struktur danych w Elmie. Ich przeznaczeniem jest trzymanie sekwencji wielu elementów tego samego typu.

Na rysunku zostały pokazane przykłady użycia list. Została zdefiniowana lista `names`, zawierająca trzy elementy typu `String`, a także tablica `numbers`, która zawiera 4 liczby (typ `Int`).

```
> names = ["Bob", "Alice", "John"]
["Bob","Alice","John"] : List String
> List.length names
3 : Int
> List.reverse names
["John","Alice","Bob"] : List String
> List.isEmpty names
False : Bool
> numbers = [4,3,2,1]
[4,3,2,1] : List number
> List.sort numbers
[1,2,3,4] : List number
> List.map negate numbers
[-4,-3,-2,-1] : List number
>
```

Rysunek: Operacje na listach

Rekordy

Rekordy są zbliżone do znanych z F# i służą do trzymania wielu wartości, gdzie każda z wartości jest przypisana do konkretnej nazwy. Na rysunku pokazane zostały przykładowe operacje związane z rekordami. Zdefiniowany został rekord `bob`, który zawiera informacje o imieniu, nazwisku oraz wieku. Podobnie jak w F# dostępna jest składnia tworzenia kopii ze wskazaniem różnic – w elmie używany jest w tym celu znak `|`.

(a) Definicja rekordu i dostęp do jednego z pól

```
> bob =
|   { first = "Robert"
|   , last = "California"
|   , age = 61
|   }
|
{ age = 61, first = "Robert", last = "California" }
    : { age : number, first : String, last : String }
> bob.last
"California" : String
>
```

(b) utworzenie nowego rekordu ze zmienioną zawartością

```
> List.map .last [bob, alice, john]
["California","Cooper","Lennon"] : List String
> { bob | last = "Pattinson" }
{ age = 61, first = "Robert", last = "Pattinson" }
    : { age : number, first : String, last : String }
>
```

Rysunek: Przykłady pracy z rekordami

W przypadku rekordów, które zawierają wiele pól, praca z nimi może stawać się problematyczna. Wygodne może być wtedy wykorzystanie tzw. „aliasów typów", które pozwalają na definicję typu rekordu i korzystanie z niego w skróconej wersji.

Na rysunku zdefiniowany został nowy typ `Person`, który jest równoznaczny ze zdefiniowanym na wcześniejszym rysunku pojedynczym rekordem o nazwie `bob`. Użycie mechanizmu aliasów zamiast każdorazowego definiowania składowych sprawia, że kod staje się krótszy, bardziej czytelny, a praca z nim dużo wygodniejsza.

Dodatkowo możliwe jest również uproszczone tworzenie rekordów np.

```
anna = Person „Anna" „Karenina" 28
```

```
> type alias Person = { first: String, last: String, age: Int }
> Person
<function> : String → String → Int → Person
> Person "Bob" "Dylan" 81
{ age = 81, first = "Bob", last = "Dylan" }
    : Person
>
```

Rysunek: Definicja aliasu typu `Person`

Custom types + pattern matching

Custom types stanowią bezpośredni odpowiednik unii dyskryminowanych znanych z języka F#, mahją podobną siłę wyrazy przy minimalnie innej składni.

Przykład:

```
type UserStatus = Regular | Visitor
```

Podobnie jak w F# oprócz samego dyskryminatora – nazwanego często konstruktorem mogą mieć dodatkowe parametry np.:

```
type User
= Regular String Int Location
| Visitor String
| Anonymous
```

Mają bardzo zbliżoną składnię do F# i Również podobnie jak w F# typowo do ich obsługi używamy dopasowania wzorców.

```
case user of
Regular name age _ -> name
Visitor name -> name
_ -> „unknown”
```

Uruchamianie aplikacji

W pewnym zakresie do uruchamiania aplikacji może posłużyć również środowisko dostępne pod adresem https://elm-lang.org/try. Ma ono jednak ograniczone możliwości pod kątem np. dołączania plików js, css, doinstalowywania bibliotek itd. Z tego powodu poniżej używane będzie środowisko skonfigurowane lokalnie.

W pierwszej kolejności należy uruchomić aplikację typu „Hello World”, aby upewnić się, że środowisko zostało prawidłowo skonfigurowane. W tym celu proszę stworzyć plik HelloWorldTest.elm i skopiować do niego zawartość https://elm-lang.org/examples/hello.

I dodać na jego początku deklarację modułu.

```
module HelloWorldTest exposing (..)
```

Aby uruchomić aplikację należy otworzyć wiersz poleceń i wykonać polecenie:

```
$ elm reactor
```

Powinien pokazać się następujący widok:

Po otwarciu w przeglądarce adresu http://localhost:8000 należy znaleźć plik

```
→ lab git:(main) ✗ elm reactor
Go to http://localhost:8000 to see your project dashboard.
```

`HelloWorldTest.elm` i go otworzyć. Powinien wyświetlić się napis „Hello, World!".

Podczas kolejnych eksperymentów warto zwrócić uwagę na b. wyczerpujące komunikaty np. kompilatora, które w zasadzie pozwalają usunąć większość problemów bez dodatkowej pomocy.

Przedstawiony wyżej sposób jest chyba najprostszym sposobem uruchamiania aplikacji. Należy pamiętać, że kod elm-a jest kompilowany do Javascript. Przy otworzeniu narzędzi developerskich w przeglądarce (F12) i odnalezieniu w zakładce Debugger pliku `HelloWorldTest.elm` można przekonać się, że zawiera on znaczniki HTML a wynikowy Javascript został zagnieżdżony w tagu script.

Inny sposób uruchomienia to wymuszenie jawnej kompilacji pliku `HelloWorldTest.elm` do `HelloWorldTest.html` lub `HelloWorldTest`.js.

Aby to zrobić należy otworzyć wiersz poleceń i wykonać polecenie:

```
$ elm make src\HelloWorldTest.elm --output HelloWorldTest.html
```

lub

```
$ elm make src\HelloWorldTest.elm --output HelloWorldTest.js
```

W pierwszym przypadku efekt jest podobny jak wcześniej (proszę pamiętać o ew. odświeżeniu zawartości katalogu w przeglądarce) w tym drugim przypadku konieczne jest stworzenie piku html, właczenie kodu js i wyrenderowanie go w sposób zbliżony do następującego:

```
<div id="main" class="main"></div>
<script src=" HelloWorldTest.js"></script>
<script>
Elm. HelloWorldTest.init({
node: document.getElementById('main')
    });
</script>
```

Warto wspomnieć, że w ogólności nazwa pliku elm musi odpowiadać nazwie modułu i użytej w kodzie nazwie - w tym przypadku `HelloWorldTest`.

Struktura strony

W aktualnym przykładzie main renderuje prosty tekst. Aby uzyskać regularny kod html można posłużyć się funkcjami zawartymi w module `html` odpowiadającymi znacznikom np., `div`, `span`, `img`, `hr`. W ogólności funkcje te przyjmują dwie listy listę atrybutów oraz listę węzłów zagnieżdżonych.

Prosty przykład zagnieżdżonego div i h1

```
main = div []
            [
              h1  []
                  [ text "Nagłówek"]
            , div []
                  [ text „Hello!" ]
            ]
```

Aby kod działał konieczne jest dopisanie do pliku deklaracji importu odpowiednich funkcji:

```
import Html exposing (div, h1, text)
```

Ew. całości modułu `Html` przez zdeklarowanie `exposing(..)`

Można oczywiście podać atrybuty poszczególnych znaczników np. dla określenia klasy css, identyfikatora czy źródła obrazka tj.: `class`, `id`, `src` itd (po zaimportowaniu odpowiednich elementów z modułu `Html.Attributes`).

```
main = div []
            [
              h1 [ id "main"
                 , class "titleClass"
                 ]
                 [ text "Nagłówek"]
            , div [ class "greetingClass" ]
                  [
                     span []
                  ,  [text "Greeting:"]
                  , text "Hello!"
                  ]
            ]
```

Refaktoryzacja

Ponieważ części strony stanowią fragmenty kodu napisane w języku elm możliwa jest refaktoryzacja kodu do mniejszych funkcji, wykorzystanie parametrów itd.

Niniejszy przykład definiuje dwie funkcje `pageContent` i wywkorzystywaną przez nią `greetingView`.

```
greetingView greetingText greetingCssClass =
div [ class greetingCssClass ]
                 [
                    span []
                 ,  [text "Greeting:"]
                 ,  text greetingText
                 ]

pageContent greetingText =
                [
                  h1  [ id "main"
                       ,  class "titleClass"
                       ]
                , [ text "Nagłówek"]
                , greetingView greetingText "greetingClass"
                ]

main = div []
           (pageContent „hello")
```

Pytanie kontrolne.

Jakie są typy zwracane przez pageContent i greetingView i jak wpływa to na ich użycie – dlaczego wymagane jest użycie nawiasu w „`(pageContent „hello")`" a nie jest potrzebne przy wywołaniu `greetingView`.

III. Zadanie 1. Galeria - statyczna strona

Proszę zdefiniować prosty layout strony wg. wzoru podanego przez prowadzącego. Strona powinna zawierać obrazki oraz wykorzystywać css. Szczegóły obrazków mogą być zahardcodowane w definicji widoku np.

```
main =
…
       div []
           [
                 viewPhotoAndDetails "url1 " "Uklad sloneczny"
           ,     viewPhotoAndDetails "url2 " "Slonce"
           ,     viewPhotoAndDetails "url2 " "Ziemia"
```

Przykładowa komplikacja pokazana na nast. rysunku.]

Galeria

Uklad sloneczny

Źródło:https://upload.wikimedia.org/wikipedia/commons/a/a9/Planety2008.jpg

Slonce

Źródło:https://upload.wikimedia.org/wikipedia/commons/3/37/Skylab_Solar_flare.jpg

Ziemia

Źródło:https://upload.wikimedia.org/wikipedia/commons/c/cb/The_Blue_Marble_%28remastered%29.jpg

Rysunek: Przykładowy layout strony

IV. Dynamika

Kolejnym krokiem jest stworzenie aplikacji mającej pewną dynamikę i zbudowanej na wzorcu MVU mającym odbicie w strukturze kodu Elma, tj. `Model`, `Update` oraz `View`, czyli elementów odpowiedzialnych odpowiednio za stan, logikę i wygląd aplikacji.

Uruchamianie aplikacji

W załączonym pliku `MainHelloSandbox.elm` zaimplementowana została podstawowa architektura aplikacji. Aby uruchomić aplikację należy dokonać drobnej modyfikacji kodu – rozwiązanie podpowie środowisko/kompilator.

W efekcie działania aplikacji powinien wyświetlić się napis „Hello, World!” oraz guzik `Click me!`, który zamienia ciąg tekstowy na „Hello, again!”.

Diagnostyka

W ogólności przy programowaniu funkcyjnym wykonanie kodu krok po kroku jest trudne i niepraktyczne. Bardzo pomocny może być jednak podgląd wartości uzyskiwanych w trakcie wykonania kodu w tym celu można wykorzystać wykorzystać funkcję Debug.log. Funkcja ta ma

dwa parametry – pierwszym jest komunikat diagnostyczny a drugim są dane do wydrukowania. Ponieważ funkcja zwraca wartość przekazaną jako drugi parametr można ją wygodnie wpinać w różne miejsca analizowanego kodu.

Dla przykładu w testowanym pliku należy dodać

```
import Debug exposing (..)
```

oraz zmodyfikować np. funkcję update

```
update msg model =
case msg of
Hello greet ->
          Debug.log "DEBUG returned by update"
                    { model | greet = greet }
```

Komunikaty diagnostyczne widoczne będą w konsoli Javascript w przeglądarce.

Struktura kodu

W dostarczonym pliku można zauważyć sekcję główną (wykorzystuje ona model `sandbox` opisany dokładniej np.: tu https://guide.elm-lang.org/architecture/)

Trzy pola rekordu, będącego parametrem Browser.sandbox to odpowiednio stan inicjalny modelu, widok – funkcja renderująca HTML w oparciu o aktualny model i update funkcja tworząca nowy model (stan) w oparciu o istniejący model i komunikat (Msg). Modelem może być praktycznie cokolwiek np. Int, String, Bool lub rekord. Komunikat również definiuje programista zwykle jest to custom type (dyskryminowana unia). W przykładzie

```
type Msg
    = Hello String
```

`Msg` to custom type z jedną możliwością `Hello` i parametrem typu `String`

W tym przykładzie umieszczone zostały deklaracje typów nie są one obowiązkowe, ale ułatwiają planowanie kodu i jako takie są zalecane.

```
init : Model
update : Msg -> Model -> Model
view : Model -> Html Msg
```

Jak widać `init` jest typu `Model`, `update` jest funkcją, która na podstawie messedza i modelu tworzy nowy model, a `view` na podstawie modelu tworzy widok typu `Html` – (jeśli chodzi o znaczenie zapisu `Html Msg` na razie możemy patrzeć na to jako na parametr `Msg` szablonu `Html`).

Warto zwrócić uwagę, że częścią widoku jest przycisk

```
button [ onClick (Hello "Hello, again!") ] [ text "Click me!" ]
```

gdzie w atrybutach zdefiniowano, że jako reakcję na zdarzenie `onClick` należy wysłać msg `Hello "Hello, again!"`.

V. Zadanie 2. Galeria – ukrywanie szczegółów

Dla zadania 1, proszę dodać przycisk pozwalający pokazywać lub chować szczegóły wszystkich obrazków (szczegóły = wszystko poza podpisem) .

Wskazówka: model może zawierać pojedynczą wartość.

VI. Zadanie 3. Galeria – ukrywanie szczegółów indywidualnie

- Proszę przenieść definicje obrazków do modelu (tj. model będzie listą rekordów opisujących poszczególne obrazki) i rozszerzyć ten rekord o pole id.
- Dla każdego obrazka na stronie proszę dodać przycisk pozwalający ukrywać/pokazywać szczegóły dla tego obrazka.

Wskazówki:

Dla listy modelu będącego listę rekordów (tj. definicji obrazów) można wykorzystać funkcję map zamieniającą model na odpowiadający mu podwidok obrazka. Tj.

```
view model =
div []
    (List.map viewSinglePhoto model)
```

A komendy mogą mieć postać np.:

```
ToggleDetails id
```

lub

```
ShowDetails id | HideDetails id
```

Sam update bazowałby w takiej sytuacji również na map która "zmieniałaby" rekord o zadanym id. Natomiast przy renderowaniu poszczególnych przycisków w widoku należy użyć odpowiednią komendy z odpowiednim parametrem.