Instrukcja laboratoryjna ELM-2

(M.Jurkiewicz, K.Manuszewski)

Niniejsza instrukcja zakłada wykonanie poprzedniego laboratorium z ELMa i nie zawiera powtórzonych informacji np. nt. podstaw języka, uruchamiania struktury programu itd.

Aplikacja frontendowa

W ramach większego projektu zbudowana zostanie strona internetową typu `startpage`, czyli strona startowa przeglądarki zawierająca najważniejsze i najczęściej używane elementy. Docelowo zaimplementowane zostaną następujące moduły — Data i czas, pogoda, wyszukiwarka oraz pasek zakładek.

Po kolei zostaną dodane do kodu kolejne fragmenty w ramach każdej z części architektury Elma, tj. `Model`, `Update` oraz `View`, czyli elementów odpowiedzialnych odpowiednio za stan, logikę i wygląd aplikacji.

Uruchamianie aplikacji

W załączonym pliku `MainHello.elm` znajdują się wszystkie potrzebne do działania klauzule importujące biblioteki podstawowe Elma. Ponadto 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!".

Struktura kodu

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

Różnica w stosunku do browser.sandbox (używanego w ramach poprzedniej lab.) polega na wykorzystaniu MSG jako typu zwracanego, przekazywanego do funkcji oraz obecności subscriptions.

Celem dalszych sekcji będzie rozwinięcie tego programu poprzez implementację wymaganych funkcjonalności omówionych na początku instrukcji.

I. Czas

Do wydobycia informacji o aktualnym czasie wykorzystana zostanie biblioteka `elm/time`. W celu zainstalowania tej biblioteki w wierszu poleceń należy wykonać komendę `$ elm install elm/time`, a następnie załączyć bibliotekę do tworzonego programu używając klauzuli `import Time exposing (..)`.

Przechodząc do edycji kodu, pierwszą rzeczą jest zdefiniowanie modelu programu. Potrzebne będą informacje o strefie czasowej oraz aktualny czas. W tym celu należy stworzyć nowy typ `DateTime`, który będzie przechowywał te dane, a następnie dodać go do modelu.

```
type alias Model = { dateTime : DateTime }

type alias DateTime = { zone : Time.Zone, time : Time.Posix }
```

Kolejnym krokiem będzie zdefiniowanie typów wiadomości, które może odebrać program. Do prawidłowego działania zegara potrzebne będzie dostosowanie strefy czasowej oraz pobranie aktualnego czasu. Oznacza to, że należy zdefiniować dwie wiadomości — `AdjustTimeZone` oraz `Tick`.

```
type Msg  = Tick Time.Posix
| AdjustTimeZone Time.Zone
```

Oba typy wiadomości muszą zostać odpowiednio przetworzone w funkcji `update`. W obu przypadkach wystarczające będzie nadpisanie stworzonego modelu nowymi wartościami. Przykład nad pisania rekordu został przedstawiony wcześniej.

Następnie należy zdefiniować funkcję `init`, gdzie wskazany zostanie sposób inicjalizacji modelu oraz operacje, jakie mają zostać wykonane na początku programu. W przypadku daty i czasu należy początkowo wyzerować obie wartości, a następnie pobrać aktualne informacje wykorzystując stworzone przed chwilą wiadomości.

```
init _ =
( Model (DateTime Time.utc (Time.millisToPosix 0))
, Cmd.batch [ Task.perform AdjustTimeZone Time.here
, Task.perform Tick Time.now
            ]
    )
```

Jednakże czas zmienia się z sekundy na sekundę, więc jednorazowe ustawienie wartości modelu niestety nie jest wystarczające — trzeba aktualizować model co sekundę. Aby to osiągnąć, wykorzystana zostanie funkcja `subscriptions`. Pozwala na nasłuchiwanie zewnętrznych zdarzeń, takich jak kliknięcie myszki, naciśnięcie klawisza na klawiaturze, zmiany w geolokalizacji lub — tykanie zegara.Jednakże czas zmienia się z sekundy na sekundę, więc jednorazowe ustawienie wartości modelu niestety nie jest wystarczające — trzeba aktualizować model co sekundę. Aby to osiągnąć, wykorzystana zostanie funkcja `subscriptions`. Pozwala na nasłuchiwanie zewnętrznych zdarzeń, takich jak kliknięcie myszki, naciśnięcie klawisza na klawiaturze, zmiany w geolokalizacji lub — tykanie zegara.

W przypadku tworzonej aplikacji należy co sekundę generować nową wiadomość `Tick`, która po jej przetworzeniu w funkcji `update` zaktualizuje aktualny czas w modelu.

```
subscriptions _ = Time.every 1000 Tick
```

Pozostaje zaktualizowanie funkcji update

```
Tick newTime ->
```

```
      (  { model | dateTime =
                      DateTime model.dateTime.zone newTime }
,  Cmd.none
      )


 AdjustTimeZone newZone ->
      (  { model | dateTime =
                      DateTime newZone model.dateTime.time }
,  Cmd.none
      )
```

Ostatnim elementem architektury Elma jest funkcja `view`, której zadaniem jest wyświetlanie programu na ekranie.

```
view model =
{ title = "Hello"
, body = [ viewTime model.dateTime
, viewDate model.dateTime
            ]
    }
```

W celu polepszenia czytelności kodu, funkcja `view` wykorzystuje dwie funkcje pomocnicze – `viewTime` oraz `viewDate`. Część implementacji jednej z nich została przedstawiona poniżej. W ramach ćwiczenia należy dokończyć implementację funkcji `viewDate` oraz stworzyć analogiczną funkcję `viewTime`. Ponadto przedstawiony został początek funkcji `toEnglishWeekday`, która przyjmuje wartość typu `Time.Weekday` i zwraca typ `String`, który może zostać następnie wyświetlony w funkcji `viewDate`. Implementację tej funkcji także należy dokończyć oraz stworzyć analogiczną funkcję pozwalającą na dekodowanie nazw miesięcy. Dla lepszego wyglądu zegara warto także stworzyć funkcję dodającą znak 0 dla liczb mniejszych od 10.

```
viewDate dateTime =
let
weekday = Time.toWeekday dateTime.zone dateTime.time
  ...
in
div [] [ text (toEnglishWeekday weekday) ]
```

```
toEnglishWeekday weekday =
case weekday of
Mon -> "Monday"
   ...
```

Jeżeli wszystko udało się pomyślnie, w przeglądarce internetowej powinien wyświetlić się aktualny czas, zmieniający się co sekundę, a pod nim data.

Wynikowy plik z kodem proszę nazwać `MainTime.elm`.

II. Wyszukiwarka

Wynik tego zadania (łącznie z poprzednim) powinien znaleźć się w pliku `MainSearch.elm`.

Wyszukiwarka jest jedną z prostszych funkcjonalności do implementacji. Potrzebne będzie jedynie stworzenie formularza obejmującego jedno pole tekstowe, gdzie użytkownik może wpisać wybraną frazę, oraz jego odpowiednie obsłużenie. Po wpisaniu frazy w formularzu i wciśnięciu klawisza Enter użytkownik powinien zostać przeniesiony do strony wyszukiwarki z wyszukanym wpisanym już wcześniej hasłem. Pod "maską" w momencie modyfikacji pola wysłany jest komunikat, który aktualizuje model o frazę z formularza. Wzięta z modelu wartość (Fraza) będzie podlegać wyszukiwaniu po przekierowaniu do wyszukiwarki.

Tak więc tym razem do modelu należy dołożyć jedynie jedną wartość `searchText : String` – ciąg znaków wpisany przez użytkownika.

Oznacza to dodanie do inicjalizacji nowego parametru.

```
Model (DateTime Time.utc (Time.millisToPosix 0)) ""
```

Natiomiast nie potrzeba inicjalnie wykonywać żadnej akcji (jak np. ustawianie czasu) więc lista komend wykonywanych inicjalnie `Cmd.batch [ … ]` nie wymaga zmiany.

Wiadomości typu `Msg` zgodnie z zapowiedzianą logiką będą dwie – jedna standardowo odpowiedzialna za zaktualizowanie modelu w wyniku modyfikacji formularza, a druga za przekierowanie strony do wyszukiwarki.

```
    | UpdateField String
    | Search
```

W `update` komunikat `UpdateField` aktualizuje wartość przechowywaną w modelu.

```
UpdateField searchText ->
              ( { model | searchText = searchText }
              , Cmd.none
              )
```

Przekierowanie odbywa się za pomocą funkcji `load` z biblioteki `Navigation`, która powinna już być załączona do programu w początkowym szkielecie. Czyli obsługa `Search` powinna wyglądać następująco:

```
Search -> ( model
          , Nav.load ("https://google.com/search?q="
                      ++ model.searchText)
          )
```

Poniżej została przedstawiona propozycja implementacji widoku odpowiadającego za wyświetlanie pola tekstowego `viewSearchBar`. Trzeci i czarty parametr odopowdają zdrzeniom generowanym przez formularz . Pomocnicza funkcja `onEnter` pozwala na wykrycie czy użytkownik wcisnął klawisz Enter.

```
onEnter msg =
let
  isEnter code = if code == 13 then succeed msg
                               else fail "not ENTER"
in
  on "keydown" (andThen isEnter keyCode)
```

```
viewSearchBar = div []
                    [ input [ type_ "text"
                            , placeholder "Search"
                            , onInput UpdateField
                            , onEnter Search
                            ]
                            []
                    ]
```

Trzeba jeszcze zaimportować `succeed, fail, andThen`

```
import Json.Decode exposing (succeed, fail, andThen)
```

i ew. zainstalować ten moduł przez

```
$ elm install elm/json
```

W ramach ćwiczenia należy już tylko rozszerzyć funkcję `view` o wywołanie przedstawionej wyżej propozycji `viewSearchBar`. Po prawidłowym złączeniu wszystkich fragmentów kodu program powinien się kompilować i wyświetlać nowy pasek pola tekstowego, a po wciśnięciu klawisza Enter strona powinna zostać przekierowana do wyszukiwarki Google.

Aby lepiej zrozumieć działanie updatów i cykl aktualizacji, można skorzystać z techniki `Debug.log` opisanej w Lab1. Np. można dopisać poniższe wołania i obserwować efekt w przeglądarce w zakładce Konsola.

```
let isEnter code =  if (Debug.log "onEnter:" code) == 13
                                then succeed msg
                                else fail "not ENTER"
```

i

```
UpdateField searchText ->
                ( { model |
              searchText = Debug.log "UpdateField:" searchText }
                , Cmd.none
                )


Search -> ( Debug.log "Search:" model
          , Nav.load ("https://google.com/search?q="
                          ++ model.searchText)
                   )
```

III. Pogoda

Wynik tego zadania (łącznie z poprzednim) powinien znaleźć się w pliku `MainWeather.elm`.

Pogoda będzie pobierana wykorzystując zapytania HTTP do API portalu OpenWeatherMap. Potrzebne będą do tego dwie biblioteki — `elm/http` do wysłania zapytania oraz `elm/json` do odebrania i przetworzenia odpowiedzi w formacie JSON. Podobnie jak poprzednio, należy je zainstalować używając poleceń:

```
$ elm install elm/http
$ elm install elm/json
```

Następnie należy dodać do pliku następujące klauzule:

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

Przed przejściem do edycji kodu programu należy założyć darmowe konto na stronie OpenWeatherMap, a następnie wygenerować klucz API, który będzie konieczny do prawidłowego działania aplikacji. Po stworzeniu prywatnego klucza API można przejść do kolejnych kroków.

Szczegóły użycia API opisane są w dokumentacji OpenWeatherMap.

Z perspektywy Elm-a najważniejsze jest sprecyzowanie rodzaju zapytania (tutaj GET) oraz adresu URL, na który ma został wysłane zapytanie. Konieczne jest także określenie formy spodziewanej odpowiedzi (JSON), rodzaju wiadomości do wygenerowania a także przekazanie dekodera odpowiedzi (np. dekodera JSON). W tym celu wykorzystamy moduł Json.Decode stąd konieczne jest użycie odpowiedniego importu w kodzie.

Dekodowanie JSON-a

Jak wygląda dekoder i do czego służy? Przykład prostego dekodera został przedstawiony poniżej. Celem dekodera jest przetworzenie pliku w formacie JSON w taki sposób, aby był zrozumiały dla Elma. Należy w nim określić nazwy oczekiwanych pól oraz opisać jak się do nich dostać.

Funkcja `personDecoder` spodziewa się trzech wartości — dwóch ciągów znaków i jednej liczby, które znajdują się w polach nazwanych odpowiednio `first`, `last` oraz `age`. Oznacza to, że taka odpowiedź JSON powinna zostać prawidłowo zmapowana na typ `Person`.

```
type alias Person = { first: String
, last : String
, age : Int
}
personDecoder = map3 Person
(field "first" string)
(field "last" string)
(field "age" int)
```

Aby przekonać się jak działa dekodowanie można wykorzystać `elm repl` wykonać powyższy przykład (proszę nie zapomnieć o wcześniejszym zaimportowaniu wszystkiego z `Json.Decode`). Żeby zobaczyć efekt dekodera ze zdefiniowanym opisem wejścia można

uruchomić poniższe polecenie dekodujące JSON-a umieszczonego w string-u.

Dla eksperymentów z dekodowaniem Jsona przydatne może być również wykorzystanie koncepcji `Debug.log`, opisanej w poprzednim laboratorium. Wynik dekodowania zwracany jest w typer result zbliżonym do analogicznego znanego z F#.

```
type Result error value =  Ok value | Err error
```

Proszę poeksperymentować z podanym przykładem – modyfikując podany jako parametr Json-a. Jak zachowuje się dekoder w przypadku pól niezdefiniowanych w dekoderze a jak w przypadku braku zdefiniowanych.

```
type alias Info = { status: String
                  , city : String
                  }
infoDecoder = map2 Info
                     (field "status" string)
                     (field "address" (field "city" string))
```

Dla

```
decodeString infoDecoder "{ \"status\":\"Test\", \"address\":
{\"city\":\"Gdansk\", \"Street\":\"Dluga\" }}"
```

Dla potrzeb laboratorium potrzebne będzie dekodowanie zwrotu z OpenWeather i wydostanie z niego pól description i temp.

Przykładowy zwrot

```
{
    "coord": {
        "lon": 18.6464,
        "lat": 54.3521
    },
    "weather": [
        {
            "id": 803,
            "main": "Clouds",
            "description": "broken clouds",
            "icon": "04d"
        }
    ],
    "base": "stations",
    "main": {
```

```
        "temp": 7.69,
        "feels_like": 4.07,
        "temp_min": 7.02,
        "temp_max": 9.39,
        "pressure": 1005,
        "humidity": 86
    },
    "visibility": 10000,
    "wind": {
        "speed": 6.69,
        "deg": 190
    },
    "clouds": {
        "all": 75
    },
    "dt": 1673532783,
    "sys": {
        "type": 2,
        "id": 2079119,
        "country": "PL",
        "sunrise": 1673506851,
        "sunset": 1673534748
    },
    "timezone": 3600,
    "id": 3099434,
    "name": "Gdańsk",
    "cod": 200
}
```


Implementacja pobierania pogody

W przypadku pogody potrzebne będzie stworzenie dwóch nowych typów: jeden będzie zawierał informacje o faktycznym stanie pogody, tj. temperatura i krótki opis słowny, a drugi informacje o statusie zapytania HTTP.

```
type WeatherStatus = Failure String
| Loading
| Success Weather
type alias Weather = { description : String
, temperature : Float
                     }
```

Pole `weatherStatus` typu `WeatherStatus` należy dodać do głównego modelu. Ponadto należy zdefiniować w programie dane potrzebne do wysłania zapytania, a następnie wykorzystać je w funkcji odpowiedzialnej za wysłanie zapytania GET do OpenWeatherAPI. Są to:

- URL do wysłania zapytania
- Prywatny klucz API do OpenWeatherMap
- Miasto
- Jednostki

Przykład implementacji został przedstawiony poniżej:

```
weatherApi =
"https://api.openweathermap.org/data/2.5/weather?"
apiKey = "???"
type Unit = Celsius
| Fahrenheit
| Kelvin
unitStr = case unitForTemp of
Celsius -> "metric"
Fahrenheit -> "imperial"
Kelvin -> ""
getWeather city unit = Http.get
         { url = weatherApi
               ++ ("&q=" ++ city)
               ++ ("&units=" ++ (unitStr unit))
               ++ ("&appid=" ++ apiKey)
         , expect = Http.expectJson GotWeather weatherDecoder
         }
getGdanskWeather = getWeather "Gdansk" Celsius
```

W zapytaniu http.get oprócz szczegółów zapytania należy jeszcze przekazać oczekiwany typ zwrotu, rodzaj wiadomości do wygenerowania po odebraniu odpowiedzi oraz dekoder. W naszym przypadku rodzaj wiadomości to `GotWeather`.

Następnie w ramach ćwiczenia należy zaimplementować funkcję `weatherDecoder`, która zmapuje odpowiednie pola z odpowiedzi JSON na typ `Weather` zaimplementowany wcześniej w modelu.

W przypadku wiadomości `Msg` sytuacja będzie podobna jak przy tworzeniu typów — potrzebne będzie **dodanie** nowych rodzajów wiadomości jeden rodzaj wiadomości odpowiedzialny za wysłanie zapytania oraz drugi odpowiedzialny za jego odebranie i odpowiednie zaktualizowanie modelu na podstawie danych odebranych z dekodera.

```
type Msg
= ...
| UpdateWeather
| GotWeather (Result Http.Error Weather)
```

```
update msg model =
   case msg of
        ...
        UpdateWeather ->
            ( { model | weatherStatus = Loading }
            , getGdanskWeather )
        GotWeather result ->
             case result of
                 Ok weather ->
                     ( { model | weatherStatus = Success weather}
                       , Cmd.none )
                 Err _ ->
                     ( { model | weatherStatus = Failure
                           "Error: Couldn't retrieve weather data"
                       }
                       , Cmd.none)
```

Konieczne również będzie zmodyfikowanie inicjalizacji przez dodanie parametru `Loading` do inicjalizacji Modelu oraz do inicjalnej listy komend do wykonania `getGdanskWeather` oraz zdefiniowanie odpowiedniego widoku.

```
viewWeather : WeatherStatus -> Html Msg
viewWeather weatherStatus =
   div [] [ viewWeatherStatus weatherStatus ]

viewWeatherStatus : WeatherStatus -> Html Msg
viewWeatherStatus weatherStatus =
  case weatherStatus of
    Failure errorMsg -> text errorMsg
    Loading -> text "Loading weather..."
    Success weather ->
        text (getDesc weather ++ " | " ++ getTemp weather)

getDesc weather = weather.description
getTemp weather = ???
```

Należy zdefiniować funkcję getTemp i dodać widok pogody do widoku głównego przed `searchView`.

IV. Zakładki

Kod Elma (obejmujący poprzednie punkty) powinien znaleźć się w pliku `MainBookmarks.elm`. W dalszej części implementacji do prawidłowego działania zakładek potrzebne będzie użycie pliku `index.html` i stworzenie dodatkowego pliku `bookmarks.js`, który posłuży do trzymania listy zakładek, gdzie każda składa się z adresu URL oraz nazwy, która zostanie wyświetlona na tworzonej stronie.

Natomiast drugi plik `index.html` potrzebny będzie do załadowania do programu poprzedniego pliku z zakładkami jak i pliku wynikowego stworzonego przez kompilator Elma z użyciem następującej komendy:

```
$ elm make src/MainBookmarks.elm --output=assets/elm.js
```

W sekcji `<head>` pliku `index.html` należy załączyć plik z zakładkami oraz wynikowy plik kompilatora. W sekcji `<body>` należy dodać fragment kodu przedstawiony poniżej, który pozwoli na wykorzystanie programu w Elmie razem z zewnętrznym plikiem HTML.

```
<head>
<script src="assets/elm.js">
</script>
<script
src="assets/bookmarks.js">
</script>
</head>
```

```
<body>
<div id="app"></div>
<script>
var app = Elm.Main.init({
node: document.getElementById("app "),
flags: bookmarks,
});
</script>
</body>
```

Po otwarciu pliku `index.html` z użyciem programu `elm reactor` aplikacja wraz z zakładkami powinna pojawić się na ekranie.

V. Style

W załączonym pliku `styles.css` znajdują się kaskadowe arkusze stylów, które mogą zostać wykorzystane wraz ze stworzoną aplikacją. W tym celu należy odpowiednio zmienić w programie funkcje odpowiedzialne za wyświetlanie elementów, dodając do nich tagi `div[][]`. Funkcja `div` przyjmuje dwa argumenty: listę atrybutów oraz listę elementów HTML. W tej części należy skupić się na pierwszym argumencie, gdyż dotychczas nie był on jeszcze używany. Atrybuty HTML to np. `class`, `id` oraz `style`.

Na przykładzie głównej funkcji `view`, wykorzystanie klasy `container` wygląda następująco:

```
view model =
{ ...
, body =
[ div [ class "container" ] [ viewTime model.clockTime
... ] ]
}
```

Jako, że plik `styles.css` został dostarczony w załączniku, w ramach ćwiczenia w aplikacji wystarczy jedynie dodać w odpowiednich miejscach funkcje `div` wraz z pasującymi atrybutami, które zostały zdefiniowane w arkuszu stylów.

VI. Efekt

Jeżeli wszystkie funkcjonalności zostały prawidłowo zaimplementowane, a arkusze stylów właściwie użyte w programie, finalna aplikacja powinna wyglądać następująco: