Linguagens e Ambientes de Programação (Aula Teórica 16)

LEI - Licenciatura em Engenharia Informática

João Costa Seco (joao.seco@fct.unl.pt)

NOVA SCHOOL OF SCIENCE & TECHNOLOGY

Agenda

- Functional reactive programming in Elm
- Live Programming
- Event based programming in Javascript/React/Redux

Linguagem de programação Elm (elm-lang.org)

- É uma linguagem funcional que compila para JavaScript
- Não tem erros de runtime na prática
- Mensagens de erro inteligíveis
- Refactoring e evolução seguras

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

main =
  Browser.sandbox { init = 0, update = update, view = view }

type Msg = Increment | Decrement

update msg model =
  case msg of
    Increment ->
      model + 1

    Decrement ->
      model - 1

view model =
  div []
    [ button [ onClick Decrement ] [ text "-" ]
    , div [] [ text (String.fromInt model) ]
    , button [ onClick Increment ] [ text "+" ]
    ]
```

Linguagem de programação Elm (elm-lang.org)

- Arquitectura de uma aplicação Elm:
- **Model** - o estado da aplicação
- **View** - uma transformação do estado para HTML
- **Update** - ações para atualizar o estado com base em mensagens.

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

main =
  Browser.sandbox { init = 0, update = update, view = view }

type Msg = Increment | Decrement

update msg model =
  case msg of
    Increment ->
      model + 1

    Decrement ->
      model - 1

view model =
  div []
    [ button [ onClick Decrement ] [ text "-" ]
    , div [] [ text (String.fromInt model) ]
    , button [ onClick Increment ] [ text "+" ]
    ]
```

Model

- Representação de todos os dados que são manipulados pela aplicação.
- Para um estado, precisamos de um tipo e de um valor inicial.

```
type alias Model = Int
```

```
init : Model
init =
  0
```

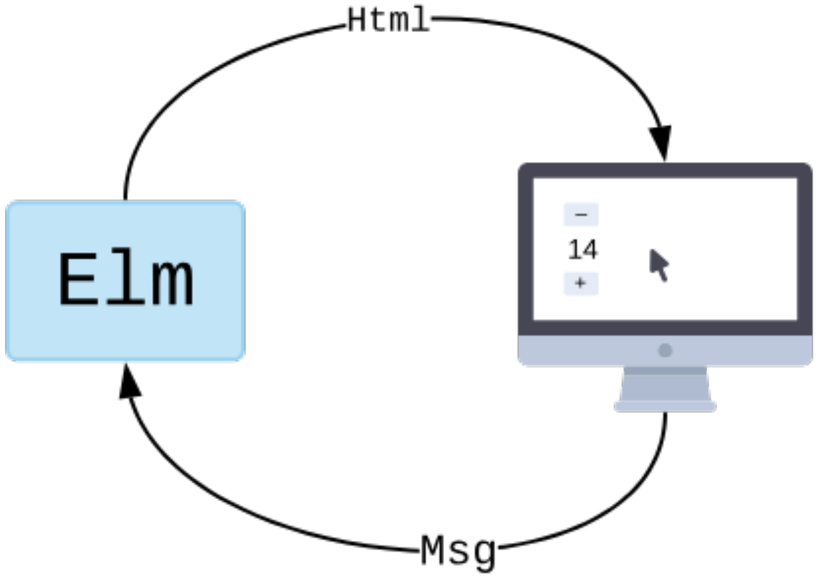
View

- Representação visual da aplicação dado os dados do modelo.
- O resultado é HTML, com a estrutura dos elementos e mensagens para os eventos.

```
view : Model -> Html Msg
view model =
  div []
    [ button [ onClick Decrement ] [ text "-" ]
    , div [] [ text (String.fromInt model) ]
    , button [ onClick Increment ] [ text "+" ]
    ]
```

Update

- Representa a evolução do modelo consoante as mensagens que são enviadas.




```
type Msg = Increment | Decrement
```

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

    Decrement ->
      model - 1
```

https://guide.elm-lang.org/architecture/buttons

Conteúdo de campos

- O Conteúdo de campos na UI é passado por parâmetro na mensagem gerada.

```
import Browser
import Html exposing (Html, Attribute, div, input, text)
import Html.Attributes exposing (..)
import Html.Events exposing (onInput)

main = Browser.sandbox { init = init, update = update, view = view }

type alias Model =  { content : String  }

init : Model
init = { content = "" }


type Msg = Change String


update : Msg -> Model -> Model
update msg model =
  case msg of
    Change newContent ->
      { model | content = newContent }


view : Model -> Html Msg
view model =
  div []
    [ input [ placeholder "Text to reverse", value model.content, onInput Change ] []
    , div [] [ text (String.reverse model.content) ]
    ]
```

Comunicação assíncrona

- Para além de mudanças no modelo também há mensagens geradas por eventos de rede e outros (subscrições).

```
view : Model -> Html Msg
view model =
  case model of
    Failure ->
      text "I was unable to load your book."

    Loading ->
      text "Loading..."

    Success fullText ->
      pre [] [ text fullText ]
```

```
main = Browser.element
       { init = init, update = update,
       subscriptions = subscriptions, view = view }

type Model = Failure | Loading | Success String

init : () -> (Model, Cmd Msg)
init _ =
  ( Loading
  , Http.get
      { url = "https://elm-lang.org/assets/public-opinion.txt"
      , expect = Http.expectString GotText
      }
  )

type Msg = GotText (Result Http.Error String)

update : Msg -> Model -> (Model, Cmd Msg)
update msg model =
  case msg of
    GotText result ->
      case result of
        Ok fullText ->
          (Success fullText, Cmd.none)

        Err _ ->
          (Failure, Cmd.none)

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

Comunicação assíncrona

- Para além de mudanças no modelo também há mensagens geradas por eventos de rede e outros (subscrições).

```
view : Model -> Html Msg
view model =
  case model of
    Failure ->
      text "I was unable to load your book."

    Loading ->
      text "Loading..."

    Success fullText ->
      pre [] [ text fullText ]
```

```
main = Browser.element
       { init = init, update = update,
       subscriptions = subscriptions, view = view }

type Model = Failure | Loading | Success String

init : () -> (Model, Cmd Msg)
init _ =
  ( Loading
  , Http.get
      { url = "https://elm-lang.org/assets/public-opinion.txt"
      , expect = Http.expectString GotText
      }
  )

type Msg = GotText (Result Http.Error String)

update : Msg -> Model -> (Model, Cmd Msg)
update msg model =
  case msg of
    GotText result ->
      case result of
        Ok fullText ->
          (Success fullText, Cmd.none)

        Err _ ->
          (Failure

subscriptions : Mo
subscriptions mode
```

```
subscriptions : Model -> Sub Msg
subscriptions model =
  Time.every 1000 Tick
```

ReactJS

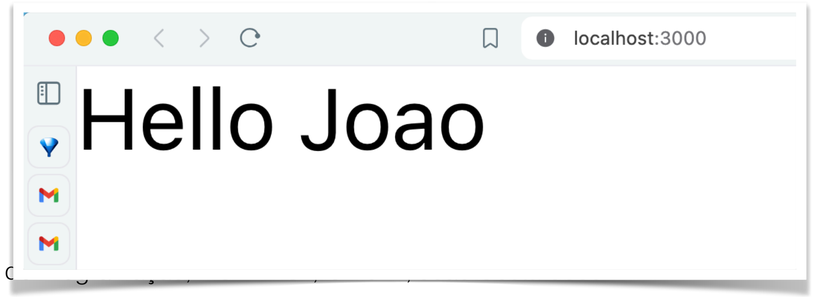
- Implementa reatividade numa framework de componentes JavaScript (Typescript)
- O interface é JavaScript-first.
- Um componente React é uma função que produz um componente React que vai corresponder a elementos HTML.
- Os parâmetros da função e atributos dos componentes podem ser usados para configurar o componente.

```
function HelloBox(props: {name: string}) {
  return <div>Hello {props.name}</div>
}

function App() {
  return <div><HelloBox name="Joao" /></div>
}

export default App;
```

```
const root = ReactDOM.createRoot(
  document.getElementById('root') as HTMLElement
);
root.render(
  <React.StrictMode>
    <App />
  </React.StrictMode>
);
```



ReactJS

- Os componentes podem ter estado próprio, são renovados cada vez que o estado muda.
- O *hook* `useState` cria uma variável de estado e devolve um seletor (`seconds`) e um modificador (`setSeconds`).
- O *hook* `useEffect` executa pela primeira vez (montar), e pela última (desmontar o componente).

```
const Timer = () => {
    const [ seconds, setSeconds ] = useState(0)

    let tick = () => { setSeconds((seconds) => seconds+1) }

    let interval: NodeJS.Timeout | null = null;

    useEffect(() => {
      interval = setInterval(() => tick(), 1000);
      return () => { interval && clearInterval(interval); }
    }, []);

    return (
        <div>
           Seconds elapsed since you arrived: {seconds}
        </div>
    )
};
```

ReactJS

- Este é um componente sem propriedades

```
const Timer = () => { … };
```

- O estado contém o número de segundos, a começar em 0.

```
const [ seconds, setSeconds ] = useState(0)
```

- A variável `seconds` contém o estado atual, a função `setSeconds` permite alterar o estado corrente em relação ao estado atual (`seconds`).

```
let tick = () => { setSeconds((seconds) => seconds+1) }
```

- É instalado um timer para fazer avançar o relógio, quando o componente é instalado (mounted), a função callback apaga o timer quando o componente é desentalado.

```
useEffect(() => {
  interval = setInterval(() => tick(), 1000);
  return () => { interval && clearInterval(interval); }
}, []);
```

Live Programming Demo