# Visualização e animação de autómatos em Ocsigen Framework*

Rita Macedo, Artur Miguel Dias, António Ravara

NOVA LINCS e DI-FCT, Universidade NOVA de Lisboa

**Resumo** Linguagens Formais e Teoria de Autómatos são bases importantes na formação em Engenharia Informática. O seu carácter rigoroso e formal torna exigente a sua aprendizagem. Um apoio importante à assimilação dos conceitos é a possibilidade de se visualizar interactivamente exemplos concretos destes modelos computacionais, facilitando a compreensão dos mesmos. As ferramentas disponíveis não são completas nem suportam completamente o aspecto interactivo.
Este projecto visa o desenvolvimento de uma ferramenta web interativa, em Português, para ajudar de forma assistida e intuitiva a compreender os conceitos e algoritmos em causa, vendo-os a funcionar passo-a-passo, através de exemplos típicos pré-carregados ou construídos pelo utilizador (um aspecto original da nossa plataforma). A ferramenta deve, por isso, permitir criar e editar autómatos, bem como executar os algoritmos clássicos relevantes, como aceitação de palavras, conversões entre modelos, etc. Pretende-se, também, visualizar não só o processo de construção do autómato, como todos os passos de aplicação de dado algoritmo.
Esta ferramenta usa o *Framework* Ocsigen, pois este proporciona o desenvolvimento de ferramentas web completas e interactivas escritas em OCaml, uma linguagem funcional com um forte sistema de verificação de tipos e, por isso, perfeita para se obter uma página web sem erros. O Ocsigen foi escolhido também porque permite a criação de páginas dinâmicas com sistema de cliente-servidor único.
Este artigo apresenta a primeira fase do desenvolvimento do projecto, sendo já possível criar autómatos, aferir a natureza dos seus estados e verificar passo-a-passo (com *undo*) a aceitação de uma palavra.

**Palavras-chave:** Linguagens Formais, Autómatos, OCaml, Ocsigen, Ensino, Páginas Web Interactivas.

## 1 Introdução

Dado o carácter matemático e formal dos tópicos abordados em matérias como linguagens e autómatos, o seu ensino e aprendizagem são exigentes e desafiantes. Vários estudos comprovam estas dificuldades e avaliam a utilidade de

* Trabalho parcialmente suportado pela Fundação Tezos através do projeto FACTOR (`http://www-ctp.di.fct.unl.pt/FACTOR/`) e por fundos nacionais através da FCT – Fundação para a Ciência e a Tecnologia, I.P., no âmbito do NOVA LINCS através do projeto UID/CEC/04516/2019

aplicações para estudar este tema [20, 21, 28, 33]. É importante dar apoio ao trabalho autónomo dos alunos com ferramentas interativas que permitam visualizar exemplos e fazer exercícios. No entanto, a maioria das aplicações são em Inglês e, por terem focos diferentes, nem sempre respondem a todas as necessidades. Enquanto umas são bastante completas, mas por serem *Desktop*, nem sempre estão acessíveis, outras, são de fácil acesso através do *browser*, embora estejam pouco desenvolvidas.

No contexto Português, no que diz respeito aos programas e bibliografias da maioria das disciplinas de Teoria da Computação (ou equivalentes) nas principais Universidades, verifica-se que o material é essencialmente teórico e, que apesar de aquele que é usado em sala de aula poder ser em português, a bibliografia é essencialmente em Inglês. Há, por isso, lugar para uma ferramenta interativa, em Português, que complemente o estudo teórico que já é feito hoje em dia nas aulas.

O objectivo deste projecto é desenvolver uma aplicação, em Português, que facilite o estudo da Teoria da Computação para alunos de Informática, disponível através de um *browser*. A ideia é criar uma ferramenta que possa vir a suportar todos os tópicos dentro da Teoria da Computação, como autómatos finitos deterministas e não deterministas, autómatos de pilha, linguagens regulares, linguagens independentes de contexto, linguagens LL e todas as funcionalidades inerentes as estes tópicos, como conversões, minimizações e testes. Procura-se, ainda, criar uma ferramenta extensível que permita acrescentar funcionalidades, de forma fácil e eficaz. Pretende-se, também, que esta ferramenta esteja, em primeiro lugar, adaptada à disciplina lecionada na FCT-UNL, que venha a incluir exercícios avaliados de forma automática e com *feedback* e que permita, ainda, aos alunos criar os seus próprios exercícios.

Esta plataforma está a ser desenvolvida em *Ocsigen Framework*, ferramenta que permite a criação de sistemas web interativos, totalmente escrito em *OCaml*. Ao tirar partido das características do *OCaml*, é possível obter páginas web completamente funcionais e menos sujeitas a erros. O *Ocsigen* facilita, ainda, a criação de ferramentas web, pois permite escrever cliente e servidor na mesma linguagem, facilitando a programação do sistema.

Neste documento, é apresentada a primeira versão da ferramenta e o seu desenvolvimento. Esta é, para já, uma página simples, mas com algumas funcionalidades importantes: visualização de autómatos, (exemplos disponíveis na aplicação ou criados pelo utilizador), teste de aceitação de palavra (passo a passo ou de forma animada) e verificação da natureza dos estados. Esta ferramenta pode ser acedida em `http://ctp.di.fct.unl.pt/FACTOR/OFLAT` e o código está disponível para consulta em `https://bitbucket.org/rpmacedo/oflat/src/master/`.

# 2 Trabalho Relacionado

## 2.1 Ferramentas de visualização existentes

Desde cedo se percebeu que as dificuldades de aprendizagem, e mesmo de ensino, no Estudo das Linguagens Formais e Teoria de Autómatos era um problema recorrente. E que, por ser um tema que gira em torno de processos e máquinas abstractas, a melhor forma de compreender esta problemática seria através de ferramentas pedagógicas. O desenvolvimento destas ferramentas tem sido feito desde o inicio da década de 60 [17]. O artigo *Fifty Years of Automata Simulation: A Review* [17] defende também que de já existirem muitas ferramentas a comunidade cientifica continua a receber novas pois cada uma é diferente, cada uma tem os seus próprios princípios e muitas das vezes novas utilidades. Para além disso cada ferramenta é influenciada pelas ferramentas de desenvolvimento disponíveis no momento.

Existem desde ferramentas textuais, como por exemplo [40], que permite criar e testar autómatos através de texto ou código, sem que estes sejam visíveis graficamente, e ferramentas baseadas na visualização gráfica. Algumas destas serão mais à frente analisadas, com maior profundidade, por terem semelhanças com o tema deste projecto.

No trabalho de pesquisa efectuado, foram encontrados muitos exemplos de aplicações que, de alguma forma, visam colmatar as dificuldades mencionadas pelo recurso a soluções diferenciadas. A título de exemplo, referem-se as seguintes ferramentas: *FSM Simulator* [36] um programa Java que possibilita fazer simulações com autómatos finitos; *Language Emulator* [37] uma ferramenta que permite trabalhar com diferentes conceitos dentro da Teoria de Autómatos e que tem sido usada por estudantes na Universidade de Minas Gerais no Brasil; *jFAST* [25] um software gráfico que permite o estudo de máquinas de estado finitas; *RegeXeX* [39] um sistema interactivo para estudar Expressões Regulares; *Forlan* [35] ferramenta embebida na linguagem Standard ML que permite a experimentação de linguagens formais e autómatos.

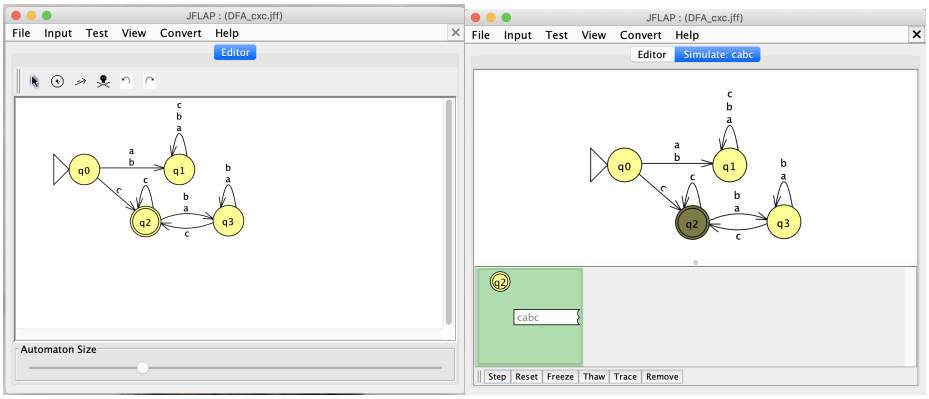
É ainda importante referir a ferramenta descrita por *Coffin et. al* em [19] por ser, provavelmente, a primeira a ser desenvolvida na área. Muitas outras são referidas no artigo *Fifty Years of Automata Simulation* [17].

Pode-se falar ainda de bibliotecas desenvolvidas para se Linguagens Formais e Autómatos como Awali [4] (evolução de Vaucanson [18,24] e Vaucanson 2 [23]) e Grail [30],escritas em c++; FAdo, escrita em Python. Awali e FAdo são também e exemplo de bibliotecas que estão a evoluir para aplicações gráficas.

Convém também referir aplicações como *TANGO* [34], *JAWAA* [27], *GUESS* [14] e *VisualAlgo* [13], por terem o objectivo de animar algoritmos ou estruturas de dados, estando relacionados com este projecto devido à sua forte componente interactiva.

Pelo que ficou dito, compreende-se que existem muitas ferramentas que poderiam ser aqui mencionadas. Decidiu-se, contudo, desenvolver um pouco mais aquelas que de alguma forma se destacam por diferentes especificidades como

JFlap por ser, provavelmente, a ferramenta mais completa disponível; Automaton simulator por ser uma ferramenta web que permite estudar AF, permitindo verificar a aceitação de frases; FSM simulator, Regular Expression Gym e FSM2Regex por serem também aplicações web e permitirem o estudo de Autómatos Finitos e Expressões Regulares e as suas conversões; Automata Tutor v2.0 por ter um sistema de avaliação e feedback; e AutoMate por ser uma ferramenta com objectivos semelhantes ao deste projecto e que se encontra em desenvolvimento ao mesmo tempo.




(a) Editor de Autómatos

(b) Página para testar passo a passo o autómato

Figura 1: JFLAP exemplos [2]

**JFlap** [8] é uma ferramenta *desktop* que se encontra em desenvolvimento desde 1990. Destaca-se por ser uma das mais completas para o estudo de Linguagens Formais e Teoria de Autómatos.

É fruto do trabalho de Susan H. Rodger e de alguns dos seus alunos que, ao longo do tempo, foram desenvolvendo novas funcionalidades [9, 10]. Apesar de ter sido inicialmente escrita em c++ e *x windows*, foi mais tarde reescrita em Java e *swing* de forma a melhorar a interface gráfica.

O código fonte está disponível na página web [8] e no GitHub, permitindo modificações por qualquer utilizador.

Este software é complementado por uma página web que contém explicações e resolução de exercícios e por um livro guia para utilização da aplicação, mas que se encontra desatualizado.

JFLAP é usado há mais de 20 anos em diversas universidades do mundo, tendo já sido testado o seu impacto positivo [26, 31, 32].

Em termos de funcionalidades, é possível trabalhar de forma interativa com Autómatos finitos (converter AFNs em AFDs, AFNs em expressões ou gramáticas

regulares, minimizar AFDs, testar se aceitam palavras e visualizar o processo de aceitação passo a passo); Máquinas de Mealy; Máquinas de Moore; Autómatos de Pilha (criação a partir de linguagens livres de contexto e vice-versa); Três tipos de Máquinas de Turing (de uma fita, de múltiplas fitas, através de blocos); Gramáticas; Sistema-L, Expressões Regulares (criar AFDs, AFNs, gramáticas regulares e expressões regulares); Lema da bombagem para Linguagens Regulares; Lema da bombagem para linguagens livre de contexto.

Apesar de todos os seus pontos positivos, *JFLAP* é uma aplicação desktop, o que significa que nem sempre está acessível, é necessário estar no computador e descarregar o software para se conseguir utilizar. Muito embora as funcionalidades tenham evoluído, o design encontra-se um pouco datado e a sua utilização nem sempre é intuitiva. É também uma aplicação não muito fácil de utilizar sendo que para os capítulos mais complexos da Teoria da Computação é necessário ler o livro de instruções para perceber como utilizar a aplicação.

**Automaton Simulator** [3] é uma ferramenta web muito simples, constituída por uma só página web (figura 2).

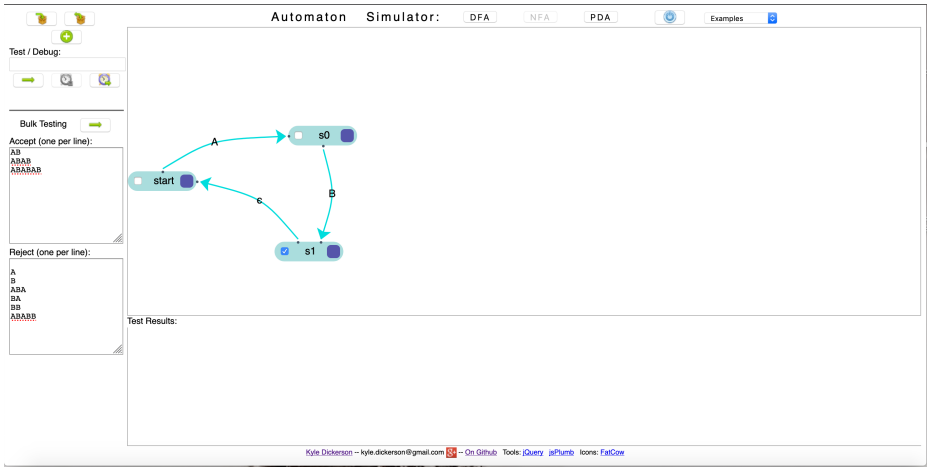



Figura 2: Página Automaton Simulator [3] com exemplo de frases para serem aceite e para serem rejeitadas

Nesta página é possível desenhar graficamente três diferentes tipos de autómatos - autómatos finitos deterministas, autómatos finitos não deterministas e autómatos de pilha. Não é possível gerar autómatos a partir de uma expressão regular, assim como também não é possível fazer a conversão de NFA em DFA.

Após a criação do autómato é possível testar a aceitação ou a rejeição de frases, bem como o reconhecimento passo a passo de uma frase pelo autómato. Contudo, só permite avançar para o passo seguinte e nunca voltar para trás.

Apesar da sua simplicidade, esta página resulta pouco intuitiva, uma vez que é desenhada à base de ícones, sem conter qualquer tipo de explicação. Além disso, contém poucas funcionalidades.

**FSM simulator [6], Regular Expressions Gym [12], FSM2Regex [7]** são três ferramentas complementares que permitem o estudo de expressões regulares e teoria de autómatos. Cada uma destas ferramentas é uma página web desenvolvida, desde 2012, por Ivan Zuzak, Web Engineer e antigo professor na universidade de Zagreb, e Vedrana Jankovic, Engenheira de Software na Google. Cada página foi desenvolvida em *Noam* - uma biblioteca *JavaScript* que permite trabalhar com máquinas de estado finitas, gramáticas e expressões regulares - , *Bootstrap, Viz.js* e *jQuery*. O código fonte está disponível no GitHub, sob a licença Apache v2.0.

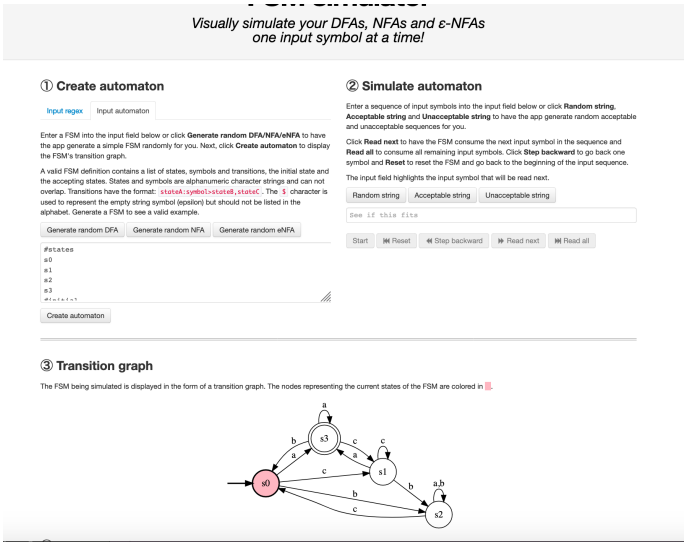



Figura 3: Página FSM Simulator [6] com exemplo de um autómato

*FSM Simulator* (figura 3) é utilizada para a criação e teste de autómatos. Os autómatos podem ser gerados através de expressões regulares ou através de texto, não sendo, no entanto, possível criá-los graficamente. É possível visualizar o processo de reconhecimento de uma frase pelo autómato, passo a passo, podendo o utilizador avançar ou retroceder, sempre que necessário.

*Regular Expressions Gym* (figura 4) é uma pagina web muito simples que permite visualizar a simplificação de uma expressão regular. O utilizador pode ver toda a simplificação de uma só vez ou pode escolher vê-la passo a passo até estar terminada. Para cada passo da simplificação, em qualquer uma das opções, é indicada a regra utilizada.

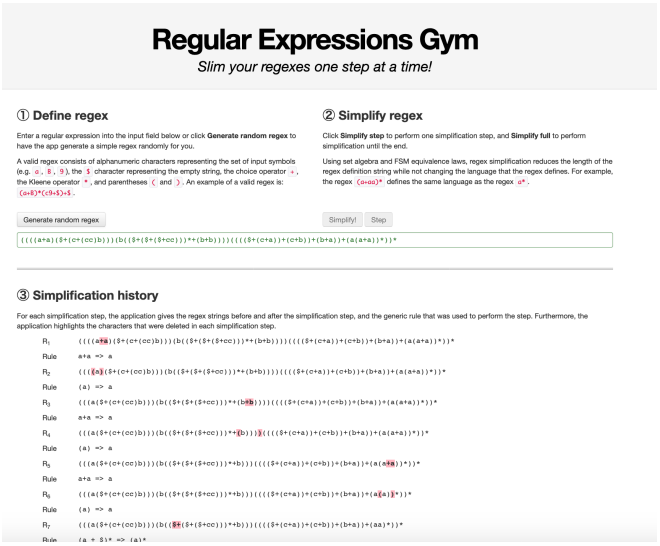



Figura 4: Página Regular Expression Gym [12] com exemplo de uma expressão a ser reduzida

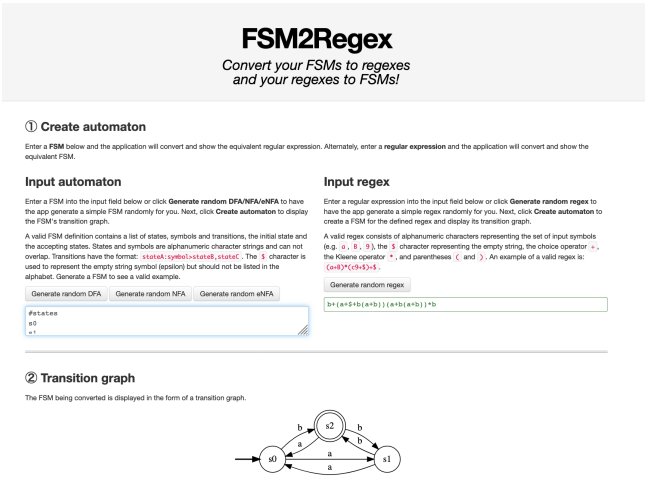



Figura 5: Página FSM2Regex [7] de um autómato e da sua correspondente expressão regular

*FSM2Regex* (figura 5) é utilizada para fazer a conversão de uma expressão regular num autómato e vice versa, não permitindo nenhum outro tipo de interacção.

As três páginas abordadas correspondem a ferramentas bastante simples e intuitivas, mas seriam, provavelmente, mais proveitosas se as funcionalidades de cada uma delas fossem integradas numa só página. Dessa forma seria também mais fácil o desenvolvimento de novas funcionalidades.

**Automata Tutor v2.0** [1,22] é uma ferramenta web que se destaca por fornecer um sistema de avaliação de exercícios de autómatos finitos e expressões regulares, facilitando o trabalho dos professores. Esta ferramenta passou por três fases de desenvolvimento e vários testes por utilizadores [21, 22], com o objectivo de melhorar a aplicação.

A página permite o registo e entrada no sistema com dois tipos de perfis: o de professor, ao qual é dada a possibilidade de criar um curso com exercícios próprios e visualizar as notas no final do curso; e o de aluno, que se pode inscrever num determinado curso ou então fazer os exercícios disponíveis na própria página. O foco principal desta ferramenta é a resolução de exercícios de criação de expressões regulares e autómatos finitos correspondentes a uma frase dada em Inglês. Quando um aluno submete uma solução, recebe como resposta, além da nota, alguns comentários, que lhe permite proceder a correcções, caso seja necessário. Apesar de ser uma aplicação muito completa quanto à avaliação de exercícios, esta foca-se somente da entrega de feedback, não dando liberdade de experimentar a resolução de forma autónoma para compreender porque está certo ou errado (como por exemplo a verificação passo a passo da aceitação da palavra).

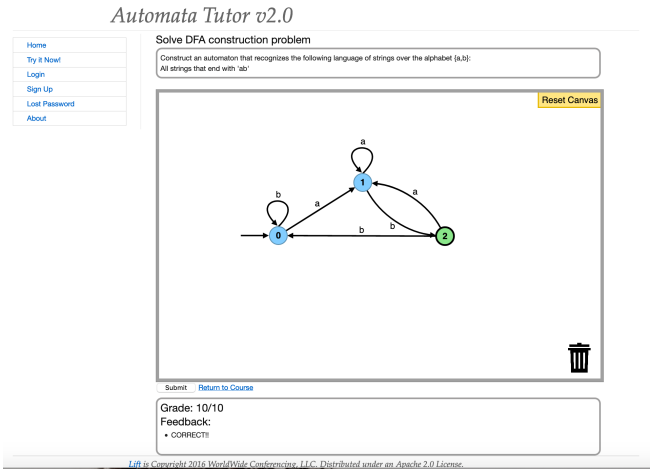
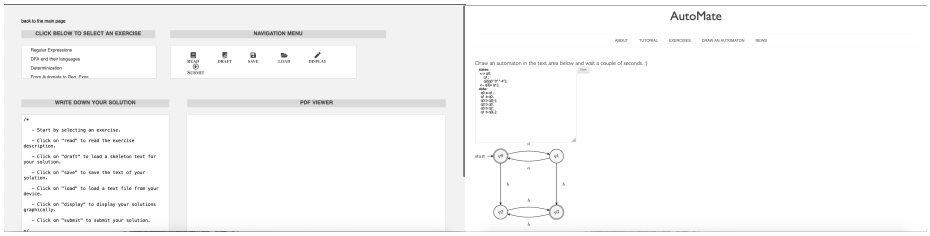



Figura 6: Exemplo da resolução de um exercicio DFA no AutomataTutor [1]

**AutoMate** [2] é uma ferramenta web muito recente, lançada em 2019 e encontrando-se ainda numa fase muito inicial. É, no entanto, relevante falar dela, uma vez que se encontra simultaneamente em desenvolvimento com a proposta deste projecto. Tem como finalidade apoiar o estudo de alunos de Informática, ou seja, pretende ajudar a resolver exercícios sobre autómatos finitos e expressões regulares.

Em termos de funcionalidades esta ferramenta é, para já, bastante simples, centrando-se principalmente na verificação e correcção de exercícios (figura 7a). Todo o processo de realização de exercícios é feito através de texto, não sendo possível a criação de autómatos graficamente.

A ferramenta permite realizar exercícios de diferentes temas dentro da Teoria da Computação - Expressões Regulares, criação de DFAs, transformação NFAs em DFAs, passagem de DFAs para expressões regulares. Após a entrega da resolução dos exercícios é devolvido um *feedback* automático, em formato pdf, relevante para que o aluno.




(a) Página para ver e resolver exercícios (b) Página para desenhar autómatos

Figura 7: Exemplo de duas páginas da Ferramenta Web Automate [2]

Esta ferramenta contém ainda uma página onde se pode desenhar um autómato (ver figura 7b), através de texto, de forma a que seja possível visualizá-lo através de uma imagem. Não é no entanto possível a verificação da aceitação de uma palavra no autómato criado.

Em termos de design esta ferramenta é ainda muito simples e rudimentar (figura 7, sendo baseada em caixas de inserção de informação.

Todas as ferramentas mencionadas têm características que, de alguma forma, são comuns ao projecto em desenvolvimento mas não foi encontrada nenhuma que permitisse aos alunos criar os seus próprios exercícios de forma a receber feedback da solução.

### 2.2 Ambientes de desenvolvimento para a Web

Tipicamente, uma ferramenta web pressupõe dois componentes: cliente e servidor. Estes são normalmente desenvolvidos em linguagens diferentes e, por tal, para que os dados possam ser partilhados entre ambos é também necessário escrevê-los num formato pré-estabelecido.

O lado do cliente é a parte da aplicação web com a qual o utilizador interage, sendo que a cada interação é enviado um pedido ao servidor, que responde com a execução de uma ação ou uma informações da base de dados. Este é maioritariamente desenvolvido pelo recurso a três linguagens: HTML, CSS, *JavaScript*.

O lado do servidor é a parte da aplicação web que indica como esta funciona. O servidor recebe pedidos do cliente e retorna a informação pedida ou a ação que pode ser realizada. O servidor pode ser escrito em diferentes tipos de linguagens como C, C++, C#, PHP, *Python*, *Java* ou até *JavaScript*.

Para que o cliente possa comunicar com o servidor é utilizado um protocolo de comunicação, como HTTP, e a mensagem deve estar num formato convencionado, sendo os mais comuns HTML, XML ou JSON.

Com a necessidade de criação de páginas web mais dinâmicas, e devido ao facto de ser necessária a aprendizagem de muitas linguagens e componentes, começaram a surgir diferentes *frameworks* e bibliotecas com o objectivo de facilitar o trabalho do programador. Alguns visam facilitar o desenvolvimento do design da página, como o *Bootstrap*, cujo o objectivo é criar páginas web *responsive*. Outros pretendem facilitar a criação de aplicações web de uma só página, isto é, reactivas. Temos, como exemplos, *AngularJS*, um framework que estende a linguagem *HTML*; e *React*, uma biblioteca *JavaScript*. De referir ainda *jQuery* uma biblioteca *JavaScript* que pretende simplificar a escrita das *queries* do mesmo e *frameworks* que visam facilitar o desenvolvimento de servidores com muitos acessos à base de dados e que pressupõem reutilização de código, como *Django* e *Ruby on Rails*.

Apesar do surgimento de tantos *frameworks* e bibliotecas, o programador, para criar páginas web, necessita sempre de saber pelo menos *HTML, CSS, JavaScript* (ou um correspondente) e uma linguagem para o servidor.

## 3 Ocsigen Framework

*Ocsigen Framework* é uma ferramenta muito completa que se destaca principalmente por permitir a criação de páginas web interativas, escritas totalmente em *OCaml*. Este *Famework* surge da ideia de que a programação funcional é uma solução elegante para alguns problemas de interação nas páginas web [16]. Vem tentar responder aos novos desafios das páginas web, isto é, à necessidade de estas se comportarem cada vez mais como aplicações [15]. Uma das grandes vantagens é compilar o código *OCaml* do cliente para *JavaScript*, o que permite trabalhar em conjunto com esta linguagem e, assim, utilizar um amplo número de bibliotecas, que de outra forma não estariam disponíveis.

### 3.1 Descrição das componentes

*Ocsigen Framework* é na realidade um conjunto de diferentes componentes, o que traduz a complexidade desta ferramenta.

**Eliom** é o componente principal do *Ocsigen*, é uma extensão do *OCaml* para programação sem camadas (*Tierless*) [29]. *Eliom* pretende ser um novo estilo

de programação que se enquadra nas necessidades das aplicações web modernas melhor do que as linguagens de programação usuais (desenhadas há muitos anos, para páginas muito mais estáticas [11]). O seu grande objectivo é permitir desenvolver uma aplicação distribuída, totalmente em *OCaml* e como um só programa [11], ou seja, não haver separação entre cliente e servidor. Para que isto seja possível existe uma sintaxe especial para distinguir os dois. O cliente pode aceder facilmente a variáveis do servidor, pois o sistema de suporte implementa esse mecanismo de forma transparente. Outra importante vantagem resulta do uso da tipificação estática do Ocaml, possibilitando verificar erros e bugs no momento da compilação, havendo a certeza de que se obtêm páginas web corretas, sem problemas nos links ou na comunicação cliente-servidor. Além disso, o *Eliom* resolve automaticamente problemas de segurança frequentes em páginas web. O *Eliom* tem ainda, por base o pressuposto de que se escreve código complexo em poucas linhas. As aplicações desenvolvidas nesta ferramenta correm em qualquer *browser* ou dispositivo móvel, não havendo necessidade de customização.

**Js_of_ocaml** é o compilador de *OCaml bytecode* para *JavaScript*. É o componente do *Ocsigen* que permite correr a aplicação escrita em *OCaml* em ambientes *JavaScript* como os *browsers* [11] [38]. Possibilita a integração de código *JavaScript* no programa *OCaml*.

**Lwt** é a biblioteca de *threads* cooperativa para *OCaml* que permite lidar com problemas de concorrência de dados e de *deadlocks*. É a forma standard de construir aplicações concorrentes. Permite fazer pedidos de forma assíncrona, através de promessas. Estas são simplesmente referências que vão ser preenchidas assincronamente, aquando da chegada da resposta.

**Tyxml** é a biblioteca que permite a construção de documentos HTML estaticamente corretos. Tyxml providencia um conjunto de combinadores, que utilizam o sistema de tipos do OCaml para confirmar a validade do documento gerado.

**Ocsigen-start** é a biblioteca e *template* de uma aplicação *Eliom* com muitos componentes típicos das páginas web, que visa facilitar a contrução de aplicações web interativas.

**Ocsigen-toolkit** é a biblioteca de *widgets* que, tal como o *Ocsigen-start*, visa facilitar o desenvolvimento rápido de aplicações web interativas.

## 4 Animação e visualização de Autómatos

O objectivo deste projecto é a criação de uma ferramenta web que venha a permitir a alunos de informática estudar os vários temas dentro da Teoria de Computação de forma intuitiva e eficaz. Pretende-se que esta ferramenta permita ao aluno trabalhar com Autómatos finitos (minimização, conversão de autómatos não deterministas em deterministas, minimização, conversão em expressões regulares, verificação de aceitação de palavras e geração de palavras de tamanho x), Expressões regulares (simplificação e conversão em AFN), linguagens não regulares, linguagens LL, linguagens independentes de contexto (lema da bombagem e conversão em autómatos de pilha), Autómatos de pilha (conversão em linguagens independentes de contexto) e máquinas de Turing. Para além disso

pretende-se ter um sistema de resolução de exercícios com entrega de feedback no qual o aluno pode fazer upload dos seus próprios exercícios.

Este projecto encontra-se agora em desenvolvimento e nesta secção apresentamos e explicamos as funcionalidades disponíveis da primeira versão da aplicação, que pode ser acedida em `http://ctp.di.fct.unl.pt/FACTOR/OFLAT`. O código completo pode ser consultado em `https://bitbucket.org/rpmacedo/oflat/src/master/`.

Ao longo do processo de criação desta versão surgiram alguns desafios que fizeram a utilização do *framework* um pouco complicada. Muito possivelmente devido à mudança de sintaxe do *OCaml* (a sintaxe *camlp4* foi descontinuada para dar lugar a *ppx*), o *Ocsigen* necessitou de ser modificado e nos exemplos de utilização ainda existem algumas inconsistências. Apesar de tudo, no final, conseguiu-se obter uma solução elegante e eficiente que comprova as características do *framework*.

Como este *framework* permite o trabalho conjunto entre *OCaml* e *JavaScript*, para a realização da parte gráfica optou-se pela utilização da biblioteca *JavaScript* de visualização de grafos *Cytoscape.js* [5], por ser muito completa, com várias opções de manipulação e edição dos grafos. Todo o trabalho de verificação ou edição dos grafos é realizado em *Eliom*, sendo a biblioteca utilizada unicamente para representar o grafo, ou seja, é feita uma separação entre o lado lógico e o lado gráfico da aplicação.

### 4.1 Visão geral

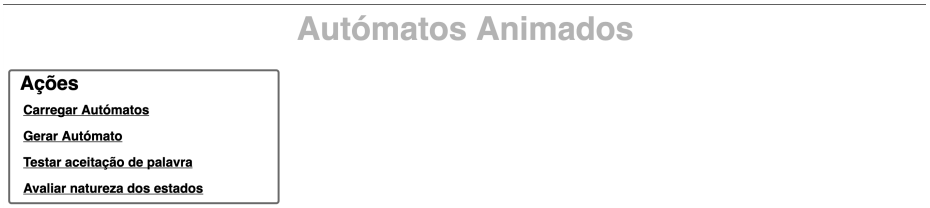



Figura 8: Entrada da Aplicação

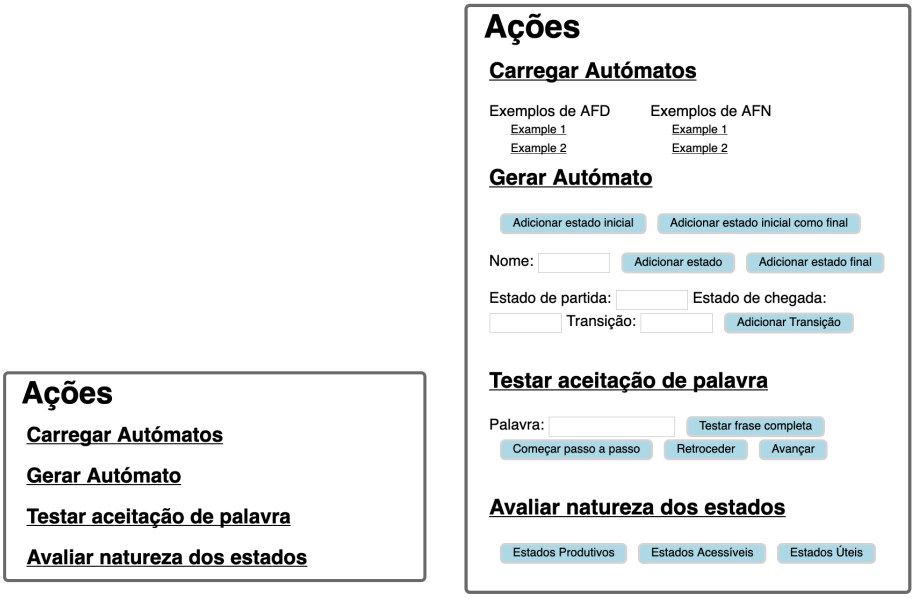
Para se criar a página é necessária a criação de um módulo onde se regista a aplicação e de um serviço que indica o caminho ou *link* para a página principal

e, por fim, o registo do serviço no módulo criado, obtendo-se, assim, uma página principal. É no registo da página que se definem os elementos da mesma.

```
module Finalexample_app =
  Eliom_registration.App (
    struct
      let application_name = "finalexample"
      let global_data_path = None
    end)
let main_service =
    Eliom_service.create
    ~path:(Eliom_service.Path [])
    ~meth:(Eliom_service.Get Eliom_parameter.unit)
    ()
let () =
  Finalexample_app.register
    ~service:main_service
    (fun () () →
      let open Eliom_content.Html.D in
       Lwt.return
         (html
            (head (title (txt "Autómatos Animados"))
                [script ~a:[a_src script_uri1] (txt "");
                 script ~a:[a_src script_uri3] (txt "");
                 script ~a:[a_src script_uri5] (txt "");
                 script ~a:[a_src script_uri4] (txt "");
                 script ~a:[a_src script_uri6] (txt "");
                 script ~a:[a_src script_uri2] (txt "");
                 css_link ~uri: (make_uri (Eliom_service.static_dir ())
                                          ["codecss2.css"]) ();
                 script ~a:[a_src script_uri] (txt "");
            ])
            (body [ div [h1 [txt "Autómatos Animados"]];
                    div ~a:[a_id "inputBox"]
                           [h2 [txt "Ações"];
                            mywidget "Carregar Autómatos"
                                     (hiddenBox2 upload);
                            mywidget "Gerar Autómato"
                                     (hiddenBox2 generate);
                            mywidget "Testar aceitação de palavra"
                                     (hiddenBox2 verify);
                            mywidget "Avaliar natureza dos estados"
                                     (hiddenBox2 evaluate);
                           ];
                    div ~a:[a_id "cy"] [];
                    ]
            )))
```

Quando se inicia a aplicação vê-se uma página simples (Figura 8) que contém um menu do lado esquerdo e um espaço em branco do lado direito. Os autómatos,

quando gerados, vão aparecer no lado direito do ecrã. No menu (Figuras 9a) são visíveis as opções disponíveis para se interagir com autómatos: carregar autómatos, gerar autómatos, testar aceitação de palavra e verificar a natureza dos estados. Ao clicar, em cada uma das opções principais, é possível visualizar as sub-opções disponíveis (Figura 9b). Cada chamada a mywidgets no código acima é a criação dos sub-menus disponiveis.




(a) Menu principal (b) Menu principal com opções

Figura 9: Vista do menu

### 4.2 Carregar Autómatos

Na opção "Carregar Autómatos"são disponibilizados diferentes exemplos pré-definidos de autómatos. Para a resolução desta etapa foi necessário responder a diferentes questões, nomeadamente: Como aceder à biblioteca *Cytoscape.js* programando em *Ocsigen*? Como chamar funções do código *JavaScript* em *Ocsigen*? Como transformar um autómato formatado em *OCaml* no grafo *Javascript*?

**Como aceder à biblioteca *Cytoscape.js* programando em *Ocsigen*?** Para responder a esta pergunta era necessário responder a duas subquestões: Como chamar *scripts* ao iniciar a página (para se aceder a biblioteca durante a execução)? Como criar links?

A pesquisa desenvolvida permitiu perceber que, no momento de registo do serviço principal, é possível chamar diferentes tipos de *scripts* (visto no código da

subsecção anterior) aquando do desenho da página. Era necessário ainda perceber como criar links para páginas externas. Após alguma pesquisa, encontrou-se alguns exemplos na página web da ferramenta que possibilitaram perceber como criar os links:

```
let script_uri1 =
  Eliom_content.Html.D.make_uri
  (Eliom_service.extern
     ~prefix:"https://unpkg.com"
     ~path:["cytoscape";"dist"]
     ~meth:(Eliom_service.Get Eliom_parameter.(suffix (all_suffix "suff")
    ))())
    ["cytoscape.min.js"]
```

**Como chamar funções do código *JavaScript* em *Ocsigen*?** Responder a esta questão acabou por não ser tarefa fácil, pois a API encontrada na página web do framework apesar de ser muito completa contém pouca informação em relação à comunicação entre código OCaml e código JavaScript. Com alguma pesquisa percebeu-se que tal como se chama o link da biblioteca, é possível chamar o link do código *JavaScript*. Em seguida, para chamar as funções JavaScript é necessário usar outras duas funções *OCaml*:

```
let js_eval s = Js_of_ocaml.Js.Unsafe.eval_string s
let js_run s = ignore (js_eval s)
```

Aquando da necessidade de aceder ao código *JavaScript* chama-se a função *js_run* com o nome da função *JavaScript* entre aspas.

```
    js_run ("makeNode1('"^f^"', '"^string_of_bool (test)^"')");
```

*js_run*, por sua vez, chamará a função *js_eval*, como se pode ver pelo código acima.

**Como transformar um autómato formatado em *OCaml* no grafo *JavaScript*?** Para se conseguir representar graficamente qualquer tipo de autómato sem que seja necessário este estar escrito tanto no código *OCaml* como no *JavaScript* foi preciso criar várias funções que fizessem a passagem da formatação *OCaml* para o *JavaScript*. Desta forma não há repetição de código, uma vez que os autómatos estão apenas representados no programa *OCaml*.

```
let example1DFA = {initialState = "START";
                   transitions = [("START", 'a', "A"); ("A", 'b', "B");
    ("B", 'a', "C"); ("C", 'b', "B"); ("C", 'a', "A")];
                   acceptStates = ["START"; "B"; "C"]}
```

A função, acima representada, demonstra a definição de um autómato no programa *OCaml*. Cada autómato é composto pelo estado inicial, transições e estados finais. A diferença para a biblioteca *Cytoscape.js* é que, nesta, são compostos só por estados e transições, sendo que as transições dependem dos estados.

A análise do código explicativo de como gerar graficamente os autómatos será feito de uma forma *down-top*, porque, no *OCaml* funções chamadas por outra devem ser colocadas acima desta.

```
let createNode (f, s, t) =
    let test = last1 f in
    js_run ("makeNode1('"^f^"', '"^string_of_bool (test)^"')");
    let test = last1 t in
    js_run ("makeNode1('"^t^ "', '"^string_of_bool (test)^"')")
let createEdge (f, s, t) =
    js_run ("makeEdge('"^f^"', '"^t^"', '"^(String.make 1 s)^"')");
let rec findtransitions (trans) =
    match trans with
        [] → false
        | x::xs → createNode(x); createEdge (x); findtransitions (xs)
let generateAutomata (gra) =
    automata := gra;
    js_run ("start()");
    ignore (findtransitions (gra.transitions))
```

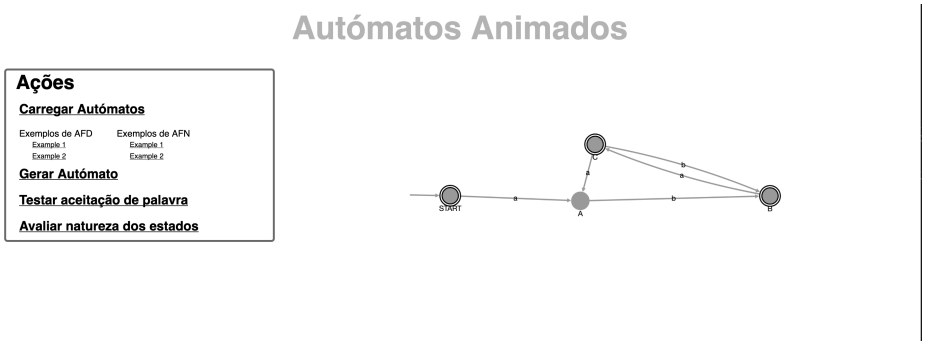
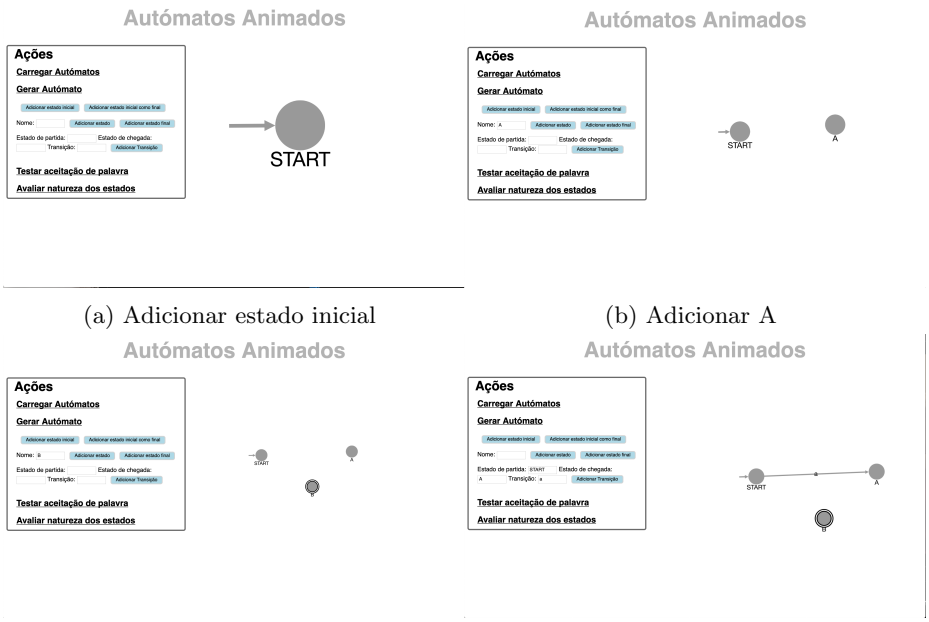



Figura 10: Carregar Autómatos - Exemplo1 de AFD

*generateAutomata* é a função chamada para iniciar a criação do autómato. Em primeiro lugar, define a referência *automata* como o autómato a ser gerado. Esta referência serve para que todos as outras funções possam saber qual o autómato representado na página (importante para as funções explicadas nas subsecções seguintes). Em seguida, é chamada a função *start()* do *JavaScript*, que define o tipo de grafo e as suas características principais. Por fim, é chamada a função *findtransitions*, função recursiva que, para cada transição em *Ocaml*, manda criar os estados e a transição correspondentes no *JavaScript*. Terminado este processo de criação do autómato pode, então, visualizar-se um grafo como o da Figura 10, que corresponde à representação *OCaml* anteriormente definida.

### 4.3 Gerar Autómatos

A opção surge no seguimento da anterior, com o objetivo de permitir ao utilizador criar um autómato do zero (exemplo, Figura 11) ou acrescentar estados




(a) Adicionar estado inicial (b) Adicionar A

(c) Adicionar B (d) Adicionar Transição

Figura 11: Geração de um Autómato

e transições aos autómatos carregados. Posto isto, a questão que se coloca é: Como permitir a criação de Autómatos por etapas?

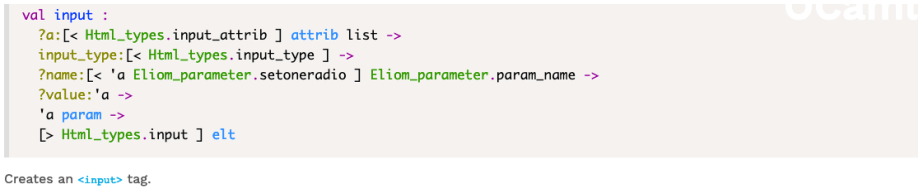
Para a resolução deste problema foi necessário criar vários botões e caixas de input, para que o utilizador possa identificar os estados e as transições. Para criar um estado é necessário dar input do nome do mesmo e para e para criar uma transição é necessário indicar o estado de partida, o estado de chegada e o símbolo de transição.

Esta fase não foi fácil pois, apesar do framework ter online uma API muito completa esta, por falta de explicações e de exemplos, não é fácil de compreender tornando-se pouco pedagógica.

A título de exemplo veja-se a Figura 12a, que mostra o ponto da API, que demonstra como criar uma caixa de *input*. Para se perceber a informação necessária em cada componente é preciso fazer uma nova pesquisa, de forma a compreender como ela é declarada (Figura 12b). Com a falta de exemplos, nem sempre é fácil encontrar a resposta.

Após alguma investigação e experimentação foi possível obter-se o código funcional para a criação de caixas de input e botões como se pode ver no código seguinte:

```
let input1 = input ~a:[a_id "box"; a_input_type `Text]() in
let onclick_handler2 = [\%client (fun _ →
    let i = (Eliom_content.Html.To_dom.of_input ~\%input1) in
```




(a) API

```
let input0 = input ~a:[a_input_type `Text]()
```

(b) Código caixa de input

Figura 12: API e código de uma caixa de input

```
    let v = Js_of_ocaml.Js.to_string i##.value in
    js_run ("makeNode1('" ^ v ^ "', '" ^ string_of_bool (false) ^ "')")
  )] in
  let button2 = button ~a:[a_onclick onclick_handler2] [txt "Adicionar
    estado"]
```

No código apresentado pode visualizar-se uma característica importante do *framework*, a chamada de código de cliente em código de servidor. As duas funções apresentadas fazem parte do servidor, pois este é que cria os a página web, mas chama código de cliente (representado por [%cliente]) pois este têm de estar disponível durante todo o funcionamento da página.

Acrescentar estados ou transições ao autómato gerado estava, em parte, resolvido, uma vez que para o desenhar graficamente era só necessário chamar as funções *JavaScript*, já criadas para o efeito. Como a representação *OCaml* do autómato não pressupõe a representação de estados, o botão para os gerar implica apenas chamar a função *makeNode* do *JavaScript*. Para o autómato ficar atualizado também no *OCaml* criou-se uma função que acrescentasse transições à representação do mesmo.

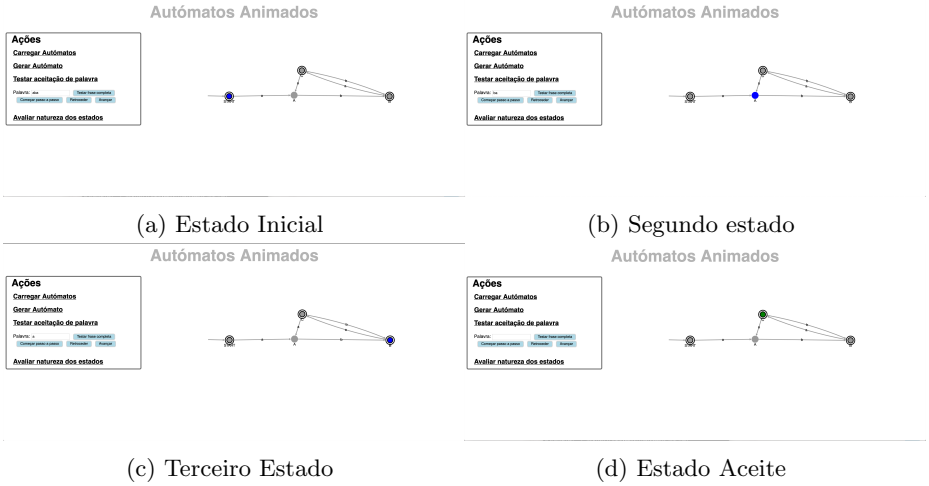
```
let newNode(c1,c2,c3) = {initialState = !automata.initialState;
                         transitions = !automata.transitions@[(c1,c2,c3)];
                         acceptStates = !automata.acceptStates}
```

Para gerar um autómato novo, criado de raiz, foi necessário criar um botão que inicializasse a biblioteca e colocasse o estado de partida na página. Este botão foi mais tarde duplicado para permitir que o estado de partida pudesse ser também final.

Por fim, para a geração de estados foi também criado um novo botão para identificar o estado como final. Este botão pressupôs a criação de uma função que o acrescentasse à representação *OCaml* como estado final.

```
let newNodeFinal final = {initialState = !automata.initialState;
                          transitions = !automata.transitions;
                          acceptStates = !automata.acceptStates@[final]}
```

### 4.4 Testar palavras




(a) Estado Inicial (b) Segundo estado

(c) Terceiro Estado (d) Estado Aceite

Figura 13: Verificação da aceitação da palavra aba - palavra aceite


(a) Quarto estado (b) Estado final: Erro

Figura 14: Verificação da aceitação da palavra abaa - palavra não aceite (começo igual a Figuras 13a, 13b, 13c)

A opção testar palavras foi a de desenvolvimento mais desafiante e, por enquanto, funciona só para AFDs e alguns AFNs. Pode visualizar-se um exemplo da aceitação da palavra nas Figuras 13 e 14. A ideia base é simples: o utilizador faz input de uma frase que quer testar e, escolhendo uma das duas opções, terá oportunidade de ver os estados a mudar de cor: azul, se é um estado de passagem, vermelho, se chegou ao fim da palavra mas não ficou num estado final (palavra não aceite) ou verde, se a palavra chegou ao fim e o estado é final (palavra aceite). Partindo do exemplo falado na Subsecção anterior, foi possível criar uma caixa de input e um botão que lê a informação da caixa. Foi, em seguida,

necessário responder a três perguntas: Como editar o autómato? Como animar o autómato? E como fazer a animação passo a passo?

**Como editar o autómato?** Como referido, a edição do autómato é feita no código *Javascript*, mas a lógica da aplicação está no *OCaml*. Assim, o importante era perceber como editar o estilo do grafo (problema resolvido com o estudo da biblioteca) e quais os dados que devem ser passados aquando da chamada da função *JavaScript*. A informação necessária é o estado para mudar a cor, o ponto da palavra em que nos encontramos e se o estado é final.

**Como animar o autómato?** Para que o utilizador tenha a possibilidade de visualizar a passagem dos estados é necessário animar o autómato gerado. Criou-se, assim, uma função que retorna um estado ao qual se chega com o símbolo dado, partindo do estado atual. Quando um novo estado é encontrado, este é colorido, chamando a função *JavaScript* para o efeito. Após esta construção percebeu-se que apenas o estado final era colorido.

Para dar tempo à página de modificar o autómato recorreu-se à biblioteca *OCaml Unix*, que contém uma função *sleep*. Problema: a função está escrita na parte de cliente e a biblioteca só funciona no servidor. Depois de alguma pesquisa percebeu-se que a solução seria utilizar a biblioteca de *threads Lwt* e uma função de *delay*. Obteve-se o seguinte:

```
let rec delay  n = if n = 0 then Lwt.return () else Lwt.bind (Lwt.pause
    ()) (fun () → delay (n-1))
let rec acceptX s w fa =
  let last = last1 s in
  js_run ("changeColor1('"^s^"', '"^(string_of_int (List.length w))^"',
    '"^ string_of_bool (last)^"')");
  match w with
        [] → Lwt.return (List.mem s fa.acceptStates)
        | x::xs → match transitionsFor (s,x) fa with
           [] → Lwt.return (js_run ("changeColor1('"^s^"', '"^(
    string_of_int (0))^"', '"^string_of_bool (false)^"')"); false)
        | (_,_,s)::_ → Lwt.bind (delay 50)
                        (fun () → Lwt.bind (Lwt.return (let last3 =
    last1 s in js_run ("changeColor1('"^s^"', '"^(string_of_int (List.
    length xs))^"', '"^string_of_bool (last3)^"')"))) (fun () → acceptX
    s xs fa))
```

O que acontece é que cada vez que se encontra uma transição, é executado um *delay* onde é usada a função *Lwt* de pausa que, trata os eventos pendentes, incluindo os de atualização do ecrã.

**E como fazer a animação passo a passo?** Para se desenvolver a opção passo a passo foi necessário criar três novas variáveis imperativas: *position, step* e *sentence*. *Position* indica em que ponto da palavra nos encontramos: quando se anda para a frente esta posição é incrementada, verificando-se o oposto quando se anda para trás. *Step* indica o último estado por onde se passou e *sentence* representa a lista de símbolos presentes na palavra.

Existe um botão para começar a iteração, que pinta o estado inicial, e inicializa as variáveis mencionadas.

Partindo da função recursiva de animação do autómato, foi possível, agora sem a recursividade e sem a utilização da biblioteca *Lwt*, criar uma função que retorna o estado seguinte a um dado estado (*step*) consoante o símbolo em dada posição (*position*) da palavra a ser testada (*sentence*). Conseguimos, assim, a função de avançar.

Para realizar a função de retroceder, foi necessário fazer uma modificação à função de inicialização. Esta última passa a chamar outra que cria uma lista ordenada de todos os estados que vão ser coloridos. Assim, quando se clica no botão "retroceder", a posição (variável *position*) é decrementada e o estado correspondente à mesma na lista é colorido.

```
let acceptStepBack w =
  if (!position > 0) then
    (position := !position - 1;
    let st = get_nth !listStates !position in
    let pos = List.length w - !position in
    let isFinal = last1 st in
    js_run ("changeColor1('"^st^"', '"^string_of_int (pos)^"', '"^
    string_of_bool (isFinal)^"')"); step := setStep st)
```

É importante referir que aquando do clique nos botões "avançar"ou "retroceder"é chamada uma função que atualiza a frase na caixa de input para a situação em que o utilizador se encontra.

### 4.5 Verificar Natureza

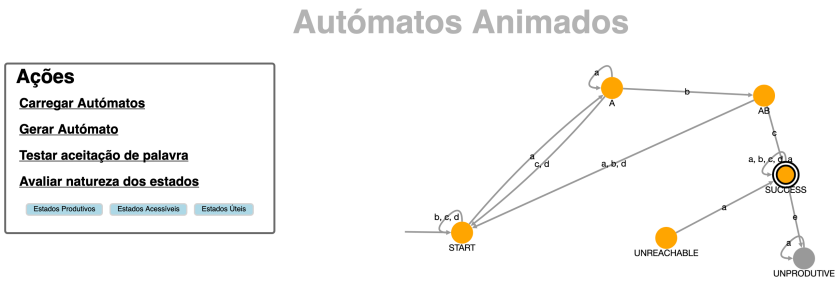



Figura 15: Indicação dos estados produtivos

Esta opção pretende que o utilizador perceba se os estados são produtivos, acessíveis ou úteis. Para cada uma das opções existe um botão que assinala, em simultâneo, todos os estados que correspondem a essa característica. A questão que surgiu foi: Como editar graficamente um estado sem alterar os outros? Anteriormente era feito um *reset* do estilo do autómato a cada afectação, neste caso foi necessária uma pesquisa na API da biblioteca para perceber como fazer a sua

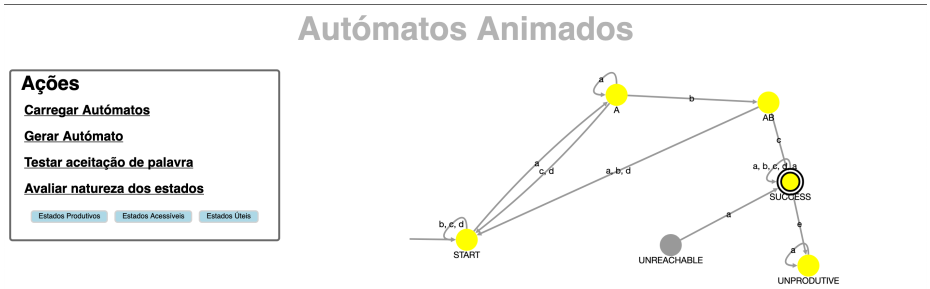



Figura 16: Indicação dos estados acessíveis

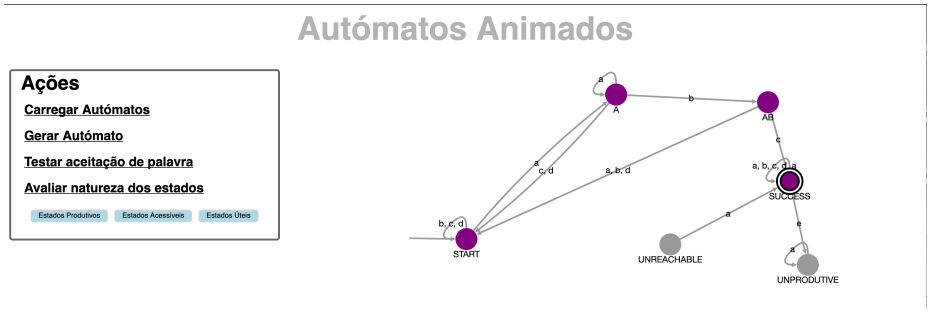



Figura 17: Indicação dos estados úteis

atualização. Em JavaScript, para cada uma das opções, foi criada uma função que colorisse um dado nó sem modificar os outros. Em *OCaml* criaram-se diferentes funções, uma para encontrar os estado produtivos, outra para encontrar os estados acessíveis e por fim uma que intersectasse os dois resultados (estados úteis). Ao selecionar uma das opções é chamada a função *OCaml* correspondente, sendo que cada estado obtido é colorido através da função *JavaScript* respetiva. Nas Figuras 15, 16 e 17 é podem ver-se exemplos de cada uma das opções.

## 5 Conclusões e Trabalhos Futuros

Utilizando o *framework Ocsigen*, está-se a desenvolver uma aplicação web interactiva para apoiar o ensino de Linguagens Formais e Autómatos. A vantagem do *Ocsigen* é permitir desenvolver de forma integrada tando a parte do servidor como do cliente, numa linguagem concisa e segura como o *OCaml*.

Um aspecto central da plataforma é ser extensível; pretende-se incluir mais funcionalidades e outras classes de linguagens, bem como integrar com suporte à avaliação, nomeadamente permitindo a submissão e classificação de exercícios.

## Referências


1. Automata tutor v2.0. `http://automatatutor.com/index`, accessed: 2019-02-11
2. Automate. `https://idea.nguyen.vg/~leon/automata_experiments/index.html`, accessed: 2019-02-11
3. Automaton simulator. `http://automatonsimulator.com/`, accessed: 2019-02-11
4. Awali. `http://vaucanson-project.org/Awali/index.html`, accessed: 2019-07-09
5. Cytoscape.js. `http://js.cytoscape.org/`, accessed: 2019-06-04
6. Fsm simulator. `http://ivanzuzak.info/noam/webapps/fsm_simulator/`, accessed: 2019-02-11
7. Fsm2regex. `http://ivanzuzak.info/noam/webapps/fsm2regex/`, accessed: 2019-02-14
8. Jflap. `http://www.jflap.org/`, accessed: 2019-02-11
9. Jflap history. `http://www.jflap.org/history.html`, accessed: 2019-02-17
10. Jflap history powerpoint. `https://www2.cs.duke.edu/csed/jflapworkshop/sigcse06/WorkshopHistory.pdf`, accessed: 2019-02-17
11. Ocsigen - multi-tier programming for web and mobile apps. `https://ocsigen.org/home/intro.html`, accessed: 2019-02-11
12. Regular expressions gym. `http://ivanzuzak.info/noam/webapps/regex_simplifier/`, accessed: 2019-02-14
13. Visualgo. `https://visualgo.net/en`, accessed: 2019-02-11
14. Adar, E.: Guess: A language and interface for graph exploration. vol. 2, pp. 791–800 (01 2006). https://doi.org/10.1145/1124772.1124889
15. Balat, V.: Ocsigen: Typing web interaction with objective caml. vol. 2006, pp. 84–94 (09 2006). https://doi.org/10.1145/1159876.1159889
16. Balat, V., Vouillon, J., Yakobowski, B.: Experience report: Ocsigen, a web programming framework. vol. 44, pp. 311–316 (09 2009). https://doi.org/10.1145/1596550.1596595

17. Chakraborty, P., C. Saxena, P., Katti, C.: Fifty years of automata simulation: A review. ACM Inroads **2** (12 2011). https://doi.org/10.1145/2038876.2038893
18. Claveirole, T., Lombardy, S., O'Connor, S., Pouchet, L.N., Sakarovitch, J.: Inside vaucanson. pp. 116–128 (01 2005)
19. Coffin, R.W., Goheen, H.E., Stahl, W.R.: Simulation of a turing machine on a digital computer. In: Proceedings of the November 12-14, 1963, Fall Joint Computer Conference. pp. 35–43. AFIPS '63 (Fall), ACM, New York, NY, USA (1963). https://doi.org/10.1145/1463822.1463827, http://doi.acm.org/10.1145/1463822.1463827
20. Cogliati, J., W. Goosey, F., T. Grinder, M., A. Pascoe, B., Ross, R., J. Williams, C.: Realizing the promise of visualization in the theory of computing. ACM Journal of Educational Resources in Computing **5**, 1–17 (06 2005). https://doi.org/10.1145/1141904.1141909
21. D'antoni, L., Kini, D., Alur, R., Gulwani, S., Viswanathan, M., Hartmann, B.: How can automatic feedback help students construct automata? ACM Trans. Comput.-Hum. Interact. **22**(2), 9:1–9:24 (Mar 2015). https://doi.org/10.1145/2723163, http://doi.acm.org/10.1145/2723163
22. D'Antoni, L., Weaver, M., Weinert, A., Alur, R.: Automata tutor and what we learned from building an online teaching tool. Bulletin of the European Association for Computer Science **117**, 143–160 (Oct 2015)
23. Demaille, A., Duret-Lutz, A., Lombardy, S., Sakarovitch, J.: Implementation concepts in vaucanson 2. vol. 7982, pp. 122–133 (07 2013). https://doi.org/10.1007/978-3-642-39274-0_12
24. Lombardy, S., Poss, R., Régis-Gianas, Y., Sakarovitch, J.: Introducing vaucanson. vol. 2759, pp. 107–134 (06 2003). https://doi.org/10.1007/3-540-45089-0_10
25. M. White, T., Way, T.: jfast: a java finite automata simulator. vol. 38, pp. 384–388 (03 2006). https://doi.org/10.1145/1124706.1121460
26. Merceron, A., Yacef, K.: Web-based learning tools: storing usage data makes a difference. pp. 104–109 (03 2007)
27. Pierson, W.C., Rodger, S.H.: Web-based animation of data structures using jawaa. SIGCSE Bull. **30**(1), 267–271 (Mar 1998). https://doi.org/10.1145/274790.274310, http://doi.acm.org/10.1145/274790.274310
28. Pillay, N.: Learning difficulties experienced by students in a course on formal languages and automata theory. SIGCSE Bulletin **41**, 48–52 (01 2009). https://doi.org/10.1145/1709424.1709444
29. Radanne, G., Vouillon, J., Balat, V.: Eliom: A core ml language for tierless web programming. pp. 377–397 (11 2016). https://doi.org/10.1007/978-3-319-47958-3_20
30. Raymond, D., Wood, D.: Grail: A c++ library for automata and expressions. J. Symb. Comput. **17**(4), 341–350 (Apr 1994). https://doi.org/10.1006/jsco.1994.1023, http://dx.doi.org/10.1006/jsco.1994.1023
31. Rodger, S.: An interactive lecture approach to teaching computer science **27** (10 1998). https://doi.org/10.1145/199691.199820
32. Rodger, S., Wiebe, E., Min Lee, K., Morgan, C., Omar, K., Su, J.: Increasing engagement in automata theory with jflap. vol. 41, pp. 403–407 (03 2009). https://doi.org/10.1145/1508865.1509011
33. Sanders, I., Pilkington, C., Van Staden, W.: Errors made by students when designing finite automata (06 2015)

34. Stasko, J.T.: Tango: A framework and system for algorithm animation. SIGCHI Bull. **21**(3), 59–60 (Jan 1990). https://doi.org/10.1145/379088.1046618, http://doi.acm.org/10.1145/379088.1046618
35. Stoughton, A.: Experimenting with formal languages using forlan (01 2008). https://doi.org/10.1145/1411260.1411267
36. T. Grinder, M.: A preliminary empirical evaluation of the effectiveness of a finite state automaton animator. vol. 35, pp. 157–161 (01 2003). https://doi.org/10.1145/611892.611958
37. Vieira, L., Vieira, M., Vieira, N.: Language emulator, a helpful toolkit in the learning process of computer theory. vol. 36, pp. 135–139 (03 2004). https://doi.org/10.1145/971300.971348
38. Vouillon, J., Balat, V.: From Bytecode to JavaScript: the Js_of_ocaml Compiler. Software: Practice and Experience (2013). https://doi.org/10.1002/spe.2187
39. W. Brown, C., A. Hardisty, E.: Regexex: an interactive system providing regular expression exercises. vol. 39, pp. 445–449 (01 2007)
40. Wermelinger, M., Dias, A.: A prolog toolkit for formal languages and automata. ACM SIGCSE Bulletin **37** (09 2005). https://doi.org/10.1145/1067445.1067536