UNIVERSIDADE FEDERAL DE SANTA CATARINA
DEPARTAMENTO DE INFORMÁTICA E ESTATÍSTICA

Ghabriel Calsa Nunes

SIMULADOR DE AUTÔMATOS E MÁQUINAS DE TURING

Florianópolis

2017

Ghabriel Calsa Nunes

SIMULADOR DE AUTÔMATOS E MÁQUINAS DE TURING

Trabalho de Conclusão de Curso submetido ao Curso de Bacharelado em Ciências da Computação para a obtenção do Grau de Bacharel em Ciências da Computação.

Orientador: Prof. Dr. Olinto José Varela Furtado
Universidade Federal de Santa Catarina
Coorientador: Profa. Dra. Jerusa Marchi
Universidade Federal de Santa Catarina

Florianópolis

2017

Ficha de identificação da obra elaborada pelo autor,
através do Programa de Geração Automática da Biblioteca Universitária da UFSC.

Nunes, Ghabriel Calsa
Simulador de autômatos e máquinas de Turing / Ghabriel Calsa Nunes ; orientador, Olinto José Varela Furtado, coorientadora, Jerusa Marchi, 2017.
150 p.

Trabalho de Conclusão de Curso (graduação) - Universidade Federal de Santa Catarina, Centro Tecnológico, Graduação em Ciências da Computação, Florianópolis, 2017.

Inclui referências.

1. Ciências da Computação. 2. Autômato. 3. Mecanismo reconhecedor. 4. Aplicação web. I. Varela Furtado, Olinto José. II. Marchi, Jerusa. III. Universidade Federal de Santa Catarina. Graduação em Ciências da Computação. IV. Título.

RESUMO

O reconhecimento de sentenças por mecanismos reconhecedores tais como autômatos finitos, autômatos de pilha e máquinas de Turing é um dos assuntos mais importantes das disciplinas de Teoria da Computação e Linguagens Formais. Apesar disso, há uma escassez de sistemas de qualidade para simular tal reconhecimento, o que muitas vezes leva os alunos a permanecerem com dúvidas a respeito desses conteúdos, dificultando a aprendizagem. Propõe-se, então, o desenvolvimento de um sistema web capaz de suprir essa necessidade, visando melhorar a compreensão dos alunos acerca desses conteúdos. Será possível realizar o *download* de tal ferramenta para executá-la sem necessidade de acesso à Internet. Além disso, a interface será de fácil utilização e compatível com dispositivos móveis. O sistema será então comparado com outras soluções existentes utilizando critérios e características como facilidade de uso, completude, corretude e número de funcionalidades oferecidas.

Palavras-chave: reconhecimento de sentenças, autômatos finitos, autômatos de pilha, máquinas de Turing, autômatos linearmente limitados

ABSTRACT

The recognition of sentences by recognizing mechanisms such as finite automata, pushdown automata and Turing machines is one of the most important subjects in disciplines such as Theory of Computation and Formal Languages. However, there's a lack of high quality systems to simulate such recognition, which frequently makes students have questions about these subjects, resulting in a more difficult learning process. We propose a new web application capable of fulfilling these needs, aiming to improve the student comprehension about these subjects. It will be possible to download the new tool to run it without needing an Internet connection, and the interface will be easy to use and compatible with mobile devices. The system will then be compared with other existing solutions using criteria and characteristics such as ease of use, completeness, correctness and number of provided functionalities.

Keywords: sentence recognition, finite automata, pushdown automata, Turing machines, linear bounded automata

LISTA DE FIGURAS

LISTA DE ALGORITMOS

LISTA DE TABELAS

LISTA DE SIGLAS E ABREVIATURAS

SUMÁRIO

1 INTRODUÇÃO

A falta de bons simuladores de mecanismos reconhecedores em geral muitas vezes faz com que estudantes de Teoria da Computação e Linguagens Formais tenham dificuldades no aprendizado das máquinas estudadas nessas disciplinas e consequentemente nos aspectos conceituais e práticos. As soluções disponíveis atualmente se encaixam em pelo menos uma das seguintes categorias:

- apresentam difícil utilização, deixando o usuário confuso sobre como projetar e testar os autômatos que deseja. Em alguns casos, além da interface ser altamente complexa, não é fornecido um manual de uso;

- são incompletas, permitindo ao usuário projetar apenas um subconjunto dos autômatos que deveriam ser possíveis com os mecanismos disponibilizados, ou então não permite simular o reconhecimento de qualquer sentença. Muitas vezes isso é causado por limitações excessivas no alfabeto de entrada;

- apresentam problemas de funcionamento, em alguns casos levando a reconhecimentos incorretos ou a travamentos no programa, podendo fazer com que autômatos projetados pelo usuário tenham que ser reconstruídos manualmente.

- são difíceis de estender, isto é, é difícil - ou até impossível sem reescrever boa parte da aplicação - adicionar funcionalidades ao sistema.

1.1 OBJETIVOS

Tendo em mente os problemas encontrados nos simuladores existentes, juntamente às dificuldades dos alunos previamente citadas, propõe-se o desenvolvimento de um sistema web focado em mitigar esses problemas, provendo aos estudantes uma ferramenta capaz de auxiliá-los no aprendizado de linguagens recursivas, isto é, linguagens com parada garantida, estabelecendo os seguintes objetivos específicos:

1.1.1 Objetivos Específicos

Possibilitar a construção e manipulação de autômatos finitos, autômatos de pilha e autômatos linearmente limitados. O sistema deve focar principalmente na solução dos problemas e das dificuldades dos sistemas existentes, atendendo às seguintes propriedades:

- **Facilidade de uso** - o uso do sistema deverá ser intuitivo, dispensando a necessidade do uso de manuais para projetar qualquer autômato, embora neste trabalho seja incluído um manual para sanar eventuais dúvidas;
- **Completude** - o sistema deverá evitar restrições excessivas no alfabeto de entrada e no conteúdo das transições, sem deixar de respeitar as limitações inerentes dos mecanismos desenvolvidos;
- **Corretude** - o sistema deverá realizar os reconhecimentos corretamente e evitar travamentos (levando em conta a complexidade do autômato que se esteja manipulando, é claro).
- **Extensibilidade** - o sistema deverá ser capaz de ser facilmente estendido, ou seja, deve ser possível adicionar novos mecanismos reconhecedores ao sistema e também novas funcionalidades aos já existentes.

A ferramenta proposta deve permitir ao usuário projetar autômatos diretamente de forma gráfica para, em seguida, testar o reconhecimento de sentenças, seja passo a passo ou rapidamente. Também deve ser possível realizar o download do sistema para que se possa executá-lo localmente sem necessidade de acesso à Internet, o que implica que nenhuma linguagem *server-side* pode ser usada.

1.2 METODOLOGIA DE PESQUISA

Será apresentada uma análise de sistemas similares existentes atualmente. Eles serão comparados sob diversos critérios como usabilidade, completude, corretude e funcionalidades providas.

O projeto não possui vínculo com nenhum laboratório, sendo feito de forma independente. Os únicos softwares necessários para sua produção são um editor de texto (para produzir o código) e navegadores (para testar o sistema). O desenvolvimento será incremental, onde cada mecanismo a ser implementado é devidamente testado e concluído antes do início da produção do próximo.

1.3 ESTRUTURA DO TRABALHO

Inicialmente, será apresentada uma fundamentação teórica a respeito dos mecanismos reconhecedores que serão utilizados ao longo desta monografia. A seguir, no capítulo 3, aplicações com objetivos similares aos propostos serão analisadas e comparadas. No capítulo 4, o sistema proposto será introduzido e descrito em detalhes. Depois, no capítulo 5, serão descritas as diferentes maneiras pelas quais o sistema desenvolvido pode ser estendido. O capítulo 6 contém instruções de como utilizar as diferentes funcionalidades da aplicação. Por fim, no capítulo 7, serão listadas possíveis extensões futuras e as conclusões gerais.

2 FUNDAMENTAÇÃO TEÓRICA

Existem diversas áreas da Ciência da Computação nas quais não há um consenso sobre como definir certos conceitos. A área de Teoria da Computação, nesse sentido, não é exceção: mesmo conceitos tão importantes à área quanto autômatos finitos e outros mecanismos reconhecedores apresentam divergências entre os autores quanto às suas definições.

Tendo em mente tal divergência, torna-se necessário explicitar quais conceitos estão sendo utilizados em qualquer contexto em que eles sejam relevantes; o sistema proposto neste trabalho certamente se enquadra nesse caso.

Assim, este capítulo se propõe a detalhar algumas definições importantes que serão utilizadas posteriormente. Todas elas, assim como propriedades e teoremas relacionados, foram obtidas de (HOPCROFT; MOTWANI; ULLMAN, 2000), exceto onde explicitamente citado. Assume-se que o leitor esteja familizariado com as definições de computação, configuração e as definições básicas de alfabeto, palavra e linguagem.

2.1 AUTÔMATOS FINITOS

Autômatos finitos são mecanismos simples de reconhecimento que possuem complexidade de execução linear ao tamanho da palavra computada. Podem ser definidos como determinísticos ou não-determinísticos.

Uma propriedade importante dos *Finite Automata* (FA) trata da *classe da linguagem* reconhecida por eles. Independentemente do FA que se construa, a linguagem reconhecida por ele é dita ser *regular*. Existem dispositivos equivalentes capazes de gerar ou reconhecer linguagens regulares, como *gramáticas regulares*, *expressões regulares* ou, ainda, *conjuntos regulares*.

2.1.1 Autômatos Finitos Determinísticos

Um *Autômato Finito Determinístico*, ou *Deterministic Finite Automaton* (DFA), consiste de uma quíntupla da forma:

$$M = (Q, \Sigma, \delta, q_0, F)$$

onde:

- Q: um conjunto finito de estados;
- Σ: um conjunto finito de símbolos de entrada, comumente chamado de *alfabeto*;
- δ: uma função de transição $\delta : Q \times \Sigma \rightarrow Q$;
- q_0: um estado inicial, $q_0 \in Q$;
- F: um conjunto de estados *finais* (ou *aceitadores*), $F \subseteq Q$.

Diz-se que um DFA M *aceita* uma entrada $w = a_0.a_1....a_n$, onde $a_i \in \Sigma \ \forall i \in \{0, 1, ..., n\}$, se:

$$\hat{\delta}(q_0, w) \in F,$$

onde $\hat{\delta}(q_0, w)$ denota a aplicação sucessiva de δ sobre os a_i que compõem w, partindo-se do estado q_0. Por exemplo, para $w = a_0.a_1.a_2$, M aceita w se

$$\delta(\delta(\delta(q_0, a_0), a_1), a_2) \in F.$$

A função $\hat{\delta}$ chama-se *função de transição estendida.*

Quando M não aceita uma entrada w, diz-se que M *rejeita* w. A linguagem de um autômato qualquer M é definida como sendo o conjunto de todas as palavras w que são aceitas por M. No caso de DFAs, isto é equivalente a:

$$L(M) = \{w \mid \hat{\delta}(q_0, w) \in F\}$$

2.1.1.1 Minimização de DFAs

Um DFA M_1 é dito ser *mínimo* se não existir um DFA M_2 que reconheça a mesma linguagem de M e possua menos estados. Matematicamente, denotando $|Q_i|$ como o número de estados de um DFA M_i:

$$\nexists M_2 \mid L(M_2) = L(M_1) \wedge |Q_2| < |Q_1|$$

Existem algoritmos que transformam um DFA M qualquer em sua forma mínima, como o proposto em (HOPCROFT; MOTWANI; ULLMAN, 2000).

2.1.2 Autômatos Finitos Não-Determinísticos

Um *autômato finito não-determinístico*, ou *Non-deterministic Finite Automaton* (NFA), consiste de uma quíntupla semelhante à que representa DFAs. Quatro dos cinco elementos da quíntupla possuem o mesmo significado, diferindo apenas no contra-domínio da função de transição:

$$M = (Q, \Sigma, \delta, q_0, F)$$

onde:

- Q: um conjunto finito de estados;
- Σ: um conjunto finito de símbolos de entrada;
- δ: uma função de transição $\delta : Q \times (\Sigma \cup \{\varepsilon\}) \to Q^+$, onde Q^+ denota o fecho positivo de Q;
- q_0: um estado inicial, $q_0 \in Q$;
- F: um conjunto de estados *finais* (ou *aceitadores*), $F \subseteq Q$.

No caso de NFAs, um autômato M pode estar em mais de um estado em um dado momento. Dado um NFA $M = (Q, \Sigma, \delta, q_0, F)$ num conjunto de estados $K \subseteq Q^+$, ao receber como entrada um símbolo $a \in \Sigma$, seu novo conjunto de estados K' será dado por:

$$K' = \bigcup_{q \in K} \delta(q, a)$$

A linguagem definida por um NFA M$(Q, \Sigma, \delta, q_0, F)$ é dada por:

$$L(M) = \{w \mid \hat{\delta}(q_0, w) \cap F \neq \emptyset\}$$

Outra questão importante relacionada a autômatos finitos refere-se às suas propriedades, bem como as das linguagens reconhecidas por eles, isto é, as linguagens regulares. Essa importância se deve à possibilidade de delimitar quais operações podem ser realizadas sobre esses autômatos de modo que o resultado ainda possa ser operado da mesma forma, isto é, quais operações sobre autômatos finitos têm como resultado outros autômatos finitos. O mesmo raciocínio se aplica para as linguagens regulares.

2.1.3 Propriedades de autômatos finitos e linguagens regulares

Sejam L_1 e L_2 linguagens regulares quaisquer. Listam-se, a seguir, algumas propriedades comuns a todas as linguagens regulares e a todos os autômatos finitos, sejam eles determinísticos ou não-determinísticos.

- Equivalência entre DFA e NFA. Para todo NFA M_N, existe um DFA M_D equivalente, isto é, $L(M_N) = L(M_D)$. Existem algoritmos que realizam a transformação NFA $\Rightarrow$ DFA, como pode ser visto tanto em (SIPSER, 2006) como em (HOPCROFT; MOTWANI; ULLMAN, 2000). Tal processo é denominado *determinização*;

- A união de duas linguagens regulares é regular. Em outras palavras, seja $L = L_1 \cup L_2 = \{w \mid w \in L_1 \vee w \in L_2\}$. L é regular;

- A interseção de duas linguagens regulares é regular. Seja $L = L_1 \cap L_2 = \{w \mid w \in L_1 \wedge w \in L_2\}$. L é regular;

- A diferença entre duas linguagens regulares é regular. A linguagem $L = L_1 - L_2 = \{w \mid w \in L_1 \wedge w \notin L_2\}$ é regular;

- A concatenação de duas linguagens regulares é regular. A linguagem $L = L_1.L_2 = \{w_1.w_2 \mid w_1 \in L_1 \wedge w_2 \in L_2\}$ é regular;

- O complemento de uma linguagem regular é regular. A linguagem $L = \overline{L_1} = \{w \mid w \in \Sigma^* \wedge w \notin L_1\}$ é regular;

- O reverso de uma linguagem regular é regular. A linguagem $L = L_1^R = \{w \mid w^R \in L_1\}$ é regular;

- O fechamento de uma linguagem regular é regular. A linguagem $L = L_1^* = \{w_0.w_1....w_n \mid w_i \in L_1 \ \forall i \in \{0, 1, ..., n\}, n \geq 0\}$ é regular;

Para encerrar esta seção sobre autômatos finitos, é necessário, ainda, enunciar problemas de decisão relacionados a eles, pois, assim como foi dito para as propriedades de FAs, tais problemas permitem delimitar quais operações são factíveis sem deixar o escopo de autômatos finitos. A diferença aqui reside no fato do objeto de interesse não ser o tipo de saída da operação, mas sim se ela é decidível ou não.

2.1.4 Problemas de decisão

A seguir, listam-se alguns problemas de decisão envolvendo autômatos finitos. Todos os seguintes problemas são decidíveis:

- Dados dois autômatos finitos M_1 e M_2, $L(M_1) = L(M_2)$?
- Dados dois autômatos finitos M_1 e M_2, $L(M_1) \subseteq L(M_2)$?
- Dado um autômato finito M e uma palavra w, w $\in$ L(M)?
- Dado um autômato finito M, L(M) = $\emptyset$?
- Dado um autômato finito M, L(M) é finita?

2.2 AUTÔMATO DE PILHA

Autômatos de pilha são mecanismos reconhecedores semelhantes aos autômatos finitos. A diferença entre eles reside na presença de uma *pilha* auxiliar de símbolos, sob a qual três operações estão definidas: *empilhar* um ou mais símbolos, *desempilhar* um símbolo e *ler* o símbolo presente no topo da pilha. Assim como no caso dos autômatos finitos, os autômatos de pilha também possuem complexidade de execução linear ao tamanho da entrada, podendo ser definidos como determinísticos ou não-determinísticos.

As linguagens que podem ser reconhecidas por algum *Push-down Automaton* (PDA) são chamadas de Linguagens Livres de Contexto (LLC). Assim como no caso de autômatos finitos, existem dispositivos equivalentes capazes de gerar ou reconhecer o mesmo tipo de linguagem, como é o caso de *gramáticas livres de contexto* (SIPSER, 2006; HOPCROFT; MOTWANI; ULLMAN, 2000). Também é importante ressaltar que toda linguagem regular também é uma linguagem livre de contexto, pois autômatos finitos podem ser vistos como autômatos de pilha em que nenhuma transição altera o estado da pilha.

Um *autômato de pilha*, ou PDA, consiste de uma sétupla da forma:

$$M = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$$

onde:

- Q: um conjunto finito de estados;

- Σ: um conjunto finito de símbolos de entrada, comumente chamado de *alfabeto de entrada*;
- Γ: um conjunto finito de símbolos de pilha, comumente chamado de *alfabeto de pilha*;
- δ: uma função de transição $\delta : Q \times (\Sigma \cup \{\varepsilon\}) \times \Gamma \to (Q \times \Gamma^*)^+$;
- q_0: um estado inicial, $q_0 \in Q$;
- Z_0: um símbolo inicial de pilha, $Z_0 \in \Gamma$;
- F: um conjunto de estados *finais* (ou *aceitadores*), $F \subseteq Q$.

Representa-se a *configuração instantânea* de um PDA através de uma tripla (q, w, γ), onde:

- q: estado atual do autômato;
- w: a parte restante da entrada;
- γ: o conteúdo da pilha.

Dado um PDA M numa configuração $(q, aw, Z\gamma)$, onde $a \in \Sigma$ e $Z \in \Gamma$, caso M contenha a transição $\delta(q, a, Z) = (q', \alpha)$, denota-se:

$$(q, aw, Z\gamma) \vdash (q', w, \alpha\gamma)$$

Similarmente, usa-se o símbolo $\vdash^*$ para denotar o fecho de $\vdash$.

Quanto à linguagem aceita por um PDA, costuma-se dividir em dois tipos: os que aceitam por pilha vazia e os que aceitam por estado final, sendo ambos equivalentes em expressividade. No caso de PDAs que aceitam por pilha vazia, tem-se:

$$L(M) = \{w \mid (q_0, w, Z_0) \vdash^* (q, \varepsilon, \varepsilon), q \in Q\}$$

Já no caso de aceitação por estado final, vale:

$$L(M) = \{w \mid (q_0, w, Z_0) \vdash^* (q, \varepsilon, Z), q \in F, Z \in \Gamma^*\}$$

2.2.1 Determinismo × Não determinismo

Assim como os autômatos finitos, existem dois tipos de autômatos de pilha: os *determinísticos* e os *não-determinísticos*. Um PDA $M = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ é dito ser *determinístico*, podendo então ser chamado de *Deterministic Pushdown Automaton* (DPDA), se e somente se as seguintes condições são atendidas:

- $\forall q \in Q, a \in (\Sigma \cup \{\varepsilon\}), Z \in \Gamma^* : \exists_{\leq 1}\ \delta(q, a, Z)$;
- $\forall q \in Q, a \in \Sigma, Z \in \Gamma^* : \delta(q, a, Z) \neq \emptyset \Rightarrow \delta(q, \varepsilon, Z) = \emptyset$.

Ou seja, para todo estado q, símbolo do alfabeto a (incluindo ε) e conteúdo da pilha Z, há no máximo uma transição $\delta(q, a, Z) \in \delta$. Além disso, se tal transição existir com $a \neq \varepsilon$, então não pode haver $\delta(q, \varepsilon, Z) \in \delta$.

O não-cumprimento de qualquer uma destas condições implica em M ser *não-determinístico*, podendo então ser chamado de *Nondeterministic Pushdown Automaton* (NPDA).

Ao contrário dos autômatos finitos, nem todo NPDA admite um DPDA equivalente. Equivalentemente, existem linguagens livres de contexto que não podem ser reconhecidas por um DPDA.

2.2.2 Propriedades de PDAs e LLCs

Sejam L_1 e L_2 linguagens livres de contexto quaisquer. Listam-se, a seguir, algumas propriedades comuns a todas as linguagens livres de contexto.

- A união de duas LLCs é livre de contexto. Em outras palavras, seja $L = L_1 \cup L_2 = \{w \mid w \in L_1 \vee w \in L_2\}$. L é livre de contexto;
- A concatenação de duas LLCs é livre de contexto. A linguagem $L = L_1.L_2 = \{w_1.w_2 \mid w_1 \in L_1 \wedge w_2 \in L_2\}$ é livre de contexto;
- O reverso de uma LLC é livre de contexto. A linguagem $L = L_1^R = \{w \mid w^R \in L_1\}$ é livre de contexto;
- O fechamento de uma LLC, tanto positivo quanto de Kleene, é livre de contexto. As linguagens $L = L_1^* = \{w_0.w_1....w_n \mid w_i \in L_1\ \forall i \in \{0, 1, ..., n\}, n \geq 0\}$ e $L' = L_1^+ = \{w_0.w_1....w_n \mid w_i \in L_1\ \forall i \in \{0, 1, ..., n\}, n > 0\}$ são livres de contexto;
- A interseção de duas LLCs não necessariamente é livre de contexto. Como exemplo, a interseção das linguagens

$$L_1 = \{a^m b^m c^n d^n \mid m, n > 0\}$$
$$L_2 = \{a^m b^n c^n d^m \mid m, n > 0\}$$

que resulta em:

$$L = \{a^n b^n c^n d^n \mid n > 0\}$$

não é livre de contexto;

- A diferença entre duas LLCs não necessariamente é livre de contexto. Como exemplo, a diferença $L_1 - L_2$ entre as linguagens

$$L_1 = \{a^m b^m c^n d^n \mid m, n > 0\}$$
$$L_2 = \{a^p b^m c^n d^q \mid p, q > 0, m \neq n\}$$

que resulta em:

$$L = \{a^n b^n c^n d^n \mid n > 0\}$$

não é livre de contexto.

Assim como foi dito para os autômatos finitos, é importante destacar quais os problemas de decisão relacionados a autômatos de pilha e linguagens livres de contexto. Tal importância é ainda maior neste caso, pois existem problemas indecidíveis relacionados a esses dois conceitos, como mostrado na seção a seguir.

2.2.3 Problemas de decisão

A seguir, listam-se alguns problemas de decisão envolvendo autômatos de pilha e linguagens livres de contexto em geral. Todos os seguintes problemas são decidíveis:

- Dado um autômato de pilha M, L(M) = $\emptyset$?
- Dado um autômato de pilha M e uma palavra x, x $\in$ L(M)?
- Dado um autômato de pilha M, L(M) é finita?

Porém, ao contrário das linguagens regulares, nem todos os problemas acerca de linguagens livres de contexto são decidíveis. Os problemas a seguir, por exemplo, são indecidíveis:

- Dados dois autômatos de pilha M_1 e M_2, $L(M_1) = L(M_2)$?
- Dados dois autômatos de pilha M_1 e M_2, $L(M_1) \subseteq L(M_2)$?
- Dado uma LLC L, L é inerentemente ambígua?

2.3 MÁQUINA DE TURING

A máquina de Turing é o mecanismo reconhecedor mais expressivo conhecido, sendo capaz de reconhecer todas as linguagens que os autômatos anteriores reconhecem, além de muitas outras, como as chamadas *linguagens sensíveis ao contexto* (ou, de forma ainda mais abrangente, as *linguagens recursivas*) e até *linguagens recursivamente enumeráveis*, isto é, linguagens para as quais não é possível produzir um mecanismo reconhecedor com parada garantida.

Ao contrário dos autômatos finitos e autômatos de pilha, máquinas de Turing não necessariamente apresentam complexidade de execução linear ao tamanho da entrada. Não há restrições de tempo que se apliquem a todas elas, visto que elas sequer necessitam apresentar parada garantida (isto é, não necessariamente são decidíveis).

Uma *máquina de Turing*, ou *Turing Machine* (TM), consiste de uma sétupla da forma:

$$M = (Q, \Sigma, \Gamma, \delta, q_0, B, F)$$

onde:

- Q: um conjunto finito de estados;
- Σ: um conjunto finito de símbolos de entrada, comumente chamado de *alfabeto de entrada*;
- Γ: um conjunto finito de símbolos de fita, comumente chamado de *alfabeto de fita*, $\Sigma \subseteq \Gamma$;
- δ: uma função de transição $\delta : Q \times \Gamma \rightarrow Q \times \Gamma \times \{\leftarrow, \rightarrow\}$;
- q_0: um estado inicial, $q_0 \in Q$;
- B: um símbolo *branco*, $B \in \Gamma - \Sigma$;
- F: um conjunto de estados *finais* (ou *aceitadores*), $F \subseteq Q$.

Uma máquina de Turing representa um dispositivo no qual a entrada é posta numa *fita*, dividida em células, cada uma podendo conter um número finito de símbolos. A fita é infinita em ambas as direções (esquerda e direita), sendo que as células não ocupadas pela entrada contêm um símbolo especial *branco*, doravante denotado como B. Além disso, a máquina possui um *cabeçote*, representando a posição atual da fita, e um estado. Tanto o conteúdo da fita quanto o estado

atual da máquina podem ser alterados dependendo dos símbolos que estão sob o cabeçote e de qual estado a máquina se encontra.

Representa-se a *configuração instantânea* de uma TM através de uma tupla $(q, X_1X_2...\underline{X_i}...X_n)$, onde:

- q representa o estado atual da máquina de Turing;
- o cabeçote da fita está posicionado sobre o i-ésimo símbolo da esquerda para direita;
- $X_1X_2...X_n$ é a porção da fita entre o *branco* mais à esquerda e o mais à direita, exceto se o cabeçote estiver fora deste intervalo. Neste caso, $\exists i \in \{1, 2, ..., n\} \mid X_i = B$.

Existem diversas variações de máquinas de Turing, nas quais certas restrições da definição geral são relaxadas ou mais restrições são adicionadas visando conferir certas propriedades aos mecanismos. A seguir, são listadas as principais variações, que serão utilizadas posteriormente para fins de comparação dos trabalhos correlatos, no capítulo 3.

2.3.1 Máquina de Turing Multifita

Uma máquina de Turing multifita possui múltiplos cabeçotes, cada um deles sobre uma fita distinta. As transições de tal máquina podem controlar os cabeçotes e alterar o conteúdo de quaisquer fitas, além de, como numa TM normal, alterar o estado atual da máquina.

Apesar de aparentar que esta variante de máquina de Turing possui maior expressividade que uma TM normal, ambas são equivalentes. Existem algoritmos que permitem transformar uma TM multifita numa de fita única equivalente, como exposto em (SIPSER, 2006; HOPCROFT; MOTWANI; ULLMAN, 2000).

2.3.2 Máquina de Turing Não-determinística

Como visto anteriormente, numa máquina de Turing convencional, a função de transição δ possui a forma:

$$\delta : Q \times \Gamma \to Q \times \Gamma \times \{\leftarrow, \rightarrow\}$$

Uma máquina de Turing M é dita ser *não-determinística* se, ao invés disso, δ possuir a seguinte forma:

$$\delta : Q \times \Gamma \to (Q \times \Gamma \times \{\leftarrow, \rightarrow\})^+$$

Isto é, a função de transição de M mapeia para um conjunto de triplas, onde cada tripla possui o mesmo significado que numa transição determinística. A cada passo de computação, M pode escolher qualquer uma das triplas do conjunto para indicar sua próxima ação.

Assim como no caso de máquinas de Turing multifita, embora esta variante aparente ser mais expressiva que uma TM determinística, elas na verdade são equivalentes, havendo inclusive algoritmos de determinização. A ideia por trás desse processo pode ser vista em (HOPCROFT; MOTWANI; ULLMAN, 2000).

2.3.3 Autômatos Linearmente Limitados

Um *autômato linearmente limitado*, ou *Linear Bounded Automaton* (LBA), é um tipo restrito de máquina de Turing no qual o cabeçote não pode se mover para fora da porção de fita que contém a entrada (SIPSER, 2006). É importante notar, em particular, que tal restrição torna a fita finita e com tamanho igual ao da entrada, restringindo os problemas que podem ser resolvidos, embora o mecanismo resultante ainda seja superior a um PDA.

Uma propriedade importante de LBAs trata do fato de eles serem sempre decidíveis. Como pode ser visto em (SIPSER, 2006), dado um LBA M de q estados, $g = |\Gamma|$ e uma entrada de tamanho n, existem exatamente qng^n configurações distintas possíveis para M. Assim, se M não para dentro de qng^n passos de computação, em algum momento a configuração de M se repetiu e, portanto, M está em *loop*. Nesse caso, rejeita-se a entrada, proporcionando a M parada garantida.

Neste capítulo, foram apresentadas as definições de todos os mecanismos reconhecedores que serão mencionados ao longo deste trabalho. O próximo capítulo se propõe a comparar aplicações que permitem a realização de reconhecimento de sentenças utilizando alguns dos mecanismos mencionados, visando posteriormente, no capítulo 4, apresentar a proposta de um novo sistema com objetivo semelhante que visa reunir diferentes aspectos de tais aplicações, além de prover funcionalidades que poucas ou até mesmo nenhuma delas oferecem.

3 TRABALHOS CORRELATOS

Em qualquer projeto de software, é natural buscar por aplicações já existentes que se proponham a resolver problemas similares. Tal busca permite a obtenção de ideias a respeito de possíveis funcionalidades, comportamento e organização da interface, entre diversos outros aspectos, além de permitir a realização de uma análise das limitações que tais sistemas possuem.

Inicia-se este capítulo apresentando-se alguns sistemas similares ao proposto, assim como uma análise de suas características e limitações sob os seguintes critérios de avaliação: usabilidade, isto é, simplicidade de uso e suporte a funcionalidades como salvar e carregar autômatos; funcionalidades suportadas a respeito de autômatos finitos, autômatos de pilha e máquinas de Turing, variando desde quais tipos são suportados (em relação a determinismo e, no caso de máquinas de Turing, quantidade de fitas) até a maneira como são fornecidas as entradas e os modos de reconhecimento suportados, além das operações permitidas envolvendo cada tipo de autômato. No caso particular de modos de reconhecimento, a seguinte nomenclatura será empregada: reconhecimento *passo-a-passo* é aquele no qual o usuário pode observar em quais estados o autômato se encontra após a leitura de cada símbolo da entrada; reconhecimento *rápido* refere-se a não haver interrupção do usuário, isto é, o reconhecimento ocorre em um único passo; por fim, reconhecimento *múltiplo* compreende o teste de diversas palavras ao mesmo tempo.

Por fim, ao final do capítulo, são apresentadas tabelas que sintetizam a análise realizada, ilustrando mais claramente as diferenças entre os trabalhos correlatos sob diferentes aspectos, além de mostrar as funcionalidades suportadas pela nova ferramenta que será introduzida nos capítulos posteriores.

3.1 AUTOMATON SIMULATOR (K. DICKERSON)

Este sistema online[1], criado por Kyle Dickerson, apresenta uma das interfaces gráficas mais intuitivas entre os trabalhos analisados. Por permitir edição via grafo dos autômatos produzidos, fica claro ao usuário como interagir com o programa. Além disso, por ser uma ferramenta online, não requer instalação. Apresenta suporte total a salvar

[1] http://automatonsimulator.com/

e carregar autômatos.

Quanto a autômatos finitos, apresenta bom suporte tanto a DFA quanto NFA, permitindo edição gráfica, reconhecimento passo-a-passo e reconhecimento múltiplo, porém não suporta edição via tabela de transições nem via expressão regular.

No caso dos autômatos de pilha, o suporte é similar. São aceitos tanto DPDA quanto NPDA, permitindo praticamente as mesmas funcionalidades que apresenta para autômatos finitos, incluindo a limitação de não permitir edição via tabela de transições. Não permite reconhecimento por pilha vazia nem a realização de um *push* de múltiplos símbolos na pilha em uma única transição. Esta ferramenta não possui suporte a nenhuma variante de máquina de Turing.

3.2 FSM SIMULATOR (I. ZUZAK E V. JANKOVIC)

Esta ferramenta[2], produzida por Ivan Zuzak e Vedrana Jankovic, é focada na criação de autômatos finitos a partir de expressões regulares, além de permitir tal geração via código numa linguagem própria de alto nível de sintaxe simples. Não permite salvar nem carregar autômatos. Não requer instalação, por ser uma ferramenta online.

Através de expressões regulares, é gerado um NFA equivalente. É possível criar um DFA apenas através da linguagem própria do sistema. Permite somente reconhecimento passo-a-passo, além de limitar-se a edição via expressão regular. Somente autômatos finitos são suportados.

3.3 TURING MACHINE SIMULATOR (A. MORPHETT)

Ferramenta[3] criada por Anthony Morphett em 2014 focada em máquinas de Turing de fita única. A entrada do programa se dá somente através de uma linguagem própria cuja sintaxe é rebuscada, dificultando seu uso. Possui suporte a salvar e carregar máquinas através de *links* de acesso. Por ser uma ferramenta online, não requer instalação.

Esta ferramenta permite reconhecimento passo-a-passo e reconhecimento rápido das seguintes variantes de máquinas de Turing (somente fita única):

- Determinística, com fita infinita em ambas as direções;

[2] http://ivanzuzak.info/noam/webapps/fsm_simulator/
[3] http://morphett.info/turing/turing.html

- Determinística, com fita semi-infinita (isto é, infinita em apenas uma direção;

- Não-determinística, na qual, caso haja mais de uma regra compatível com a configuração atual, uma é escolhida aleatoriamente para ser usada.

3.4 ONLINE TM SIMULATOR (M. UGARTE)

Sistema[4] criado por Martin Ugarte em 2015. A entrada do programa se dá através de uma linguagem própria com sintaxe simples, o que torna seu uso simples embora não possibilite edição via diagrama. Possui suporte a salvar e carregar máquinas através de *links* de acesso ou localmente (esta última exige cadastro). Por ser uma ferramenta online, não requer instalação.

Esta ferramenta suporta somente máquinas de Turing. Apesar disso, ao contrário da anterior, esta suporta até 3 fitas, incluindo reconhecimento passo-a-passo e reconhecimento com controle de velocidade.

Por fim, esta ferramenta limita-se a edição via código de máquinas de Turing determinísticas.

3.5 TURING MACHINE SIMULATOR (P. RENDELL)

Assim como as duas ferramentas anteriores, esta[5] também possui enfoque em máquinas de Turing. Criada por Paul Rendell, esta aplicação permite entrada através de tabela de transições, que, embora simples, apresenta usabilidade inferior a edição via diagrama. Além disso, não possui suporte a salvar nem carregar máquinas. É uma ferramenta online, portanto dispensa instalação.

Esta ferramenta limita-se a máquinas de Turing determinísticas *single tape* com fita limitada em 200 posições para cada lado, totalizando 401 posições contando com a central. Permite reconhecimento passo-a-passo e reconhecimento rápido assim como as aplicações anteriores, e, assim como elas, permite edição apenas via código.

[4] https://turingmachinesimulator.com/
[5] http://www.rendell-attic.org/gol/TMapplet/

3.6 AUTOMATON SIMULATOR (C. BURCH)

Este sistema[6], desenvolvido por Carl Burch, permite construir máquinas facilmente, porém sem muita agilidade devido à falta de atalhos de teclado para alternar de função, além de possuir alfabeto limitado a {a, b, c, d}. Ao contrário dos trabalhos correlatos descritos até agora, esta requer *download*, visto que não é online. Apresenta suporte completo a salvar e carregar máquinas construídas.

Esta aplicação possui bom suporte tanto a DFA quanto NFA, permitindo edição via diagrama e reconhecimento passo-a-passo, apesar de não suportar reconhecimento múltiplo, edição via tabela de transições nem edição via expressão regular.

Quanto a autômatos de pilha, suporta somente DPDA. Também possui a limitação de não permitir edição via tabela de transições nem reconhecimento múltiplo. Não permite reconhecimento por pilha vazia nem a realização de um *push* de múltiplos símbolos na pilha em uma única transição.

Diferentemente das ferramentas vistas até aqui, esta é a primeira alternativa a suportar tanto autômatos finitos quanto alguma variante de máquina de Turing. É suportada somente a variante *single tape*, com suporte a funcionalidades semelhantes às que suporta para autômatos de pilha. Novamente, não suporta reconhecimento múltiplo nem outro tipo de entrada que não a via diagrama.

3.7 JFAST (T. M. WHITE)

Esta ferramenta[7], produzida por Timothy M. White em 2006, não é tão simples de usar quanto a anterior, principalmente devido à necessidade de definir o alfabeto de entrada separadamente. Não proporciona muita agilidade devido à falta de atalhos de teclado para alternar de função e por criar muitas janelas auxiliares durante a construção. Requer *download*, pois não é online. Suporte completo a salvar e carregar máquinas construídas.

Uma desvantagem bastante significativa desta ferramenta trata da presença de diversos *bugs* que podem tornar seu uso significativamente menos agradável, além de não possuir algumas funcionalidades básicas que agravam ainda mais essa situação, como, por exemplo, não permitir que uma transição em construção seja cancelada.

[6] http://www.cburch.com/proj/autosim/

[7] http://www46.homepage.villanova.edu/timothy.m.white/

Embora suporte DFA e NFA com edição via diagrama, não há a possibilidade de realizar reconhecimento múltiplo nem edição via tabela de transições ou via expressão regular. No reconhecimento de NFA, quando há caminhos bem sucedidos, somente um é mostrado. Quando não há, um caminho viável aparentemente aleatório é mostrado, de modo que o autômato nunca é mostrado em mais de um estado. Este comportamento é confuso e pouco intuitivo.

Já quanto a autômatos de pilha, são suportados tanto DPDA quanto NPDA, mas possui as mesmas limitações do reconhecimento de autômatos finitos. Não permite reconhecimento por pilha vazia nem a realização de um *push* de múltiplos símbolos na pilha em uma única transição. No reconhecimento de NPDA, possui o mesmo comportamento confuso de NFAs.

Por fim, esta aplicação suporta máquinas de Turing *single tape*, com suporte a funcionalidades semelhantes às que suporta para autômatos de pilha. Novamente, não suporta reconhecimento múltiplo nem outro tipo de entrada que não a via diagrama. Também não suporta estado aceitador.

3.8 JFLAP (S. H. RODGER)

Esta ferramenta[8], produzida originalmente por Susan H. Rodger em 2003, é a mais completa entre os trabalhos correlatos analisados. Embora sua primeira versão seja antiga, diversas melhorias e novas funcionalidades foram adicionadas desde sua concepção original.

O JFLAP apresenta fácil utilização, apesar de exigir frequentes trocas de função para construir máquinas. Este problema é mitigado pela presença de atalhos de teclado, porém possibilitar diferentes ações sem trocar de função traria melhor usabilidade.

Algumas funcionalidades são confusas ou apresentam *bugs*, como melhor descrito posteriormente. Não é online, portanto o *download* é necessário e, assim como as duas ferramentas anteriores, requer o Java instalado. Apresenta suporte completo a salvar e carregar máquinas construídas, embora o formato do arquivo salvo seja XML, que não é compacto.

Quanto a autômatos finitos, são suportados tão somente DFA e NFA. A edição ocorre via diagrama. Suporta entrada via expressão regular, embora gere um NFA com muitos estados desnecessários (a aplicação suporta conversão de NFA em DFA, mas é necessário salvar o

[8] http://www.jflap.org/

NFA gerado e então abri-lo para só então realizar a conversão). Suporta reconhecimento passo-a-passo, rápido e múltiplo.

O JFLAP também apresenta suporte a DPDA e NPDA. Possui funcionalidades semelhantes às do suporte a autômatos finitos, incluindo a impossibilidade de edição via tabela de transições. O reconhecimento passo-a-passo no caso de NPDA é confuso, embora seja compreensível dada a complexidade desta tarefa. Possui um modo no qual é permitido múltiplos *pushs* numa única transição.

Há suporte a máquinas de Turing tanto *single tape* quanto *multi tape* (até 5 fitas), com suporte a funcionalidades semelhantes às que suporta para autômatos de pilha, porém não suporta não-determinismo nem edição via outros meios que não via diagrama. O reconhecimento rápido não funciona corretamente para máquinas *multi tape* em alguns casos, nos quais o programa encontra caminhos bem sucedidos que na verdade não o são.

A aplicação não suporta especificamente autômatos linearmente limitados, o que implica uma inexistência de detecção de parada para uma entrada em particular. Por outro lado, a aplicação emite um aviso a cada 500 configurações geradas, perguntando ao usuário se ele deseja continuar. Embora interessante para máquinas simples, essa funcionalidade, que não pode ser desabilitada, torna desagradável testar máquinas mais complexas que exijam milhares de configurações para terminar seu processamento.

3.9 ANÁLISE COMPARATIVA

Nesta seção, como dito anteriormente, no início deste capítulo, serão apresentadas tabelas que exibem uma comparação dos trabalhos correlatos sob os diversos critérios analisados, juntamente com a nova ferramenta proposta, que será explicada em detalhes nos capítulos posteriores. As ferramentas de 1 a 8 referenciam as aplicações apresentadas anteriormente na mesma ordem, seguidas pela nova ferramenta, isto é:

1. Automaton Simulator (Kyle Dickerson);
2. FSM Simulator (Ivan Zuzak e Vedrana Jankovic);
3. Turing machine simulator (Anthony Morphett);
4. Online Turing Machine Simulator (Martin Ugarte);
5. Turing Machine Simulator (Paul Rendell);

6. Automaton Simulator (Carl Burch);
7. jFAST (Timothy M. White);
8. JFLAP (Susan H. Rodger);
9. Nova ferramenta proposta.

Com relação a funcionalidades, os seguintes critérios foram avaliados:

- Online (sim/não);
- I/O, subdividida nos seguintes níveis:
  - S: suporte completo (armazenamento local);
  - N: sem suporte;
  - L: *link* de acesso (armazenamento não-local);
- Alfabeto irrestrito (sim/não).

A tabela 1 apresenta um resumo das funcionalidades nas ferramentas estudadas:

Funcionalidade		Ferramentas								
		1	2	3	4	5	6	7	8	9
Online		S	S	S	S	S	N	N	N	S
I/O*	Salvar	S	N	L	**	N	S	S	S	S
	Carregar	S	N	L	**	N	S	S	S	S
Alfabeto irrestrito		S	S	S	S	S	N	S	S	S

Tabela 1: Funcionalidades genéricas dos trabalhos correlatos

onde ** utiliza *links* de acesso mas, com cadastro, também permite armazenamento local.

A tabela 2 sumariza as características suportadas para os mecanismos analisados:

Mecanismo	Característica		Ferramentas								
			1	2	3	4	5	6	7	8	9
FA	Entrada	Diagrama	S	N	N	N	N	S	S	S	S
		Tabela	N	N	N	N	S	N	N	N	S
		ER	N	S	N	N	N	N	N	S	N
	Rec.	Passo a passo	S	S	N	N	N	S	S	S	S
		Rápido	S	S	N	N	N	S	S	S	S
		Múltiplo	S	N	N	N	N	N	N	S	S
	ER → NFA		N	S	N	N	N	N	N	S	N
	ER → DFA		N	N	N	N	N	N	N	N*	N
	Determinização		N	N	N	N	N	N	N	S	N
	Minimização		N	N	N	N	N	N	N	S	N
PDA	Entrada	Diagrama	S	N	N	N	N	S**	S	S	S
		Tabela	N	N	N	N	N	N	N	N	S
	Rec.	Passo a passo	S	N	N	N	N	S**	S	S	S
		Rápido	S	N	N	N	N	S**	S	S	S
		Múltiplo	S	N	N	N	N	N	N	S	S
	Tipo de rec.	Estado final	S	N	N	N	N	S**	S	S	S
		Pilha vazia	N	N	N	N	N	N	N	S	S
	Múltiplos *pushs* numa transição		N	N	N	N	N	N	N	S	S
TM	Entrada	Diagrama	N	N	N	N	N	S	S	S	S
		Tabela	N	N	N	N	N	N	N	N	S
		Código	N	N	S	S	S	N	N	N	N
	Rec.	Passo a passo	N	N	S	S	S	S	S	S	S
		Rápido	N	N	S	S	S	S	S	S	S
		Múltiplo	N	N	N	N	N	N	N	S	S
	Limite de fitas		-	-	1	3	1	1	1	5	1
	Autômato Linearmente Limitado		N	N	N	N	N	N	N	N	S

Tabela 2: Mecanismos dos trabalhos correlatos

* o JFLAP não permite a conversão direta ER → DFA, mas, por pos-

suir suporte a determinização, é possível realizar o processo ER → NFA → DFA, embora este procedimento exija que o NFA intermediário gerado seja salvo na máquina do usuário e, então, carregado para que a determinização possa ser realizada.
** somente para DPDA.

Como visto, as ferramentas analisadas possuem grande variedade de funcionalidades, embora existam algumas interessantes que nenhuma delas suporta como, por exemplo, suporte a autômatos linearmente limitados.

Assim, no próximo capítulo será descrito um novo sistema que, além de prover boa parte das funcionalidades apresentadas, visa, em particular, suportar algumas menos comuns, como os já citados LBAs.

4 PROPOSTA

No capítulo anterior, foram listados diversos critérios e características para avaliar aplicações relacionadas ao reconhecimento de sentenças em diversos mecanismos reconhecedores. Neste capítulo, é descrito um novo sistema com objetivo semelhante, focado principalmente em facilidade de uso, variedade de funcionalidades e extensibilidade.

4.1 FUNCIONALIDADES GERAIS DO SISTEMA

A aplicação suporta três mecanismos reconhecedores[1]: autômatos finitos, tanto determinísticos quanto não-determinísticos; autômatos de pilha, tanto determinísticos quanto não-determinísticos e autômatos linearmente limitados, que, como visto no capítulo 2, compreendem um subconjunto das máquinas de Turing. Para cada um desses mecanismos, são suportados dois tipos de entrada, isto é, duas diferentes maneiras de se criar um autômato de algum dos tipos mencionados:

- Entrada via diagrama: neste tipo de entrada, o usuário cria diretamente um grafo que representa o autômato sendo desenvolvido. Costuma ser a principal forma de entrada, por ser mais intuitiva devido a seu aspecto gráfico;
- Entrada via tabela de transição: o usuário pode alterar diretamente a tabela de transições de um autômato, que atualiza automaticamente o grafo correspondente.

A qualquer momento, o usuário pode realizar um reconhecimento sobre o autômato criado. Três tipos de reconhecimento são suportados:

- Passo a passo: o usuário pode observar cada caractere da entrada sendo lido pelo autômato, um de cada vez;
- Rápido: uma entrada é fornecida e um *feedback* é retornado em um único passo, mostrando se a entrada foi aceita ou não. Ao chegar ao término do reconhecimento, o autômato é exibido em sua configuração final;
- Múltiplo: execução em paralelo de múltiplas entradas. É informado quais entradas foram aceitas e quais foram rejeitadas.

[1] Como será apresentado no capítulo 5, é possível adicionar mais mecanismos reconhecedores ao sistema.

No caso de autômatos de pilha, o usuário pode escolher se deseja que o reconhecimento seja feito via estado final, via pilha vazia ou ambos. É permitido realizar múltiplos *pushs* numa única transição.

A aplicação apresenta suporte completo ao salvamento e carregamento de autômatos. Os arquivos salvos ficam em formato JSON, que é naturalmente mais compacto e simples de manusear que XML. Adicionalmente, foi utilizada uma notação enxuta visando minimizar redundâncias para diminuir ainda mais o tamanho do arquivo.

Para facilitar o uso da aplicação, todas as funcionalidades do sistema possuem botões correspondentes na interface, além de boa parte delas também possuir atalhos de teclado associados. Assim, mesmo usuários que estejam utilizando a ferramenta pela primeira vez possuem fácil acesso a 100% das funcionalidades da aplicação.

O sistema proposto possui ainda suporte a multi-idiomas, sendo este um dos aspectos de fácil extensibilidade, como será apresentado no capítulo 5. Por padrão, os idiomas português e inglês estão disponíveis, podendo ser alternados livremente a critério do usuário.

4.2 METODOLOGIA DE IMPLEMENTAÇÃO

O software proposto foi desenvolvido com a linguagem de programação TypeScript, que, por sua vez, gera códigos em JavaScript. Por ser um sistema web, a interface - na qual conteúdos dinâmicos serão criados via TypeScript - será feita com HTML e CSS. Outro ponto importante trata do uso de um *makefile*, que, embora não seja usado pelo usuário final, é necessário para realizar modificações no código da aplicação. O propósito exato desse *makefile* será detalhado posteriormente, na seção 4.3.7.

A linguagem TypeScript foi escolhida por dois motivos principais. Primeiramente, ela é uma linguagem orientada a objetos, que é um paradigma familiar que permite o desenvolvimento de sistemas com alta manutenibilidade e extensibilidade. Além disso, tal linguagem é *client-side*, o que permite que o usuário do sistema realize *download* da ferramenta para utilizá-la localmente, sem necessidade de acesso à Internet.

4.3 ESTRUTURA

Com o propósito de facilitar o entendimento da estrutura do sistema, o conjunto de classes que o compõe será apresentado em partes focadas em diferentes aspectos da aplicação. Inicialmente, são mostradas as principais classes relacionadas à *Graphical User Interface* (GUI).

4.3.1 Interface gráfica

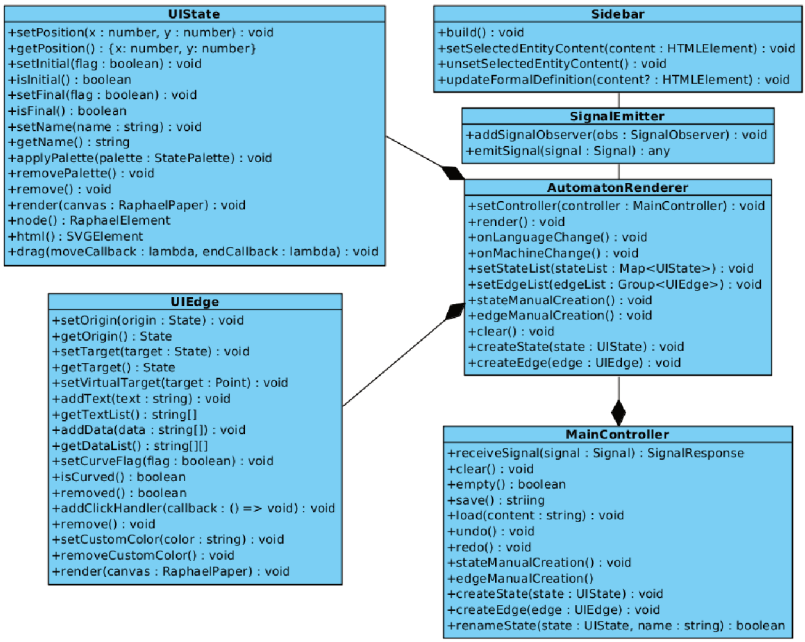

Figura 1: Classes relacionadas à GUI

A classe *AutomatonRenderer* presente no diagrama acima é responsável pelo controle do canvas no qual os autômatos são exibidos, além de interagir com a barra lateral do sistema (classe *Sidebar*). Já a classe *MainController* é a principal classe do sistema, controlando as chamadas ao *AutomatonRenderer* e aos controladores, mostrados na próxima sub-seção. A classe *SignalEmitter* atua como um *middleware* que provê maior flexibilidade à interação entre as classes; em particular,

ela torna possível remover a barra lateral do sistema sem que a região principal deixe de funcionar.

As classes *UIState* e *UIEdge* representam, respectivamente, um estado e uma aresta, onde cada aresta corresponde a uma ou mais transições no autômato. É importante ressaltar que essas duas classes são apenas representações visuais, isto é, elas são estruturas genéricas que não contêm as informações necessárias para um reconhecimento, mas contêm o suficiente para serem exibidos na tela corretamente, independentemente de qual tipo de autômato representam. As informações necessárias para um reconhecimento ficam armazenadas em classes especializadas de cada mecanismo, como é mostrado[2] a seguir.

4.3.2 Mecanismos reconhecedores

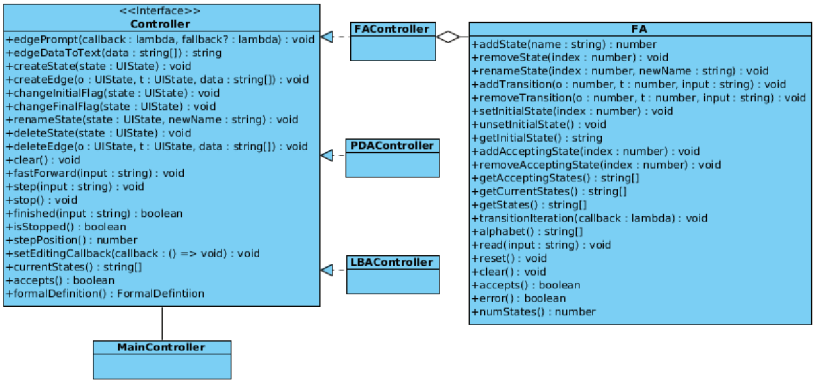

Figura 2: Classes relacionadas aos mecanismos reconhecedores

Um conceito chave presente neste diagrama é a ideia de um *controller* (controlador). Um controlador é um meio de comunicação entre a GUI e o mecanismo específico que o usuário está utilizando (do ponto de vista de engenharia de software, um controlador pode ser visto como um *adapter* (GAMMA et al., 1994) que converte a interface do mecanismo na interface esperada pela GUI). A existência desses controladores permite um maior desacoplamento entre cada mecanismo específico e a GUI, simplificando o desenvolvimento de ambas as partes.

[2]Nos diagramas posteriores, visando simplificação visual, não serão mostrados novamente os métodos da classe *MainController*.

A interface *Controller* exibida no diagrama anterior serve para padronizar os diferentes controladores da aplicação. Vale notar que tal modelagem proporciona grande flexibilidade, pois novos mecanismos podem ser adicionados à aplicação bastando que um controlador correspondente (que implemente a interface *Controller*) seja também implementado (na verdade, há outra estrutura que também é necessária, que será explicada mais adiante, no capítulo 5).

4.3.3 Suporte multi-idioma

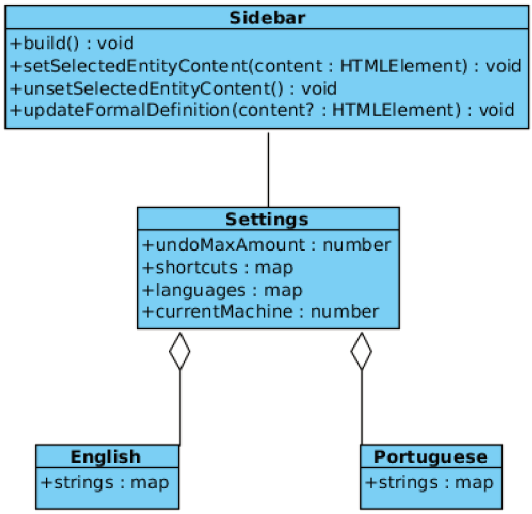

Figura 3: Classes relacionadas ao suporte multi-idioma

Na figura 3, exposta acima, é mostrado o conjunto de classes relacionado ao suporte multi-idioma do sistema. *Settings* é um *namespace*[3] que contém diversas configurações do sistema, como o tamanho de cada elemento da interface, os atalhos do sistema, o idioma atual, entre outros. No diagrama, foram listados apenas alguns poucos atributos para não torná-lo muito poluído visualmente.

O *Makefile* reúne todos os idiomas do sistema num arquivo chamado *LanguageList.ts*, de modo que qualquer parte do sistema possa

[3]Mais informações sobre *namespaces* podem ser encontradas em `https://www.typescriptlang.org/docs/handbook/namespaces.html`

obter facilmente os idiomas suportados. Na classe *Sidebar* encontra-se uma lista *drop-down*[4] que permite ao usuário alterar o idioma do sistema para qualquer outro suportado. Essa estrutura permite facilmente a adição de novos idiomas ao sistema, como será melhor detalhado no capítulo 5.

4.3.4 Persistência

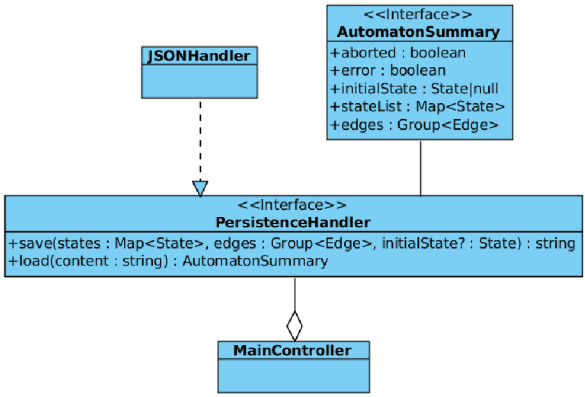

Figura 4: Classes relacionadas à persistência

São mostradas agora as estruturas que compõem o sub-sistema de persistência. A interface *PersistenceHandler* permite que novos formatos de salvar e/ou carregar arquivos possam também ser facilmente adicionadas ao sistema. A classe *JSONHandler* implementa essa interface, gerenciando arquivos numa notação compacta em JSON. A interface *AutomatonSummary* representa uma estrutura que contém todas as informações necessárias para que o *MainController* possa realizar o carregamento.

4.3.5 Definição formal

Uma funcionalidade bastante importante presente no sistema trata da exibição da definição formal do autômato sob edição. Para

[4] `https://en.wikipedia.org/wiki/Drop-down_list`

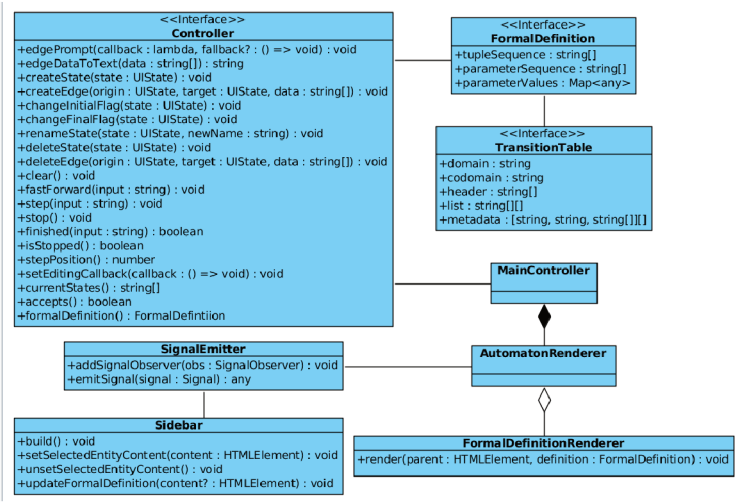

Figura 5: Classes relacionadas à definição formal

implementar tal funcionalidade, diversos componentes atuam em conjunto, como mostra a figura 5. Inicialmente, o usuário da aplicação realiza alguma modificação no autômato sob edição. Tal ação, após processada, faz com que o *AutomatonRenderer* chame o método *formalDefinition* do controlador, que então retorna uma estrutura que implementa a interface *FormalDefinition*. Essa estrutura é então transmitida ao método *render* de um renderizador de definições formais, que é uma instância da classe *FormalDefinitionRenderer*. Essa instância cria a representação visual da nova definição formal do autômato, que é então exibida na barra lateral.

4.3.6 Troca de mecanismo

Cada mecanismo do sistema pode oferecer menus personalizados, que são exibidos na barra lateral quando ele é selecionado. A implementação dessa funcionalidade compreende as relações expostas no diagrama abaixo. Inicialmente, o usuário clica no mecanismo que deseja na lista da barra lateral. Isso causa a chamada do método *changeMachine* da classe *System*, que por sua vez chama a função *changeMachine* de *Settings*. Essa função chama o método *onExit* do mecanismo atual, troca para o novo mecanismo e chama seu método *onEnter* (mais de-

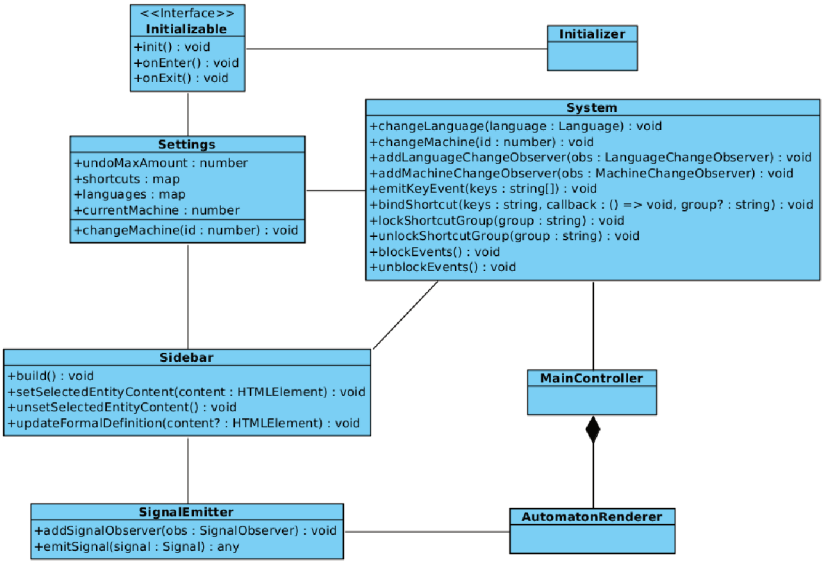

Figura 6: Classes relacionadas à troca de mecanismo

talhes sobre esses métodos no capítulo 5). A seguir, a barra lateral é atualizada para exibir os menus exclusivos do novo mecanismo. O *MainController* também é notificado de tal mudança e avisa o *AutomatonRenderer*, que comunica à barra lateral uma alteração de definição formal, seguindo as interações da seção anterior.

Os menus exclusivos de cada mecanismo são armazenados numa estrutura que é criada quando o sistema é carregado. Para isso, uma classe chamada *Initializer* percorre todos os mecanismos suportados e chama o método *init* de seus respectivos inicializadores (descritos também no capítulo 5), que populam uma estrutura com a lista de menus personalizados. Quando ocorre uma troca de mecanismo, a lista correspondente é percorrida e seus menus são exibidos na barra lateral.

4.3.7 Arquivos auxiliares automáticos

Como dito anteriormente, na seção 4.2, um dos arquivos mais importantes do sistema é o *makefile*. Esse arquivo, além de realizar a conversão TypeScript → JavaScript de todo o código da aplicação, também gera alguns arquivos auxiliares que tornam o sistema mais

facilmente extensível. Os seguintes arquivos são gerados, todos localizados na pasta *scripts/lists*:

- *ControllerList.ts*: simplifica o acesso do sistema ao controlador de cada mecanismo;
- *InitializerList.ts*: reúne o inicializador de cada mecanismo do sistema (mais detalhes sobre inicializadores no capítulo 5);
- *LanguageList.ts*: como dito na seção 4.3.3, este arquivo reúne todos os idiomas do sistema para que eles possam ser facilmente enumerados e lidos;
- *MachineList.ts*: contém uma *enum* com o nome de cada mecanismo suportado. É utilizado como base para indexar os controladores e inicializadores dos mecanismos e conectá-los entre si.

4.4 ALGORITMOS

Na seção anterior, foram apresentadas as principais classes do sistema e suas interações de modo a tornar possível diferentes funcionalidades presentes na aplicação. Esta seção visa expor uma análise em um nível mais baixo, ilustrando os principais algoritmos utilizados para a implementação dos mecanismos reconhecedores presentes através de pseudo-códigos comentados.

É importante ressaltar que desempenho não foi o principal motivador por trás das implementações utilizadas; ao invés disso, foi dado enfoque à integração de tais algoritmos com o restante da aplicação, em particular tendo em mente que simulação passo a passo é um requisito fundamental para o reconhecimento de sentenças.

4.4.1 Autômatos finitos

Os autômatos finitos são estruturas bastante simples, o que acaba resultando em uma implementação intuitiva e flexível. Um FA possui:

- uma lista de estados, implementada com um simples vetor com seus nomes;
- um alfabeto, armazenado como um mapeamento símbolo $\rightarrow$ quantidade de transições que utilizam este símbolo. Esta estrutura facilita a adição e remoção de símbolos ao alfabeto conforme transições são adicionadas ou removidas;

- uma tabela de transições, armazenada como um mapeamento da forma:
  índice do estado-origem → mapeamento(
  símbolo de entrada → conjunto de índices dos estados-alvo
)

- uma tabela de ε-transições, semelhante à anterior:
  índice do estado-origem → conjunto de índices dos estados-alvo

- um estado inicial, representado como o índice do estado inicial (ou -1, caso não haja nenhum definido);

- um conjunto de estados finais, armazenado como um conjunto de índices;

- um conjunto com os estados em que a máquina atualmente se encontra, representado também como um conjunto de índices.

Estes atributos são suficientes para que todas as operações desejadas de um FA possam ser implementadas. O algoritmo utilizado para realizar o reconhecimento de um símbolo da sentença de entrada é esquematizado a seguir.

Algoritmo 1: FA: Reconhecimento de um símbolo

```
reconhecer-símbolo (símbolo) :
    novosEstados ← {}
    for índice ∈ this.estadosAtuais do
        estadosAlvo ← this.transições[índice][símbolo]
        if estadosAlvo é não nulo then
            novosEstados ← novosEstados ∪ estadosAlvo
        end
    end
    this.ε-expansão(novosEstados)
    this.estadosAtuais ← novosEstados
```

O algoritmo da função auxiliar ε-expansão é apresentado no algoritmo 2.

Como visto, o algoritmo foi quebrado em duas funções: *reconhecer-simbolo*, que primeiro determina quais estados são alcançados sem considerar ε-transições e então chama a outra função; *ε-expansão*, que considera apenas ε-transições para completar a operação. Em termos de complexidade assintótica de tempo, a operação como um todo possui

Algoritmo 2: FA: ε-expansão

```
ε-expansão (listaDeEstados) :
    fila ← nova Fila()
    for índice ∈ listaDeEstados do
        fila.enqueue(índice)
    end
    while fila não está vazia do
        origem ← fila.dequeue()
        estadosAlvo ← this.ε-transições[origem]
        for índice ∈ estadosAlvo do
            if listaDeEstados não contém índice then
                listaDeEstados ← listaDeEstados∪{índice}
                fila.enqueue(índice)
            end
        end
    end
```

complexidade $O(m)$, sendo m o número de transições do grafo correspondente, uma vez que a quantidade de estados percorridos pelos laços *for* é limitada pela quantidade de transições. O laço *while*, por sua vez, percorre no máximo uma vez cada transição.

4.4.2 Autômatos de pilha

Autômatos de pilha são consideravelmente mais complexos que os autômatos finitos. Não-determinismo, em particular, exige uma reflexão mais aprofundada não só em termos de implementação como também a respeito de como apresentar esse comportamento na interface gráfica. A partir dessa reflexão, optou-se por implementar não-determinismo via *backtracking*. Esta técnica consiste em escolher uma transição dentre as possíveis de forma aleatória; caso o autômato eventualmente rejeite, ele retorna (faz *backtracking*) à configuração anterior a essa escolha e opta por outro caminho. O autômato só rejeita uma sentença quando todos os caminhos possíveis tiverem sido testados e rejeitados.

O não-determinismo via *backtracking* cria diversas complexidades na implementação que não estão presentes nos autômatos finitos. Assim como na seção anterior, inicialmente é mostrada a estrutura interna da classe que representa os autômatos de pilha. Um PDA possui:

- uma lista com os nomes dos estados;
- dois alfabetos: um de entrada e um de pilha. Armazenados da mesma forma que o alfabeto dos autômatos finitos, isto é, um mapeamento símbolo → quantidade de transições que utilizam o símbolo;
- uma tabela de transições, armazenada como um mapeamento da forma:
  índice do estado-origem → mapeamento(
  símbolo de entrada → mapeamento(
  símbolo de pilha → array de pares (índice-alvo, símbolos escritos)
)
- o índice do estado inicial;
- um conjunto de índices dos estados finais;
- índice do estado atual (no não-determinismo via *backtracking*, o autômato nunca está em mais de um estado em um dado momento);
- lista de símbolos que compõem a pilha;
- a entrada sendo lida;
- o número do passo de computação no ramo atual, utilizado para detectar a ocorrência de *backtracking*;
- uma lista de ações, que representa uma busca em profundidade no reconhecimento;
- uma *flag* que diz se o reconhecimento foi concluído;
- uma heurística de aceitação (estado final, pilha vazia ou ambos).

Assim como foi feito para os autômatos finitos, o algoritmo de reconhecimento de um símbolo da entrada será esquematizado em pseudocódigo. Porém, como neste caso a complexidade é consideravelmente maior, tal algoritmo foi quebrado em mais partes, conforme mostrado a seguir. Começa-se com uma análise do método principal.

Inicialmente, é determinado se o autômato está num estado de erro: em caso positivo, o método é imediatamente abortado. A seguir, é verificado se o autômato já está pronto para aceitar. Esta verificação

só possui resultado positivo quando o autômato aceita a sentença vazia e é ela que está sendo testada. O terceiro condicional verifica se um estado de erro foi atingido observando se não há nenhuma ação possível de ser realizada, isto é, todos os caminhos possíveis já foram testados e uma aceitação não foi atingida. Neste momento, a próxima ação é efetivamente executada e suas ramificações são adicionadas ao final da lista de ações, o que resulta em uma busca em profundidade. Se a árvore de ações fica vazia e o autômato está apto a aceitar, o reconhecimento é concluído.

Algoritmo 3: PDA: Reconhecimento de um símbolo de entrada

```
reconhecer-símbolo () :
    if this.erro() then
        return
    end
    árvoreDeAções ← this.árvoreDeAções
    if this.entrada está vazia e this.aceita() then
        this.concluído ← verdadeiro
        return
    end
    if árvoreDeAções está vazia then
        this.estadoAtual ← nulo
        this.entrada ← ""
        this.concluído ← verdadeiro
        return
    end
    próximaAção ← árvoreDeAções.últimoElemento
    this.processarAção(próximaAção)
    this.árvoreDeAções.pop()
    this.concluído ← falso
    possiveisAções ← this.calcularPossíveisAções()
    for ação ∈ possiveisAcoes do
        this.árvoreDeAções.push(ação)
    end
    if possíveisAções está vazio e this.entrada está vazia e
    this.aceita() then
        this.concluído ← verdadeiro
    end
```

Agora, o método *processarAção* é esquematizado. Ele é responsável por transferir os atributos de uma ação ao próprio autômato e então atualizar tanto a entrada quanto a pilha.

Algoritmo 4: PDA: Processamento de uma ação

```
processarAção (ação) :
    this.passo ←ação.passo
    this.entrada ←ação.entradaAtual
    this.pilha ←ação.pilhaAtual
    if ação.simboloLido ≠ ε then
        símbolo inicial de this.entrada é removido
    end
    this.pilha.pop()
    for símboloEscrito ∈ ação.símbolosEscritos do
        this.pilha.push(símboloEscrito)
    end
    this.estadoAtual ← ação.estadoAlvo
```

A seguir, tem-se o método (separado em duas partes) que determina e retorna as possíveis ações do ramo atual de computação. O método auxiliar *tratarSimbolo* popula uma lista com todas as ações possíveis a partir de um certo símbolo (que pode ser *epsilon*).

Algoritmo 5: PDA: Cálculo de ações possíveis

```
calcularPossíveisAções (ação) :
    resultado ← []
    if this.entrada não está vazia then
        this.tratarSímbolo(this.entrada[0], resultado)
    end
    this.tratarSímbolo(ε, resultado)
    return resultado
```

Algoritmo 6: PDA: Ações possíveis a partir de um símbolo

```
tratarSimbolo (símboolo, resultado) :
    if this.estadoAtual é nulo then
        return
    end
    transiçõesDisponíveis ←
        this.transições[this.estadoAtual]
    if não há transições disponíveis com o símbolo then
        return
    end
    transiçõesDisponíveis ←
        transiçõesDisponíveis[símbolo]
    topoDaPilha = this.pilha.top()
    if não há transições disponíveis com topoDaPilha then
        return
    end
    transicoesDisponíveis ←
        transiçõesDisponíveis[topoDaPilha]
    for transição ∈ transiçõesDisponíveis do
        resultado.push({
            passo : this.passo + 1,
            entradaAtual : this.entrada,
            pilhaAtual : this.pilha,
            símboloLido: símbolo,
            símbolosEscritos: transição[1],
            estadoAlvo: transição[0]
        })
    end
```

4.4.3 Autômatos linearmente limitados

Assim como os autômatos de pilha, os autômatos linearmente limitados são bem mais complexos que os autômatos finitos, principalmente no caso de haver não-determinismo. Novamente, optou-se por utilizar *backtracking* para representá-lo, reusando alguns elementos gráficos usados para PDAs para exibição na interface. Internamente, um LBA possui:

- uma lista com os nomes dos estados;
- dois alfabetos: um de entrada e um de fita. Armazenados da mesma forma que o alfabeto dos autômatos finitos e os dos autômatos de pilha;
- uma tabela de transições, armazenada como um mapeamento da forma:
  índice do estado-origem → mapeamento(
  símbolo lido → array(
  índice do estado-alvo,
  símbolo escrito,
  direção
)
)
- o índice do estado inicial;
- um conjunto de índices dos estados finais;
- o índice do estado atual;
- o estado atual da fita (conteúdo e posição do cabeçote);

- o número do passo de computação no ramo atual, utilizado para detectar a ocorrência de *backtracking*;

- o número de passo de computação de todo o reconhecimento atual, utilizado para detectar *loops* e assim prover parada garantida;

- o comprimento da entrada original, utilizado na verificação de *loops*;

- uma lista de ações, que representa uma busca em profundidade no reconhecimento;

- uma *flag* que diz se o reconhecimento foi concluído;

- uma *flag* que diz se a entrada foi aceita, utilizado em algumas otimizações internas.

Novamente, pseudo-códigos serão utilizados para esquematizar os algoritmos utilizados para efetuar o reconhecimento de sentenças. Assim como foi feito para PDAs, o algoritmo foi quebrado em partes devido à sua complexidade.

Primeiramente, é verificado se o reconhecimento atual já foi finalizado e, em caso positivo, ele é abortado. A próxima ação da árvore, caso ela não esteja vazia, é então executada e suas ramificações são adicionadas ao final da lista de ações, representando, assim como foi feito para PDAs, uma busca em profundidade. Se o autômato está apto a aceitar ou a quantidade limite de passos de computação foi atingida, o reconhecimento é encerrado.

Algoritmo 7: LBA: Execução de um passo de computação

```
próximo-passo () :
    if this.concluído then
        return
    end
    árvoreDeAções ← this.árvoreDeAções
    finalizado ← verdadeiro
    if árvoreDeAções está vazia then
        próximaAção ← árvoreDeAções.últimoElemento
        this.processarAção(próximaAção)
        this.árvoreDeAções.pop()
        possíveisAções ← this.calcularPossíveisAções()
        for ação ∈ possíveisAções do
            this.árvoreDeAções.push(ação)
        end
        finalizado ← falso
        if possíveisAções está vazio e this.aceita() then
            finalizado ← verdadeiro
        end
    end
    if this.atingiuLimite() then
        finalizado ← verdadeiro
    end
    if finalizado then
        this.concluído ← verdadeiro
        if this.aceita() then
            this.aceitação ← verdadeiro
        else
            this.estadoAtual ← nulo
        end
    end
```

O método *processarAção* é esquematizado a seguir. Sua lógica é bastante semelhante ao método de mesmo nome dos PDAs: ele copia os atributos da ação para o próprio autômato e atualiza alguma estrutura auxiliar, que, no caso dos LBAs, é a fita.

Algoritmo 8: LBA: Processamento de uma ação

```
processarAção (ação) :
    this.fita.carregar(ação.fitaAtual)
    this.passosDeComputação ←
        ação.passosDeComputação
    this.estadoAtual ← ação.estadoAtual
    this.fita.escrever(ação.escreverNaFita)
    this.fita.moverCabeçote(ação.direçãoDeMovimento)
    this.passosDeComputação++
```

Assim como foi feito para os PDAs, há um método dedicado a determinar as próximas ações possíveis do ramo atual de computação. Ele também foi dividido em duas partes, as quais possuem os mesmos nomes e objetivos.

Algoritmo 9: LBA: Cálculo de ações possíveis

```
calcularPossíveisAções (ação) :
    resultado ← []
    entrada ← this.fita.ler()
    if entrada é indefinida then
        this.tratarSímbolo(entrada, resultado)
    else
        this.tratarSímbolo("_", resultado)
    end
    return resultado
```

Algoritmo 10: LBA: Ações possíveis a partir de um símbolo

```
tratarSímbolo (símbolo, resultado) :
    if this.estadoAtual é nulo then
        return
    end
    if não há transições a partir do estado atual then
        return
    end
    transiçõesDisponíveis ←
        this.transições[this.estadoAtual]
    if não há transições disponíveis com o símbolo then
        return
    end
    transiçõesDisponíveis ←
        transiçõesDisponíveis[símbolo]
    for transição ∈ transiçõesDisponíveis do
        resultado.push({
            fitaAtual: this.fita.salvar(),
            passosDeComputação:
                this.passosDeComputação,
            escreverNaFita: transição.escreverNaFita,
            direçãoDeMovimento: transição.direção,
            índicePasso: this.índicePasso,
            estadoAlvo: transição.estadoAlvo
        })
    end
```

Neste capítulo, as funcionalidades gerais do sistema proposto foram apresentadas, além de sua estrutura e principais algoritmos. Há, ainda, um aspecto importante a ser tratado: extensibilidade. Como mencionado anteriormente, diversas ferramentas similares disponíveis não são desenvolvidas com extensibilidade em mente, o que muitas vezes inviabiliza a adição de novas funcionalidades.

A aplicação proposta, por outro lado, possui a extensibilidade como ponto-chave desde o início de seu desenvolvimento. O próximo capítulo visa apresentar as maneiras pelas quais o sistema pode ser estendido.

5 EXTENSIBILIDADE

O sistema foi projetado de modo a permitir grande extensibilidade. Neste capítulo, são mostrados os procedimentos necessários para estendê-lo de diferentes maneiras.

5.1 ADIÇÃO DE MECANISMOS

A pasta *scripts/machines* do sistema contém uma pasta para cada mecanismo reconhecedor suportado. Cada pasta contém os seguintes arquivos (como exemplo, são mostrados os arquivos da pasta *FA*, que corresponde a um autômato finito):

- FAController.ts: uma classe que implementa a interface *Controller*, como descrito mais detalhadamente na seção 4.3;
- FA.ts: classe que encapsula toda a lógica do mecanismo em si, ignorando aspectos da interface;
- initializer.ts: *namespace* que deve conter três funções: *init*, responsável por inicializar componentes personalizados na barra lateral; *onEnter*, chamado quando este mecanismo se torna o atual; *onExit*, chamado quando este mecanismo deixa de ser o atual.

Para adicionar um novo mecanismo reconhecedor ao sistema, basta criar um novo diretório com o nome dele que siga a mesma convenção de arquivos, substituindo "FA" pelo nome correspondente.

Outro aspecto que pode ser alterado é a ordem em que os mecanismos são exibidos na interface. Para isso, utiliza-se o arquivo *priority.txt*, localizado na mesma pasta dos mecanismos, isto é, em *scripts/machines*. Esse arquivo contém o nome de mecanismos reconhecedores na ordem em que eles devem ser exibidos. Não é necessário que todos os nomes estejam presentes nele; nomes faltantes são exibidos depois dos presentes. O mecanismo selecionado quando a aplicação é iniciada é sempre o primeiro exibido.

5.2 ADIÇÃO DE IDIOMAS

Para adicionar um novo idioma ao sistema, inicialmente entre na pasta *scripts/languages/* e duplique o arquivo *English.ts*, renomeando

a cópia para um nome qualquer com a extensão ".ts"(aconselha-se usar o nome do idioma).

Dentro do novo arquivo, mude o nome do *namespace* para um nome qualquer (novamente, aconselha-se usar o nome do idioma).

A variável *strings* contém as traduções propriamente ditas, basta editá-las de acordo com o idioma que se está adicionando e, em seguida, executar *make* na pasta raiz do projeto.

O idioma adicionado aparecerá automaticamente no sistema em sua próxima execução. Note que em algumas traduções é utilizado um símbolo de porcentagem (%) para se referir a uma string que é substituída em tempo de execução e não requer tradução manual. Nesses casos, é altamente aconselhável também usar esse símbolo na respectiva tradução.

No capítulo anterior, o sistema foi apresentado principalmente em termos de sua engenharia de software. Neste, novamente uma visão técnica foi apresentada, visto que o usuário final da aplicação não necessita saber como estendê-la para poder utilizá-la. Com isso em mente, o próximo capítulo dedica-se a cobrir a lacuna restante: apresentar o sistema desenvolvido sob a ótica do usuário final, para que este possa utilizar todas as suas funcionalidades.

6 INSTRUÇÕES DE USO

Dada a quantidade de funcionalidades e possibilidades providas pelo sistema proposto, torna-se importante reunir todas as informações necessárias para que usuários possam utilizá-lo completamente, sanando possíveis dúvidas que podem surgir durante o uso da ferramenta. Assim, este capítulo propõe-se a detalhar como utilizar cada uma das funcionalidades presentes.

6.1 ALTERAÇÃO DE IDIOMA

Na figura 7, mostrada abaixo, é exibido o primeiro menu da barra lateral, que contém as configurações de sistema. A primeira configuração deste menu refere-se ao idioma da aplicação. Ao clicar nesta lista *drop-down*, que por padrão exibe a língua atual, os demais idiomas suportados são exibidos. Quando algum deles é clicado, uma janela de confirmação é exibida, como mostrado na figura 8.

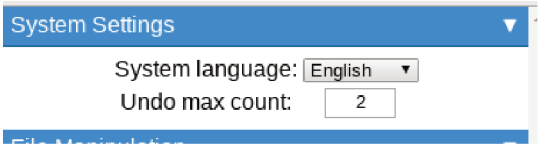

Figura 7: Menu de configurações de sistema

Figura 8: Confirmação de alteração de idioma

6.2 ALTERAÇÃO DE MECANISMO RECONHECEDOR

O terceiro menu da barra lateral contém uma lista de botões, como ilustra a figura 9, na qual o botão correspondente ao mecanismo atualmente selecionado é desabilitado. O usuário pode trocar de mecanismo facilmente clicando no desejado. Caso haja algum autômato sob edição, é exibida uma janela de confirmação (fig. 10), visto que ele será perdido ao realizar a troca.

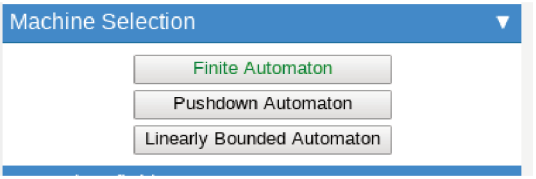

Figura 9: Lista de mecanismos do sistema

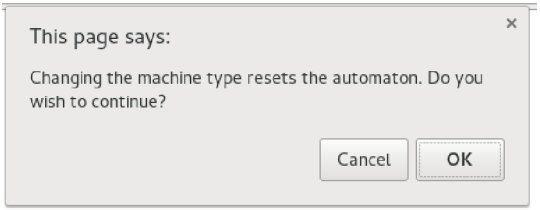

Figura 10: Confirmação de alteração de mecanismo

6.3 EDIÇÃO VIA DIAGRAMA

A região principal do sistema é dedicada à edição via diagrama. Todas as formas possíveis de interagir com ela possuem botões correspondentes na barra lateral que realizam a mesma função, como mostrado na figura 11. As ações possíveis são:

- Criar estado: clique duplo na posição-alvo. Um diálogo requisitando o nome do estado aparece e então ele é efetivamente criado. O botão correspondente (*create state*) cria o estado no canto superior esquerdo da região principal.

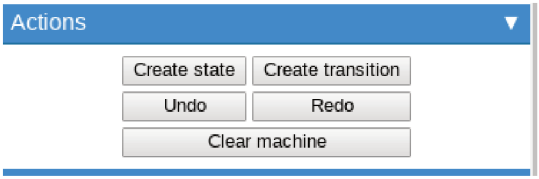

Figura 11: Menu de ações

- Criar transição: clique com o botão direito no estado-origem e, em seguida, clique no estado-alvo. Será exibido um diálogo requisitando informações sobre a transição (que variam conforme o mecanismo). O botão correspondente (*create edge*) pergunta quais os estados envolvidos e então exibe o mesmo diálogo de informações de transição.

- Tornar um estado o inicial/torná-lo não-inicial: primeiro, clique no estado desejado. Pressione o botão "i"para alternar se ele é inicial ou não. Outra possibilidade é, após clicar no estado, clicar no botão *toggle* no menu lateral de item selecionado (fig. 12) ao lado do campo *is initial*.

- Tornar um estado final/torná-lo não-final: similar ao item anterior. Clique no estado e então pressione o botão "f". Outra possibilidade é, no mesmo menu lateral do item anterior, clicar no botão *toggle* do campo *is final*.

- Renomear estado: semelhante aos anteriores. Clique no estado e então no botão *rename* do menu lateral de item selecionado. Um diálogo aparecerá requisitando o novo nome.

- Remover estado: Clique no estado e então no botão *Delete state* do menu de item selecionado. Também é possível utilizar o botão "delete"após clicar no estado para excluí-lo.

- Alterar a origem, destino ou conteúdo de uma transição: Clique na aresta do grafo que deseja-se alterar diretamente através dele ou através da tabela de transições exibida na definição formal. No menu lateral de item selecionado (fig. 13), utilize o botão *change* do campo desejado para alterá-lo.

- Remover transição ou aresta: Selecione a aresta desejada do mesmo modo do item anterior. Para remover uma das transições da

aresta, selecione-a na lista e utilize o botão *Delete selected transition*. Para excluir a aresta em si clique em *Delete all transitions*.

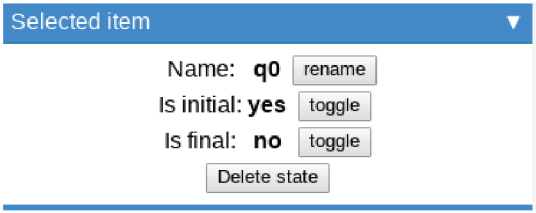

Figura 12: Menu de item selecionado (estado)

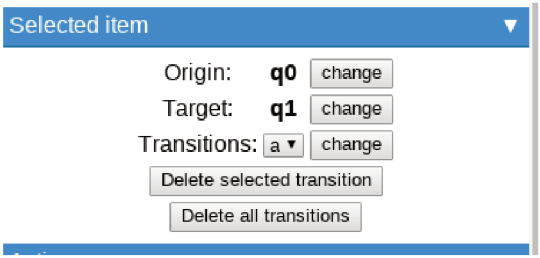

Figura 13: Menu de item selecionado (aresta)

- Desfazer última ação: pressione *Ctrl+Z* ou clique em *Undo* no menu de ações.
- Refazer último "desfazer": pressione *Ctrl+Y* ou clique em *Redo* no menu de ações.
- Limpar máquina: pressione *C* ou clique em *Clear machine* no menu de ações.

6.4 EDIÇÃO VIA TABELA DE TRANSIÇÕES

Ao passar o mouse sobre a tabela de transições, a transição correspondente é realçada no diagrama do autômato. Pode-se, então, clicar nela para que suas informações apareçam na barra lateral, onde podem ser livremente editadas. A criação de estados e transições pode ser feita através dos botões descritos na seção anterior, de modo que nenhuma interação com o diagrama é obrigatória para a criação de autômatos.

6.5 RECONHECIMENTO

Como visto no capítulo 4, três tipos de reconhecimento são suportados: passo a passo, rápido e múltiplo. As subseções a seguir descrevem como utilizar cada um.

6.5.1 Reconhecimento passo a passo

Para utilizar reconhecimento passo a passo, inicialmente digita-se a palavra a ser testada na caixa de texto, conforme exibido abaixo.

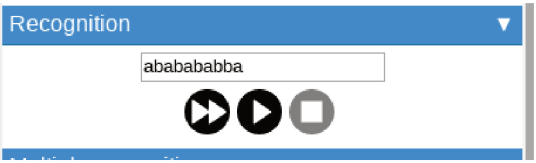

Figura 14: Menu de reconhecimento

O botão central da imagem, com um ícone de *play*, representa o reconhecimento passo a passo. Ao clicá-lo pela primeira vez (ou ao pressionar a tecla *enter*), é carregada a configuração inicial da máquina. Cliques sucessivos (ou *enters* sucessivos) então continuam o processo de reconhecimento, um passo de cada vez, atualizando a configuração atual da máquina. A qualquer momento, o usuário pode utilizar o botão *stop* para encerrar o reconhecimento atual. Clicar na caixa de texto automaticamente realiza um *stop*.

6.5.2 Reconhecimento rápido

Assim como no caso de reconhecimento passo a passo, o usuário inicialmente digita a palavra desejada na caixa de texto. Nas figuras 15 e 16, pressiona-se o botão com ícone de *fast forward*, que representa o reconhecimento rápido. Este botão revela o estado final do autômato e quaisquer outras propriedades relevantes ao reconhecimento o mais rápido possível. Assim como no reconhecimento passo a passo, o botão *stop* encerra o reconhecimento, removendo da tela todas as informações produzidas por ele.

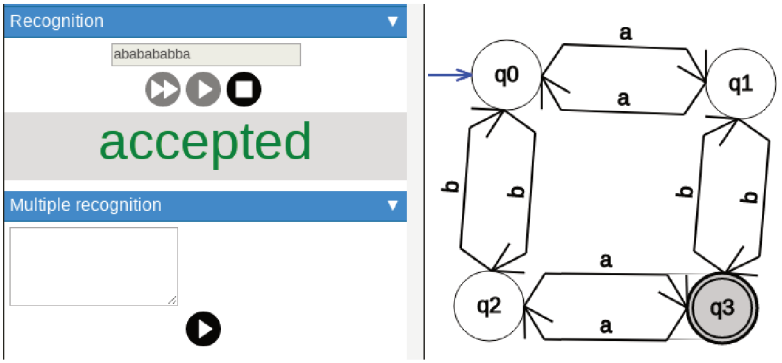

Figura 15: Reconhecimento rápido em um FA

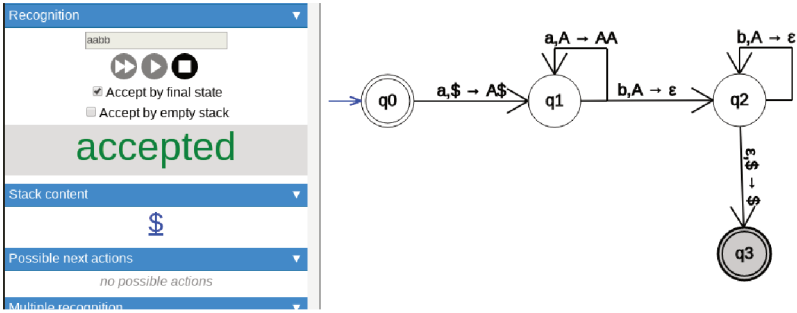

Figura 16: Reconhecimento rápido em um PDA

6.5.3 Reconhecimento múltiplo

O reconhecimento múltiplo encontra-se num menu separado do passo a passo e do rápido, conforme ilustra a figura 17. Para utilizá-lo, digita-se as palavras a serem testadas na área de texto, separando-as por quebras de linha (fig. 18). O botão com ícone de *play* abaixo da mesma revela o estado do reconhecimento (aceitação, rejeição ou quaisquer outros providos pelo mecanismo) de cada uma.

Figura 17: Reconhecimento múltiplo

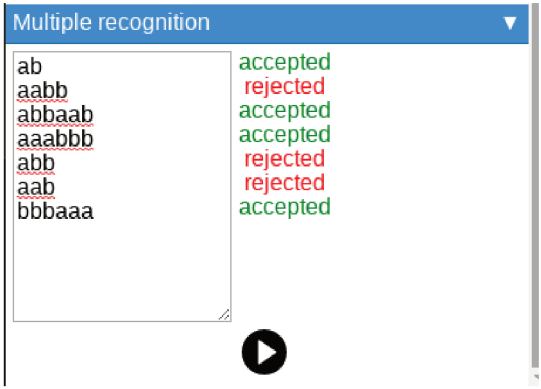

Figura 18: Teste com reconhecimento múltiplo

6.6 SALVAR E CARREGAR AUTÔMATOS

Na barra lateral encontra-se um menu de persistência, como mostrado na figura 19.

Figura 19: Menu de persistência

Para salvar o mecanismo sob edição, basta clicar no botão *save*. Uma caixa de diálogo abrirá requisitando ao usuário um nome[1]. Para

[1]Dependendo do navegador e do sistema operacional do usuário, este processo pode ser levemente diferente.

carregar mecanismos salvos, clica-se em *open*, que restaura não só os estados e transições do mecanismo como também suas posições.

Neste capítulo, foram apresentadas instruções de como utilizar cada uma das funcionalidades da ferramenta proposta, visando, assim, sanar eventuais dúvidas que usuários podem ter enquanto a utilizam, principalmente em seus primeiros contatos com ela. A seguir, apresentam-se as consideraçoes finais e possíveis trabalhos futuros.

7 CONCLUSÃO

Neste trabalho, foi desenvolvido um sistema web de caráter didático focado na simulação do reconhecimento de sentenças em autômatos finitos, autômatos de pilha e autômatos linearmente limitados. Ferramentas de propósito semelhante foram analisadas sob critérios como tipos de reconhecimento suportados, a forma pela qual os autômatos são criados, quais os autômatos suportados, entre outros. A partir dessa análise, foram definidas as funcionalidades que seriam importantes à nova ferramenta para atingir o objetivo a que se propôs.

A partir da lista de funcionalidades propostas, o novo sistema foi então descrito em detalhes. Seus detalhes arquiteturais foram mostrados e explicados através de diagramas de classes focados em diferentes aspectos da aplicação. Os principais algoritmos utilizados para efetuar o reconhecimento foram apresentados com pseudo-códigos.

A seguir, mostrou-se as principais maneiras pelas quais o sistema desenvolvido pode ser estendido: adição de mecanismos e adição de idiomas. Por fim, foi apresentado um manual de uso detalhando todas as funcionalidades providas pela aplicação.

Na seção 3.9 foram expostas duas tabelas comparativas: uma com funcionalidades genéricas (disponibilidade *online*, persistência e restrições do alfabeto) e outra com características relacionadas aos mecanismos suportados por cada uma das ferramentas. Em relação à primeira, todas as funcionalidades citadas foram atendidas: a ferramenta é online[1], possui suporte completo a salvar e carregar autômatos e não possui restrições no alfabeto, com uma exceção: para LBA, convencionou-se utilizar letras minúsculas e números para símbolos de entrada e o caractere "_"para representar um símbolo *branco*.

Dois métodos de entrada são suportados: via diagrama e via tabela de transições. O suporte a expressões regulares não foi implementado devido a questões de tempo. As três formas de reconhecimento mencionadas na segunda tabela foram implementadas: passo a passo, rápido e múltiplo. Os algoritmos de determinização e minimização também não foram concluídos a tempo.

No caso dos PDAs, todas as características apresentadas na segunda tabela foram concluídas. Para LBAs, optou-se por implementar entrada via tabela ao invés de via código, estabelecendo assim um diferencial sobre todas as ferramentas similares analisadas. A variante escolhida de máquina de Turing possui fita única.

[1] Disponível em: `https://ghabriel.github.io/AutomatonSimulator/`

7.1 TRABALHOS FUTUROS

Embora as funcionalidades propostas tenham sido todas implementadas, um dos objetivos do sistema desenvolvido é ser extensível e, como tal, há diversas possibilidades de trabalhos futuros.

Primeiramente, novos tipos de mecanismos reconhecedores podem ser adicionados. Os três já suportados foram escolhidos por sua relevância nas disciplinas de Teoria da Computação e Linguagens Formais, mas certamente existem outros que agregariam ainda mais valor à ferramenta. Também pode-se implementar mecanismos já existentes mas com técnicas diferentes de reconhecimento, como um PDA não-determinístico que admita mais de um estado atual ao mesmo tempo ao invés de utilizar *backtracking*, por exemplo.

Outra possibilidade refere-se à adição de novos idiomas. Português e inglês já são suportados; adicionar suporte a outros idiomas aumentaria a acessibilidade da aplicação.

Também pode ser interessante adicionar suporte a outros formatos de arquivo, tanto para abrir como para salvar. Em particular, pode-se permitir uma compatibilidade com outros sistemas de propósito similar, em que o usuário poderia criar um autômato no sistema proposto e abri-lo em outra ferramenta e vice-versa. Assim, funcionalidades não suportadas em uma das aplicações poderiam ser utilizadas na outra sem necessidade de criar o mesmo autômato múltiplas vezes.

Outras extensões possíveis tratam da adição de operações com autômatos, como, por exemplo, determinização, e adicionar um gerador de sentenças aceitas à barra lateral do sistema, o que facilitaria a tarefa de verificar a corretude dos autômatos produzidos pelo usuário. Esse gerador poderia limitar o tamanho das sentenças geradas ou restringir seu formato utilizando, por exemplo, expressões regulares.

Por fim, é importante avaliar a eficácia real da aplicação no ensino. Devido a questões de tempo, não foi possível fazê-lo.

REFERÊNCIAS

BURCH, C. **Automaton Simulator**. Disponível em: <http://cburch.com/proj/autosim/>. Acesso em: 16 março 2017.

DICKERSON, K. **Automaton Simulator**. Disponível em: <http://automatonsimulator.com/>. Acesso em: 15 março 2017.

GAMMA, E. et al. **Design Patterns: Elements of Reusable Object-Oriented Software**. First editon. [S.l.]: Addison Wesley, 1994.

HOPCROFT, J. E.; MOTWANI, R.; ULLMAN, J. D. **Introduction to Automata Theory, Languages, and Computation**. Second editon. [S.l.]: Addison Wesley, 2000.

MORPHETT, A. **Turing machine simulator**. 2014. Disponível em: <http://morphett.info/turing/turing.html>. Acesso em: 16 março 2017.

RENDELL, P. **Turing Machine Simulator**. Disponível em: <http://rendell-attic.org/gol/TMapplet/>. Acesso em: 15 março 2017.

RODGER, S. H. **JFLAP**. Disponível em: <http://jflap.org/>. Acesso em: 17 março 2017.

SIPSER, M. **Introduction to the Theory of Computation**. Second editon. [S.l.]: Course Technology, 2006.

TypeScript. **TypeScript - JavaScript that scales.** Disponível em: <https://www.typescriptlang.org/>. Acesso em: 03 novembro 2016.

UGARTE, M. **Online Turing Machine Simulator**. 2015. Disponível em: <https://turingmachinesimulator.com/>. Acesso em: 15 março 2017.

WHITE, T. M. **jFAST - a Java Finite Automata Simulator**. Disponível em: <http://www46.homepage.villanova.edu/timothy.m.white/>. Acesso em: 15 março 2017.

ZUZAK, I.; JANKOVIC, V. **FSM Simulator**. Disponível em: <http://ivanzuzak.info/noam/webapps/fsm_simulator/>. Acesso em: 15 março 2017.

ANEXO A – Artigo

Simulador de Autômatos e Máquinas de Turing

Ghabriel C. Nunes[1]

[1]Departamento de Informática e Estatística
Universidade Federal de Santa Catarina (UFSC) – Florianópolis, SC – Brasil

ghabriel.nunes@grad.ufsc.br

Abstract. *The recognition of sentences by recognizing mechanisms is one of the most important subjects in disciplines such as Theory of Computation and Formal Languages. However, there's a lack of high quality systems to simulate such recognition, which frequently makes students have questions about these subjects, resulting in a more difficult learning process. We propose a new web application capable of fulfilling these needs, aiming to improve the student comprehension about these subjects.*

Resumo. *O reconhecimento de sentenças por mecanismos reconhecedores é um dos assuntos mais importantes das disciplinas de Teoria da Computação e Linguagens Formais. Apesar disso, há uma escassez de sistemas de qualidade para simular tal reconhecimento, o que muitas vezes leva os alunos a permanecerem com dúvidas a respeito desses conteúdos, dificultando a aprendizagem. Propõe-se, então, o desenvolvimento de um sistema web capaz de suprir essa necessidade, visando melhorar a compreensão dos alunos acerca desses conteúdos.*

1. Introdução

A falta de bons simuladores de mecanismos reconhecedores em geral muitas vezes faz com que estudantes de Teoria da Computação e Linguagens Formais tenham dificuldades no aprendizado das máquinas estudadas nessas disciplinas e consequentemente nos aspectos conceituais e práticos. As soluções disponíveis atualmente se encaixam em pelo menos uma das seguintes categorias:

- apresentam difícil utilização, deixando o usuário confuso sobre como projetar e testar os autômatos que deseja. Em alguns casos, além da interface ser altamente complexa, não é fornecido um manual de uso;
- são incompletas, permitindo ao usuário projetar apenas um subconjunto dos autômatos que deveriam ser possíveis com os mecanismos disponibilizados, ou então não permite simular o reconhecimento de qualquer sentença. Muitas vezes isso é causado por limitações excessivas no alfabeto de entrada;
- apresentam problemas de funcionamento, em alguns casos levando a reconhecimentos incorretos ou a travamentos no programa, podendo fazer com que autômatos projetados pelo usuário tenham que ser reconstruídos manualmente.
- são difíceis de estender, isto é, é difícil - ou até impossível sem reescrever boa parte da aplicação - adicionar funcionalidades ao sistema.

2. Objetivos

Tendo em mente os problemas encontrados nos simuladores existentes, juntamente às dificuldades dos alunos previamente citadas, propõe-se o desenvolvimento de um sistema web focado em mitigar esses problemas, provendo aos estudantes uma ferramenta capaz de auxiliá-los no aprendizado de linguagens recursivas, isto é, linguagens com parada garantida, estabelecendo os seguintes objetivos específicos:

2.1. Objetivos Específicos

Possibilitar a construção e manipulação de autômatos finitos, autômatos de pilha e autômatos linearmente limitados. O sistema deve focar principalmente na solução dos problemas e das dificuldades dos sistemas existentes, atendendo às seguintes propriedades:

- **Facilidade de uso** - o uso do sistema deverá ser intuitivo, dispensando a necessidade do uso de manuais para projetar qualquer autômato, embora neste trabalho seja incluído um manual para sanar eventuais dúvidas;
- **Completude** - o sistema deverá evitar restrições excessivas no alfabeto de entrada e no conteúdo das transições, sem deixar de respeitar as limitações inerentes dos mecanismos desenvolvidos;
- **Corretude** - o sistema deverá realizar os reconhecimentos corretamente e evitar travamentos (levando em conta a complexidade do autômato que se esteja manipulando, é claro).
- **Extensibilidade** - o sistema deverá ser capaz de ser facilmente estendido, ou seja, deve ser possível adicionar novos mecanismos reconhecedores ao sistema e também novas funcionalidades aos já existentes.

A ferramenta proposta deve permitir ao usuário projetar autômatos diretamente de forma gráfica para, em seguida, testar o reconhecimento de sentenças, seja passo a passo ou rapidamente. Também deve ser possível realizar o download do sistema para que se possa executá-lo localmente sem necessidade de acesso à Internet, o que implica que nenhuma linguagem *server-side* pode ser usada.

3. Trabalhos correlatos

Em qualquer projeto de software, é natural buscar por aplicações já existentes que se proponham a resolver problemas similares. Tal busca permite a obtenção de ideias a respeito de possíveis funcionalidades, comportamento e organização da interface, entre diversos outros aspectos, além de permitir a realização de uma análise das limitações que tais sistemas possuem. Assim, esta seção apresenta alguns sistemas similares ao proposto, analisando suas funcionalidades oferecidas e aspectos como facilidade de uso.

Ao final da seção, são apresentadas tabelas que sintetizam a análise realizada, ilustrando mais claramente as diferenças entre os trabalhos correlatos sob diferentes aspectos, além de mostrar as funcionalidades suportadas pela nova ferramenta que será introduzida na seção seguinte.

3.1. Automaton Simulator (K. Dickerson)

Este sistema online[1], criado por Kyle Dickerson, apresenta uma das interfaces gráficas mais intuitivas entre os trabalhos analisados. É online e apresenta suporte total a sal-

[1] `http://automatonsimulator.com/`

var e carregar autômatos. Quanto a FA e PDA, apresenta bom suporte (incluindo não-determinismo), permitindo edição gráfica, reconhecimento passo-a-passo e reconhecimento múltiplo, porém não suporta edição via tabela de transições nem via expressão regular. Esta ferramenta não possui suporte a nenhuma variante de máquina de Turing.

3.2. FSM Simulator (I. Zuzak e V. Jankovic)

Esta ferramenta[2] online, produzida por Ivan Zuzak e Vedrana Jankovic, é focada na criação de autômatos finitos a partir de expressões regulares, além de permitir tal geração via código numa linguagem própria de alto nível de sintaxe simples. Não permite salvar nem carregar autômatos. Através de expressões regulares, é gerado um NFA equivalente. É possível criar um DFA apenas através da linguagem própria do sistema. Permite somente reconhecimento passo-a-passo, além de limitar-se a edição via expressão regular.

3.3. Turing machine simulator (A. Morphett)

Ferramenta[3] online criada por Anthony Morphett em 2014 focada em máquinas de Turing de fita única. A entrada do programa se dá somente através de uma linguagem própria cuja sintaxe é rebuscada, dificultando seu uso. Possui suporte a salvar e carregar máquinas através de *links* de acesso. Esta ferramenta permite reconhecimento passo-a-passo e reconhecimento rápido das seguintes variantes de máquinas de Turing (somente fita única):

- Determinística, com fita infinita em ambas as direções;
- Determinística, com fita semi-infinita (isto é, infinita em apenas uma direção;
- Não-determinística, na qual, caso haja mais de uma regra compatível com a configuração atual, uma é escolhida aleatoriamente para ser usada.

3.4. Online TM Simulator (M. Ugarte)

Sistema[4] online criado por Martin Ugarte em 2015. A entrada do programa se dá através de uma linguagem própria com sintaxe simples, o que torna seu uso simples embora não possibilite edição via diagrama. Possui suporte a salvar e carregar máquinas através de *links* de acesso ou localmente (esta última exige cadastro). Esta ferramenta suporta somente máquinas de Turing (e somente determinísticas). Apesar disso, ao contrário da anterior, esta suporta até 3 fitas, incluindo reconhecimento passo-a-passo e reconhecimento com controle de velocidade.

3.5. Turing Machine Simulator (P. Rendell)

Assim como as duas ferramentas anteriores, esta[5] também é online e possui enfoque em máquinas de Turing. Criada por Paul Rendell, esta aplicação permite entrada através de tabela de transições, que, embora simples, apresenta usabilidade inferior a edição via diagrama. Além disso, não possui suporte a salvar nem carregar máquinas. Esta ferramenta limita-se a máquinas de Turing determinísticas *single tape* com fita limitada em 200 posições para cada lado. Permite reconhecimento passo-a-passo e reconhecimento rápido assim como as aplicações anteriores, e, assim como elas, permite edição apenas via código.

[2] `http://ivanzuzak.info/noam/webapps/fsm_simulator/`
[3] `http://morphett.info/turing/turing.html`
[4] `https://turingmachinesimulator.com/`
[5] `http://www.rendell-attic.org/gol/TMapplet/`

3.6. Automaton Simulator (C. Burch)

Este sistema[6], desenvolvido por Carl Burch, permite construir máquinas facilmente, porém sem muita agilidade devido à falta de atalhos de teclado para alternar de função, além de possuir alfabeto limitado a {a, b, c, d}. Ao contrário dos trabalhos correlatos descritos até agora, esta requer *download*, visto que não é online. Apresenta suporte completo a salvar e carregar máquinas construídas. Possui bom suporte a DFA, NFA, DPDA e máquinas de Turing *single tape*.

3.7. jFAST (T. M. White)

Esta ferramenta[7], produzida por Timothy M. White em 2006, exige a definição do alfabeto de entrada separadamente. Não proporciona muita agilidade devido à falta de atalhos de teclado para alternar de função e por criar muitas janelas auxiliares durante a construção. Requer *download*, pois não é online. Suporte completo a salvar e carregar máquinas construídas. Apresenta diversos problemas de usabilidade e *bugs*, mas suporta FA, PDA e máquinas de Turing *single tape*.

3.8. JFLAP (S. H. Rodger)

Esta ferramenta[8], produzida originalmente por Susan H. Rodger em 2003, é a mais completa entre os trabalhos correlatos analisados. Apresenta fácil utilização, apesar de exigir frequentes trocas de função para construir máquinas, embora apresente atalhos de teclado. Suporta FA, PDA e máquinas de Turing com até 5 fitas. Não suporta edição via outros meios que não via diagrama. O reconhecimento rápido não funciona corretamente para máquinas *multi tape* em alguns casos, nos quais o programa encontra caminhos bem sucedidos que na verdade não o são.

Como visto, as ferramentas analisadas possuem grande variedade de funcionalidades, embora existam algumas interessantes que nenhuma delas suporta como, por exemplo, suporte a autômatos linearmente limitados. Assim, na próxima seção será descrito uma nova ferramenta que, além de prover boa parte das funcionalidades apresentadas, visa, em particular, suportar algumas menos comuns, como os LBAs.

4. Nova ferramenta

Na seção anterior, foram listados diversos critérios e características para avaliar aplicações relacionadas ao reconhecimento de sentenças em diversos mecanismos reconhecedores. Nesta seção, é descrito um novo sistema com objetivo semelhante, focado principalmente em facilidade de uso, variedade de funcionalidades e extensibilidade.

4.1. Funcionalidades gerais do sistema

A aplicação suporta três mecanismos reconhecedores[9]: autômatos finitos, tanto determinísticos quanto não-determinísticos; autômatos de pilha, tanto determinísticos quanto não-determinísticos e autômatos linearmente limitados, que, como visto na seção 2, compreendem um subconjunto das máquinas de Turing. Para cada um desses mecanismos, são

[6] `http://www.cburch.com/proj/autosim/`
[7] `http://www46.homepage.villanova.edu/timothy.m.white/`
[8] `http://www.jflap.org/`
[9] É possível adicionar mais mecanismos reconhecedores ao sistema.

suportados dois tipos de entrada, isto é, duas diferentes maneiras de se criar um autômato de algum dos tipos mencionados:

- Entrada via diagrama: neste tipo de entrada, o usuário cria diretamente um grafo que representa o autômato sendo desenvolvido. Costuma ser a principal forma de entrada, por ser mais intuitiva devido a seu aspecto gráfico;
- Entrada via tabela de transição: o usuário pode alterar diretamente a tabela de transições de um autômato, que atualiza automaticamente o grafo correspondente.

A qualquer momento, o usuário pode realizar um reconhecimento sobre o autômato criado. Três tipos de reconhecimento são suportados:

- Passo a passo: o usuário pode observar cada caractere da entrada sendo lido pelo autômato, um de cada vez;
- Rápido: uma entrada é fornecida e um *feedback* é retornado em um único passo, mostrando se a entrada foi aceita ou não. Ao chegar ao término do reconhecimento, o autômato é exibido em sua configuração final;
- Múltiplo: execução em paralelo de múltiplas entradas. É informado quais entradas foram aceitas e quais foram rejeitadas.

No caso de autômatos de pilha, o usuário pode escolher se deseja que o reconhecimento seja feito via estado final, via pilha vazia ou ambos. É permitido realizar múltiplos *pushs* numa única transição.

A aplicação apresenta suporte completo ao salvamento e carregamento de autômatos. Os arquivos salvos ficam em formato JSON, que é naturalmente mais compacto e simples de manusear que XML. Adicionalmente, foi utilizada uma notação enxuta visando minimizar redundâncias para diminuir ainda mais o tamanho do arquivo.

Para facilitar o uso da aplicação, todas as funcionalidades do sistema possuem botões correspondentes na interface, além de boa parte delas também possuir atalhos de teclado associados. Assim, mesmo usuários que estejam utilizando a ferramenta pela primeira vez possuem fácil acesso a 100% das funcionalidades da aplicação.

O sistema proposto possui ainda suporte a multi-idiomas, sendo este um dos aspectos de fácil extensibilidade. Por padrão, os idiomas português e inglês estão disponíveis, podendo ser alternados livremente a critério do usuário.

4.2. Metodologia de implementação

O software proposto foi desenvolvido com a linguagem de programação TypeScript, que, por sua vez, gera códigos em JavaScript. Por ser um sistema web, a interface - na qual conteúdos dinâmicos serão criados via TypeScript - será feita com HTML e CSS. Outro ponto importante trata do uso de um *makefile*, que, embora não seja usado pelo usuário final, é necessário para realizar modificações no código da aplicação.

A linguagem TypeScript foi escolhida por dois motivos principais. Primeiramente, ela é uma linguagem orientada a objetos, que é um paradigma familiar que permite o desenvolvimento de sistemas com alta manutenibilidade e extensibilidade. Além disso, tal linguagem é *client-side*, o que permite que o usuário do sistema realize *download* da ferramenta para utilizá-la localmente, sem necessidade de acesso à Internet.

4.3. Exemplos de uso

Esta sub-seção dedica-se a mostrar alguns exemplos de uso da ferramenta. Na barra lateral da aplicação encontra-se um menu de reconhecimento, como ilustrado na figura 1.

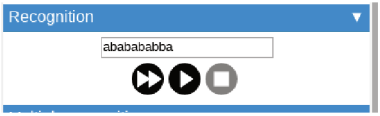

Figura 1. Menu de reconhecimento

O botão central, com um ícone de *play*, representa o reconhecimento passo a passo. Ao clicá-lo pela primeira vez (ou ao pressionar a tecla *enter*), é carregada a configuração inicial da máquina. Cliques sucessivos (ou *enters* sucessivos) então continuam o processo de reconhecimento, um passo de cada vez, atualizando a configuração atual da máquina. A qualquer momento, o usuário pode utilizar o botão *stop* para encerrar o reconhecimento atual. Clicar na caixa de texto automaticamente realiza um *stop*.

4.3.1. Reconhecimento rápido

Assim como no caso de reconhecimento passo a passo, o usuário inicialmente digita a palavra desejada na caixa de texto. Na figura 2, pressiona-se o botão com ícone de *fast forward*, que representa o reconhecimento rápido. Este botão revela o estado final do autômato e quaisquer outras propriedades relevantes ao reconhecimento o mais rápido possível. Assim como no reconhecimento passo a passo, o botão *stop* encerra o reconhecimento, removendo da tela todas as informações produzidas por ele.

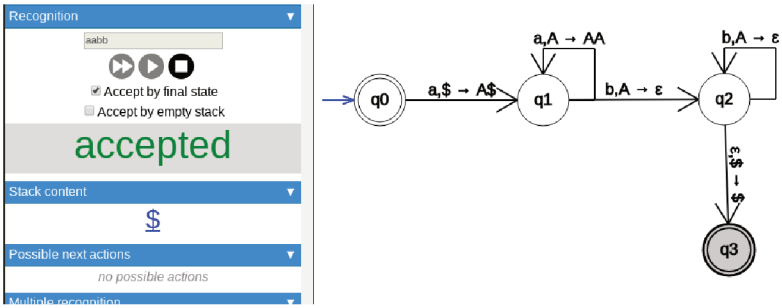

Figura 2. Reconhecimento rápido em um PDA

4.3.2. Reconhecimento múltiplo

O reconhecimento múltiplo encontra-se num menu separado do passo a passo e do rápido, conforme ilustra a figura 3. Para utilizá-lo, digita-se as palavras a serem testadas na área

de texto, separando-as por quebras de linha (fig. 4). O botão com ícone de *play* abaixo da mesma revela o estado do reconhecimento (aceitação, rejeição ou quaisquer outros providos pelo mecanismo) de cada uma.

Figura 3. Reconhecimento múltiplo

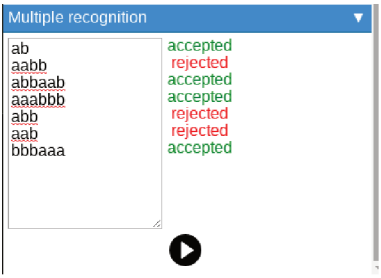

Figura 4. Teste com reconhecimento múltiplo

4.3.3. Alteração de idioma

Na figura 5, mostrada abaixo, é exibido o primeiro menu da barra lateral, que contém as configurações de sistema. A primeira configuração deste menu refere-se ao idioma da aplicação. Ao clicar nesta lista *drop-down*, que por padrão exibe a língua atual, os demais idiomas suportados são exibidos. Quando algum deles é clicado, uma janela de confirmação é exibida, como mostrado na figura 6.

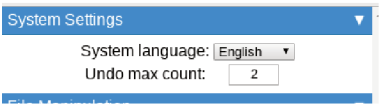

Figura 5. Menu de configurações de sistema

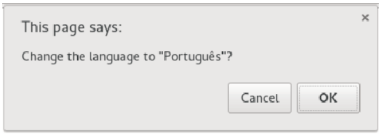

Figura 6. Confirmação de alteração de idioma

5. Conclusão

Neste trabalho, foi desenvolvido um sistema web de caráter didático focado na simulação do reconhecimento de sentenças em autômatos finitos, autômatos de pilha e autômatos linearmente limitados. Ferramentas de propósito semelhante foram analisadas sob critérios como tipos de reconhecimento suportados, a forma pela qual os autômatos são criados, quais os autômatos suportados, entre outros. As funcionalidades do novo sistema foram então expostas na seção 4, incluindo alguns exemplos de uso da aplicação.

Dois métodos de entrada são suportados: via diagrama e via tabela de transições. O suporte a expressões regulares não foi implementado devido a questões de tempo. As três formas de reconhecimento mencionadas na segunda tabela foram implementadas: passo a passo, rápido e múltiplo. Os algoritmos de determinização e minimização também não foram concluídos a tempo.

No caso dos PDAs, todas as características apresentadas na segunda tabela foram concluídas. Para LBAs, optou-se por implementar entrada via tabela ao invés de via código, estabelecendo assim um diferencial sobre todas as ferramentas similares analisadas. A variante escolhida de máquina de Turing possui fita única.

5.1. Trabalhos futuros

Embora as funcionalidades propostas tenham sido todas implementadas, um dos objetivos do sistema desenvolvido é ser extensível e, como tal, há diversas possibilidades de trabalhos futuros.

Primeiramente, novos tipos de mecanismos reconhecedores podem ser adicionados. Os três já suportados foram escolhidos por sua relevância nas disciplinas de Teoria da Computação e Linguagens Formais, mas certamente existem outros que agregariam ainda mais valor à ferramenta. Também pode-se implementar mecanismos já existentes mas com técnicas diferentes de reconhecimento, como um PDA não-determinístico que admita mais de um estado atual ao mesmo tempo ao invés de utilizar *backtracking*, por exemplo.

Outra possibilidade refere-se à adição de novos idiomas. Português e inglês já são suportados; adicionar suporte a outros idiomas aumentaria a acessibilidade da aplicação.

Também pode ser interessante adicionar suporte a outros formatos de arquivo, tanto para abrir como para salvar. Em particular, pode-se permitir uma compatibilidade com outros sistemas de propósito similar, em que o usuário poderia criar um autômato no sistema proposto e abri-lo em outra ferramenta e vice-versa. Assim, funcionalidades não suportadas em uma das aplicações poderiam ser utilizadas na outra sem necessidade de criar o mesmo autômato múltiplas vezes.

Outras extensões possíveis tratam da adição de operações com autômatos, como, por exemplo, determinização, e adicionar um gerador de sentenças aceitas à barra lateral do sistema, o que facilitaria a tarefa de verificar a corretude dos autômatos produzidos pelo usuário. Esse gerador poderia limitar o tamanho das sentenças geradas ou restringir seu formato utilizando, por exemplo, expressões regulares.

Referências

Burch, C. Automaton simulator.

Dickerson, K. Automaton simulator.

Morphett, A. (2014). Turing machine simulator.

Rendell, P. Turing machine simulator.

Rodger, S. H. Jflap.

TypeScript. Typescript - javascript that scales.

Ugarte, M. (2015). Online turing machine simulator.

White, T. M. jfast - a java finite automata simulator.

Zuzak, I. and Jankovic, V. Fsm simulator.

ANEXO B – Código

Neste apêndice encontra-se uma parte do código-fonte da aplicação. O código completo pode ser encontrado em `https://github.com/Ghabriel/AutomatonSimulator/`.

Listing B.1: index.html

```
<html>
<head>
    <title>Automaton Simulator</title>
    <meta http-equiv="Content-type" content="text/html;
    charset=utf-8">
    <link rel="stylesheet" href="css/styles.css">
    <script src="lib/jQuery.js"></script>
    <script src="lib/raphael.js"></script>
    <script src="lib/filesaver.js"></script>
    <script src="out/main.js"></script>
</head>
<body>

    <div id="container">
        <div id="wrapper">
            <div class="row">
                <div id="sidebar"></div>
                <div id="mainbar"></div>
            </div>
        </div>
        <div id="footer">
            Created by Ghabriel Nunes. UFSC, 2017.
        </div>
    </div>

</body>
</html>
```

Listing B.2: scripts/SignalEmitter.ts

```
export interface Signal {
    targetID: string;
    identifier: string;
    data: any;
}

export interface SignalResponse {
    reacted: boolean;
    response: any;
}

export interface SignalObserver {
    receiveSignal: (signal: Signal) => SignalResponse|null;
}

```

```
/**
 * Encapsulates a general-purpose publish-subscribe system.
 * Used mainly to decouple several parts of the application
 * (e.g the mainbar and the sidebar). This class allows, for
 * example, that the sidebar be completely removed and the
 * mainbar continues to work perfectly, even when the latter
 * sends signals to the former.
 */
export class SignalEmitter {
    // Registers a new signal observer, which will be notified
    // when any signal is transmitted.
    static addSignalObserver(observer: SignalObserver): void {
        this.signalObservers.push(observer);
    }

    // Emits a signal to all signal observers, returning
    // the response of the first one that reacts to it.
    // Returns null if there was no reaction.
    static emitSignal(signal: Signal): any {
        for (let observer of this.signalObservers) {
            let response = observer.receiveSignal(signal);
            if (response && response.reacted) {
                return response.response;
            }
        }

        return null;
    }

    private static signalObservers: SignalObserver[] = [];
}
```

Listing B.3: scripts/interface/UIState.ts

```
/// <reference path="../defs/raphael.d.ts" />
/// <reference path="../types.ts" />

import {Browser} from "../Browser"
import {GUI} from "./GUI"
import {Settings} from "../Settings"
import {StatePalette} from "../Palette"
import {utils} from "../Utils"

/**
 * Represents the visual representation of a state.
 */
export class UIState implements State {
    // The position and radius of this state
    public x: number;
```

```
16      public y: number;
17
18      // Is this the initial state?
19      public initial: boolean = false;
20
21      // Is this the final state?
22      public final: boolean = false;
23
24      // Name of this state (which is written in its body)
25      public name: string = "";
26
27      public type: "state" = "state";
28
29      constructor(base?: State) {
30          if (base) {
31              this.x = base.x;
32              this.y = base.y;
33              this.initial = base.initial;
34              this.final = base.final;
35              this.name = base.name;
36          }
37
38          this.radius = Settings.stateRadius;
39      }
40
41      public getPosition(): Point {
42          return {
43              x: this.x,
44              y: this.y
45          };
46      }
47
48      public getRadius(): number {
49          return this.radius;
50      }
51
52      public applyPalette(palette: StatePalette): void {
53          this.palette = palette;
54      }
55
56      public removePalette(): void {
57          this.palette = this.defaultPalette;
58      }
59
60      public remove(): void {
61          if (this.body) {
62              this.body.remove();
63              this.body = null;
64          }
65
66          if (this.ring) {
67              this.ring.remove();
```

```
68              this.ring = null;
69          }
70
71          for (let part of this.arrowParts) {
72              part.remove();
73          }
74          this.arrowParts = [];
75
76          if (this.textContainer) {
77              this.textContainer.remove();
78              this.textContainer = null;
79          }
80      }
81
82      public render(canvas: GUI.Canvas): void {
83          this.renderBody(canvas);
84          this.renderInitialMark(canvas);
85          this.renderFinalMark(canvas);
86          this.renderText(canvas);
87      }
88
89      public node(): GUI.Element|null {
90          return this.body;
91      }
92
93      public html(): SVGElement|null {
94          if (!this.body) {
95              return null;
96          }
97
98          return this.body.node;
99      }
100
101     public drag(moveCallback: (event?: any) => void,
102                 endCallback: (distSquared: number, event:
    any) => boolean): void {
103
104         if (!this.body) {
105             throw Error("Cannot call drag() on a non-
    rendered state");
106         }
107
108         interface MovingEntity {
109             ox: number;
110             oy: number;
111         }
112
113         let self = this;
114
115         let begin = function(this: MovingEntity, x: any, y:
    any, event: any) {
116             let position = self.getPosition();
```

```
117             this.ox = position.x;
118             this.oy = position.y;
119             return {};
120         };
121 
122         // This is used to optimize the dragging process.
    The
123         // "callbackFrequency" variable controls the
    frequency in
124         // which dragging pixels actually trigger the move
    callback.
125         let moveController = 0;
126         let callbackFrequency: number;
127 
128         if (Browser.name == "chrome") {
129             // Chrome is really good at rendering SVG
130             callbackFrequency = 3;
131         } else {
132             callbackFrequency = 4;
133         }
134         let move = function(this: MovingEntity, dx: number,
    dy: number,
135             x: any, y: any, event: any) {
136 
137             self.setVisualPosition(this.ox + dx, this.oy +
    dy);
138             if (moveController == 0) {
139                 moveCallback.call(this, event);
140             }
141 
142             moveController = (moveController + 1) %
    callbackFrequency;
143             return {};
144         };
145 
146         let end = function(this: MovingEntity, event: any) {
147             let position = self.getPosition();
148             let dx = position.x - this.ox;
149             let dy = position.y - this.oy;
150             let distanceSquared = dx * dx + dy * dy;
151             let accepted = endCallback.call(this,
    distanceSquared, event);
152 
153             if (!accepted && (dx != 0 || dy != 0)) {
154                 self.setVisualPosition(this.ox, this.oy);
155             }
156 
157             // Calls the moveCallback here to prevent the
    visual
158             // detachment of edges in low callback frequency
     rates
159             // after the dragging has stopped.
```

```
160             moveCallback.call(this, event);
161             return {};
162         };
163 
164         this.body.drag(move, begin, end);
165         if (this.textContainer) {
166             this.textContainer.drag(move, begin, end);
167         }
168     }
169 
170     private fillColor(): string {
171         return this.palette.fillColor;
172     }
173 
174     private strokeColor(): string {
175         return this.palette.strokeColor;
176     }
177 
178     private strokeWidth(): number {
179         return this.palette.strokeWidth;
180     }
181 
182     private ringStrokeWidth(): number {
183         return this.palette.ringStrokeWidth;
184     }
185 
186     private renderBody(canvas: GUI.Canvas): void {
187         if (!this.body) {
188             this.body = canvas.circle(this.x, this.y, this.
    radius);
189         } else {
190             this.body.attr({
191                 cx: this.x,
192                 cy: this.y
193             });
194         }
195 
196         this.body.attr("fill", this.fillColor());
197         this.body.attr("stroke", this.strokeColor());
198         this.body.attr("stroke-width", this.strokeWidth());
199     }
200 
201     private updateInitialMarkOffsets(): void {
202         if (this.initialMarkOffsets.length) {
203             return;
204         }
205 
206         let length = Settings.stateInitialMarkLength;
207         let x = this.x - this.radius;
208         let y = this.y;
209 
210         // Arrow head
```

```
211             let arrowLength = Settings.
    stateInitialMarkHeadLength;
212             let alpha = Settings.stateInitialMarkAngle;
213             let u = 1 - arrowLength / length;
214             let ref = {
215                 x: x - length + u * length,
216                 y: y
217             };
218
219             // The reference points of the arrow head
220             let target = {x: x, y: y};
221             let p1 = utils.rotatePoint(ref, target, alpha);
222             let p2 = utils.rotatePoint(ref, target, -alpha);
223             this.initialMarkOffsets = [
224                 {
225                     x: p1.x - x,
226                     y: p1.y - y
227                 },
228                 {
229                     x: p2.x - x,
230                     y: p2.y - y
231                 }
232             ];
233         }
234
235         private renderInitialMark(canvas?: GUI.Canvas): void {
236             if (this.initial) {
237                 let length = Settings.stateInitialMarkLength;
238                 let x = this.x - this.radius;
239                 let y = this.y;
240
241                 if (this.arrowParts.length) {
242                     let parts = this.arrowParts;
243                     let body = parts[0];
244                     let topLine = parts[1];
245                     let bottomLine = parts[2];
246
247                     body.attr("path", utils.linePath(x - length,
     y, x, y));
248
249                     this.updateInitialMarkOffsets();
250
251                     let topOffsets = this.initialMarkOffsets[0];
252                     let botOffsets = this.initialMarkOffsets[1];
253
254                     topLine.attr("path", utils.linePath(
    topOffsets.x + x, topOffsets.y + y,
255                                                         x, y));
256                     bottomLine.attr("path", utils.linePath(
    botOffsets.x + x, botOffsets.y + y,
257                                                            x, y)
    );
```

```
258             } else {
259                 if (!canvas) {
260                     // shouldn't happen, just for type
    safety
261                     throw Error();
262                 }
263 
264                 let strokeColor = Settings.
    stateInitialMarkColor;
265                 let strokeWidth = Settings.
    stateInitialMarkThickness;
266 
267                 let body = utils.line(canvas, x - length, y,
     x, y);
268                 body.attr("stroke", strokeColor);
269                 body.attr("stroke-width", strokeWidth);
270 
271                 this.updateInitialMarkOffsets();
272 
273                 let topOffsets = this.initialMarkOffsets[0];
274                 let botOffsets = this.initialMarkOffsets[1];
275 
276                 let topLine = utils.line(canvas, topOffsets.
    x + x, topOffsets.y + y,
277                                                         x, y);
278                 topLine.attr("stroke", strokeColor);
279                 topLine.attr("stroke-width", strokeWidth);
280 
281                 let bottomLine = utils.line(canvas,
    botOffsets.x + x, botOffsets.y + y,
282                                                           x, y);
283                 bottomLine.attr("stroke", strokeColor);
284                 bottomLine.attr("stroke-width", strokeWidth)
    ;
285 
286                 let parts = this.arrowParts;
287                 parts.push(body);
288                 parts.push(topLine);
289                 parts.push(bottomLine);
290             }
291         } else {
292             let parts = this.arrowParts;
293             while (parts.length) {
294                 parts[parts.length - 1].remove();
295                 parts.pop();
296             }
297             // this.arrow.remove();
298             // this.arrow = null;
299         }
300     }
301 
302     private renderFinalMark(canvas: GUI.Canvas): void {
```

```
303             if (this.final) {
304                 if (!this.ring) {
305                     this.ring = canvas.circle(this.x, this.y,
    Settings.stateRingRadius);
306                 } else {
307                     this.ring.attr({
308                         cx: this.x,
309                         cy: this.y
310                     });
311                 }
312 
313                 this.ring.attr("stroke", this.strokeColor());
314                 this.ring.attr("stroke-width", this.
    ringStrokeWidth());
315             } else if (this.ring) {
316                 this.ring.remove();
317                 this.ring = null;
318             }
319         }
320 
321         private renderText(canvas?: GUI.Canvas): void {
322             if (!this.textContainer) {
323                 this.textContainer = canvas!.text(this.x, this.y
    , this.name);
324                 this.textContainer.attr("font-family", Settings.
    stateLabelFontFamily);
325                 this.textContainer.attr("font-size", Settings.
    stateLabelFontSize);
326                 this.textContainer.attr("stroke", Settings.
    stateLabelFontColor);
327                 this.textContainer.attr("fill", Settings.
    stateLabelFontColor);
328             } else {
329                 this.textContainer.attr("x", this.x);
330                 this.textContainer.attr("y", this.y);
331                 this.textContainer.attr("text", this.name);
332             }
333         }
334 
335         private setVisualPosition(x: number, y: number): void {
336             this.x = x;
337             this.y = y;
338 
339             this.body!.attr({
340                 cx: x,
341                 cy: y
342             });
343 
344             if (this.ring) {
345                 this.ring.attr({
346                     cx: x,
347                     cy: y
```

```
348                 });
349             }
350
351             if (this.initial) {
352                 this.renderInitialMark();
353             }
354
355             this.renderText();
356         }
357
358         private radius: number;
359
360         // Used to calculate the coordinates of the
361         // 'initial state arrow'.
362         private initialMarkOffsets: {x: number, y: number}[] =
            [];
363
364         // The default and current palettes of this state.
365         private defaultPalette: StatePalette = Settings.
            stateDefaultPalette;
366         private palette: StatePalette = this.defaultPalette;
367
368         // The GUI components of this state.
369         private body: GUI.Element|null = null;
370         private ring: GUI.Element|null = null;
371         private arrowParts: GUI.Element[] = [];
372         private textContainer: GUI.Element|null = null;
373     }
```

Listing B.4: scripts/interface/UIEdge.ts

```
1   /// <reference path="../types.ts" />
2
3   import {GUI} from "./GUI"
4   import {UIState} from "./UIState"
5   import {Settings} from "../Settings"
6   import {EdgePalette} from "../Palette"
7   import {utils} from "../Utils"
8
9   enum EdgeType {
10      NORMAL,
11      LOOP,
12      CURVED
13  }
14
15  /**
16   * Represents the visual representation of an edge,
17   * which may contain multiple transitions.
18   */
19  export class PartialUIEdge implements PartialEdge<UIState> {
20      // The state that this edge comes from
```

```
21      public origin?: UIState;
22
23      // The state that this edge points to
24      public target?: UIState;
25
26      // A list of texts written in this edge
27      public textList: string[] = [];
28
29      // A list of data lists used by the controllers to
30      // precisely define this transition
31      public dataList: string[][] = [];
32
33      public type: "edge" = "edge";
34
35      public constructor() {
36          let self = this;
37          this.clickCallback = function(e) {
38              for (let callback of self.clickHandlers) {
39                  callback.call(self);
40              }
41          };
42      }
43
44      public setVirtualTarget(target: Point): void {
45          this.virtualTarget = target;
46      }
47
48      public setCurveFlag(flag: boolean): void {
49          this.forcedRender = this.forcedRender || (this.
        curved != flag);
50          this.curved = flag;
51      }
52
53      public isCurved(): boolean {
54          return this.curved;
55      }
56
57      public removed(): boolean {
58          return this.deleted;
59      }
60
61      public addClickHandler(callback: () => void): void {
62          this.clickHandlers.push(callback);
63          this.rebindClickHandlers();
64      }
65
66      public remove(): void {
67          for (let elem of this.body) {
68              elem.remove();
69          }
70          this.body = [];
71
```

```
72          for (let elem of this.head) {
73              elem.remove();
74          }
75          this.head = [];
76
77          if (this.textContainer) {
78              this.textContainer.remove();
79              this.textContainer = null;
80          }
81
82          this.deleted = true;
83      }
84
85      public applyPalette(palette: EdgePalette): void {
86          this.palette = palette;
87          this.forcedRender = true;
88      }
89
90      public removePalette(): void {
91          this.palette = this.defaultPalette;
92          this.forcedRender = true;
93      }
94
95      public render(canvas: GUI.Canvas): void {
96          let preservedOrigin = this.origin
97                                && utils.samePoint(this.
    prevOriginPosition,
98                                                   this.origin.
    getPosition());
99          let preservedTarget = this.target
100                               && utils.samePoint(this.
    prevTargetPosition,
101                                                  this.target.
    getPosition());
102
103         // Don't re-render this edge if neither the origin
    nor the target
104         // states have moved since we last rendered this
    edge, unless
105         // the forced re-render is active.
106         if (!preservedOrigin || !preservedTarget || this.
    forcedRender) {
107             this.renderBody(canvas);
108             this.renderHead(canvas);
109
110             if (this.origin) {
111                 this.prevOriginPosition = this.origin.
    getPosition();
112             }
113
114             if (this.target) {
```

```
115                 this.prevTargetPosition = this.target.
    getPosition();
116             }
117
118             this.forcedRender = false;
119         }
120
121         for (let elem of this.body) {
122             elem.attr("stroke", this.palette.strokeColor);
123         }
124
125         for (let elem of this.head) {
126             elem.attr("stroke", this.palette.strokeColor);
127         }
128
129         // Only re-renders this edge's text if this edge is
130         // complete (i.e it already has a target state)
131         if (this.target) {
132             this.renderText(canvas);
133         }
134     }
135
136     // Re-binds all click events of this edge.
137     private rebindClickHandlers(): void {
138         for (let elem of this.body) {
139             elem.unclick(this.clickCallback);
140             elem.click(this.clickCallback);
141         }
142     }
143
144     private stateCenterOffsets(dx: number, dy: number):
    Point {
145         let angle = Math.atan2(dy, dx);
146         let sin = Math.sin(angle);
147         let cos = Math.cos(angle);
148         let offsetX = Settings.stateRadius * cos;
149         let offsetY = Settings.stateRadius * sin;
150         return {
151             x: offsetX,
152             y: offsetY
153         };
154     }
155
156     private renderBody(canvas: GUI.Canvas): void {
157         let origin = this.origin!.getPosition();
158         let target: typeof origin;
159         if (!this.target) {
160             if (this.virtualTarget) {
161                 target = {
162                     x: this.virtualTarget.x,
163                     y: this.virtualTarget.y
164                 };
```

```
165                 let dx = target.x - origin.x;
166                 let dy = target.y - origin.y;
167                 // The offsets are necessary to ensure that
    mouse events are
168                 // still correctly fired, since not using
    them makes the edge
169                 // appear directly below the cursor.
170                 target.x = origin.x + dx * 0.98;
171                 target.y = origin.y + dy * 0.98;
172             } else {
173                 target = origin;
174             }
175         } else {
176             target = this.target.getPosition();
177         }
178 
179         let dx = target.x - origin.x;
180         let dy = target.y - origin.y;
181         let radius = Settings.stateRadius;
182         let offsets = this.stateCenterOffsets(dx, dy);
183         // Makes the edge start at the border of the state
    rather than
184         // at its center, unless the virtual target is
    inside the state.
185         // That condition makes it easier to create loops.
186         if (dx * dx + dy * dy > radius * radius) {
187             origin.x += offsets.x;
188             origin.y += offsets.y;
189         }
190 
191         if (this.target) {
192             // Adjusts the edge so that it points to the
    border of the state
193             // rather than its center.
194             target.x -= offsets.x;
195             target.y -= offsets.y;
196         }
197 
198         if (this.origin == this.target) {
199             this.loop(canvas);
200         } else if (this.isCurved()) {
201             this.curve(canvas, origin, target);
202         } else {
203             this.normal(canvas, origin, target);
204         }
205     }
206 
207     // Adjusts the this.body array so that its length and
    its type
208     // is equal to given values. If the current type is
    different
```

```
209         // than the passed type, all event clicks are rebound to
             ensure
210         // that they still work properly with a body of a
            potentially
211         // different size.
212         private adjustBody(canvas: GUI.Canvas, length: number,
            type: EdgeType): void {
213             while (this.body.length > length) {
214                 this.body[this.body.length - 1].remove();
215                 this.body.pop();
216             }
217
218             // Expands the array to the correct length
219             while (this.body.length < length) {
220                 this.body.push(utils.line(canvas, 0, 0, 0, 0));
221             }
222
223             if (type != this.currentEdgeType) {
224                 this.currentEdgeType = type;
225
226                 // Re-binds all click events
227                 this.rebindClickHandlers();
228             }
229         }
230
231         // Renders a loop-style body.
232         private loop(canvas: GUI.Canvas): void {
233             let radius = Settings.stateRadius;
234             let pos = this.origin!.getPosition();
235             this.adjustBody(canvas, 4, EdgeType.LOOP);
236             for (let elem of this.body) {
237                 elem.attr("stroke-width", this.palette.
            arrowThickness);
238             }
239
240             this.body[0].attr("path", utils.linePath(
241                 pos.x + radius, pos.y,
242                 pos.x + 2 * radius, pos.y
243             ));
244             this.body[1].attr("path", utils.linePath(
245                 pos.x + 2 * radius, pos.y,
246                 pos.x + 2 * radius, pos.y - 2 * radius
247             ));
248             this.body[2].attr("path", utils.linePath(
249                 pos.x + 2 * radius, pos.y - 2 * radius,
250                 pos.x, pos.y - 2 * radius
251             ));
252             this.body[3].attr("path", utils.linePath(
253                 pos.x, pos.y - 2 * radius,
254                 pos.x, pos.y - radius
255             ));
256         }
```

```
257
258     // Renders a curved body.
259     private curve(canvas: GUI.Canvas, origin: Point, target:
         Point): void {
260         let dx = target.x - origin.x;
261         let dy = target.y - origin.y;
262
263         let hypot = Math.sqrt(dx * dx + dy * dy);
264
265         // A normalized vector that is perpendicular to the
266         // line joining the origin and the target.
267         let perpVector: Point = {
268             x: dy / hypot,
269             y: -dx / hypot
270         };
271
272         let distance = 30;
273         let offsets = {
274             x: distance * perpVector.x,
275             y: distance * perpVector.y
276         };
277
278         this.adjustBody(canvas, 3, EdgeType.CURVED);
279         for (let elem of this.body) {
280             elem.attr("stroke-width", this.palette.
    arrowThickness);
281         }
282
283         this.body[0].attr("path", utils.linePath(
284             origin.x, origin.y,
285             origin.x + offsets.x + dx * 0.125, origin.y +
    offsets.y + dy * 0.125
286         ));
287
288         this.body[1].attr("path", utils.linePath(
289             origin.x + offsets.x + dx * 0.125, origin.y +
    offsets.y + dy * 0.125,
290             origin.x + offsets.x + dx * 0.875, origin.y +
    offsets.y + dy * 0.875
291         ));
292
293         this.body[2].attr("path", utils.linePath(
294             origin.x + offsets.x + dx * 0.875, origin.y +
    offsets.y + dy * 0.875,
295             target.x, target.y
296         ));
297     }
298
299     // Renders a normal body (i.e a straight line)
300     private normal(canvas: GUI.Canvas, origin: Point, target
    : Point): void {
301         this.adjustBody(canvas, 1, EdgeType.NORMAL);
```

```
302             for (let elem of this.body) {
303                 elem.attr("stroke-width", this.palette.
    arrowThickness);
304             }
305 
306             this.body[0].attr("path", utils.linePath(
307                 origin.x, origin.y,
308                 target.x, target.y
309             ));
310         }
311 
312         private renderHead(canvas: GUI.Canvas): void {
313             if (!this.target) {
314                 // Don't render the head of the arrow if there's
     no target
315                 return;
316             }
317 
318             let origin: Point;
319             let target: Point;
320             let dx: number;
321             let dy: number;
322 
323             if (this.origin == this.target) {
324                 // Loop case
325                 let pos = this.origin.getPosition();
326                 let radius = Settings.stateRadius;
327                 origin = {
328                     x: pos.x,
329                     y: pos.y - 2 * radius
330                 };
331                 target = {
332                     x: pos.x,
333                     y: pos.y - radius
334                 };
335 
336                 dx = 0;
337                 dy = radius;
338             } else if (this.isCurved()) {
339                 let path = this.body[2].attr("path");
340                 origin = {
341                     x: path[0][1],
342                     y: path[0][2],
343                 };
344                 target = {
345                     x: path[1][1],
346                     y: path[1][2]
347                 };
348 
349                 dx = target.x - origin.x;
350                 dy = target.y - origin.y;
351             } else {
```

```
352             // Non-loop case
353             origin = this.origin!.getPosition();
354             target = this.target.getPosition();
355 
356             dx = target.x - origin.x;
357             dy = target.y - origin.y;
358             let offsets = this.stateCenterOffsets(dx, dy);
359             target.x -= offsets.x;
360             target.y -= offsets.y;
361             dx -= offsets.x;
362             dy -= offsets.y;
363         }
364 
365         // Arrow head
366         let arrowLength = this.palette.arrowLength;
367         let alpha = this.palette.arrowAngle;
368         let edgeLength = Math.sqrt(dx * dx + dy * dy);
369         let u = 1 - arrowLength / edgeLength;
370         let ref = {
371             x: origin.x + u * dx,
372             y: origin.y + u * dy
373         };
374 
375 
376         // The reference points of the arrow head
377         let p1 = utils.rotatePoint(ref, target, alpha);
378         let p2 = utils.rotatePoint(ref, target, -alpha);
379 
380         let isHeadEmpty = (this.head.length == 0);
381 
382         if (isHeadEmpty) {
383             this.head.push(utils.line(canvas, 0, 0, 0, 0));
384             this.head.push(utils.line(canvas, 0, 0, 0, 0));
385         }
386 
387         if (this.forcedRender || isHeadEmpty) {
388             // Re-set the stroke-width if there's a forced
    render
389             // because it might have been caused by a change
     of
390             // palette.
391             for (let elem of this.head) {
392                 elem.attr("stroke-width", this.palette.
    arrowThickness);
393             }
394         }
395 
396         this.head[0].attr("path", utils.linePath(
397             p1.x, p1.y,
398             target.x, target.y
399         ));
400 
```

```
401             this.head[1].attr("path", utils.linePath(
402                 p2.x, p2.y,
403                 target.x, target.y
404             ));
405         }
406
407         private preparedText(): string {
408             return this.textList.join("\n");
409         }
410
411         private renderText(canvas: GUI.Canvas): void {
412             // We can assume that there's a target state, since
413             // otherwise we wouldn't be rendering the text.
414             let origin = this.origin!.getPosition();
415             let target = this.target!.getPosition();
416             let x: number;
417             let y: number;
418
419             if (this.origin == this.target) {
420                 // Loop case
421                 let radius = Settings.stateRadius;
422                 x = origin.x + radius;
423                 y = origin.y - 2 * radius;
424             } else if (this.isCurved()) {
425                 // Curved case
426                 let path = this.body[1].attr("path");
427                 let x1 = path[0][1];
428                 let y1 = path[0][2];
429                 let x2 = path[1][1];
430                 let y2 = path[1][2];
431                 x = (x1 + x2) / 2;
432                 y = (y1 + y2) / 2;
433             } else {
434                 // Normal case
435                 x = (origin.x + target.x) / 2;
436                 y = (origin.y + target.y) / 2;
437             }
438
439             if (!this.textContainer) {
440                 this.textContainer = canvas.text(x, y, this.
    preparedText());
441                 this.textContainer.attr("font-family", this.
    palette.textFontFamily);
442                 this.textContainer.attr("font-size", this.
    palette.textFontSize);
443                 this.textContainer.attr("stroke", this.palette.
    textFontColor);
444                 this.textContainer.attr("fill", this.palette.
    textFontColor);
445             } else {
446                 this.textContainer.attr("x", x);
447                 this.textContainer.attr("y", y);
```

```
448             this.textContainer.attr("text", this.
    preparedText());
449             this.textContainer.transform("");
450         }
451
452         let angleRad = Math.atan2(target.y - origin.y,
    target.x - origin.x);
453         let angle = utils.toDegrees(angleRad);
454
455         if (angle < -90 || angle > 90) {
456             angle = (angle + 180) % 360;
457         }
458
459         this.textContainer.rotate(angle);
460
461         y -= this.palette.textFontSize * .6;
462         y -= this.palette.textFontSize * (this.textList.
    length - 1) * .7;
463         this.textContainer.attr("y", y);
464     }
465
466     // The position where the origin state was when we last
    rendered
467     // this edge. Used to optimize rendering when both the
    origin and
468     // the target didn't move since the previous rendering.
469     private prevOriginPosition: Point;
470
471     // The position where the target state was when we last
    rendered
472     // this edge. See prevOriginPosition for more context.
473     private prevTargetPosition: Point;
474
475     // If this edge is not yet completed, it might point to
476     // a position in space rather than a state
477     private virtualTarget: Point|null = null;
478
479     // Is this a curved edge?
480     private curved: boolean = false;
481
482     // Should this edge be re-rendered regardless if its
    position changed?
483     private forcedRender: boolean = false;
484
485     // Was this edge previously removed?
486     private deleted: boolean = false;
487
488     // The default and current palettes of this edge.
489     private defaultPalette: EdgePalette = Settings.
    edgeDefaultPalette;
490     private palette: EdgePalette = this.defaultPalette;
491
```

```
492     // The GUI components of this edge.
493     private body: GUI.Element[] = [];
494     private head: GUI.Element[] = [];
495     private textContainer: GUI.Element|null = null;
496
497     // The click events of this edge.
498     private clickHandlers: (() => void)[] = [];
499
500     private clickCallback: (event: Event) => void;
501
502     private currentEdgeType: EdgeType;
503 }
504
505 export class UIEdge extends PartialUIEdge implements Edge<
        UIState> {
506     public origin: UIState;
507     public target: UIState;
508 }
```

Listing B.5: scripts/Settings.ts

```
1  import * as automata from "./lists/MachineList"
2  import * as controllers from "./lists/ControllerList"
3  import * as lang from "./lists/LanguageList"
4  import * as init from "./lists/InitializerList"
5  import * as operations from "./lists/OperationList"
6
7  // import {Regex} from "./misc/Regex"
8  import {Controller} from "./Controller"
9  import {Initializable, Initializer} from "./Initializer"
10 import {StatePalette, EdgePalette} from "./Palette"
11 import {utils} from "./Utils"
12
13 type Operation = (...args: any[]) => any;
14 type OperationMap = {[name: string]: Operation};
15 type MachineName = keyof typeof operations;
16
17 interface OperationDefinition {
18     name: string;
19     command: Operation;
20 }
21
22 interface MachineTraits {
23     name: string;
24     abbreviatedName: string;
25     sidebar: any[];
26     controller: Controller;
27     initializer: Initializable;
28     operations: OperationMap;
29 }
30
```

```
31 /**
32  * Encapsulates the constants used by the application and
33  * other configurable variables (notably, the current
       language
34  * and the current machine).
35  */
36 export namespace Settings {
37     export const sidebarID = "sidebar";
38     export const mainbarID = "mainbar";
39
40     // May be changed if this application is running locally
41     export let imageFolder = "images/";
42
43     export const sidebarSignalID = "sidebar";
44     export const mainControllerSignalID = "mainController";
45
46     export const disabledButtonClass = "disabled";
47
48     export const canvasShortcutID = "canvas";
49
50     export let undoMaxAmount = 3;
51
52     export const menuSlideInterval = 300;
53     export const promptSlideHideInterval = 100;
54     export const promptSlideShowInterval = 200;
55     export const machineSelectionColumns = 1;
56     export const machineActionColumns = 2;
57
58     export const tapeDisplayedChars = 7; // should be odd
59
60     export const multRecognitionAreaRows = 8;
61     export const multRecognitionAreaCols = 15;
62
63     export const stateRadius = 32;
64     export const stateRingRadius = 27;
65     export const stateDragTolerance = 50;
66     export const stateNameMaxLength = 6;
67
68     export const stateLabelFontFamily = "arial";
69     export const stateLabelFontSize = 20;
70     export const stateLabelFontColor = "black";
71
72     export const stateInitialMarkLength = 40;
73     export const stateInitialMarkHeadLength = 15;
74     export const stateInitialMarkAngle = utils.toRadians(20)
       ;
75     export const stateInitialMarkColor = "blue";
76     export const stateInitialMarkThickness = 2;
77
78     export const stateDefaultPalette: StatePalette = {
79         fillColor: "white",
80         strokeColor: "black",
```

```
81          strokeWidth: 1,
82          ringStrokeWidth: 1
83      };
84
85      export const stateHighlightPalette: StatePalette = {
86          fillColor: "#FFD574",
87          strokeColor: "red",
88          strokeWidth: 3,
89          ringStrokeWidth: 2
90      };
91
92      export const stateRecognitionPalette: StatePalette = {
93          fillColor: "#CCC",
94          strokeColor: "black",
95          strokeWidth: 3,
96          ringStrokeWidth: 2
97      };
98
99      export const edgeDefaultPalette: EdgePalette = {
100         strokeColor: "black",
101         arrowThickness: 2,
102         arrowLength: 30,
103         arrowAngle: utils.toRadians(30),
104         textFontFamily: "arial",
105         textFontSize: 20,
106         textFontColor: "black"
107     };
108
109     export const edgeHighlightPalette: EdgePalette = {
110         strokeColor: "red",
111         arrowThickness: edgeDefaultPalette.arrowThickness,
112         arrowLength: edgeDefaultPalette.arrowLength,
113         arrowAngle: edgeDefaultPalette.arrowAngle,
114         textFontFamily: edgeDefaultPalette.textFontFamily,
115         textFontSize: edgeDefaultPalette.textFontSize,
116         textFontColor: edgeDefaultPalette.textFontColor
117     };
118
119     export const edgeFormalDefinitionHoverPalette:
        EdgePalette = {
120         strokeColor: "blue",
121         arrowThickness: 3,
122         arrowLength: edgeDefaultPalette.arrowLength,
123         arrowAngle: edgeDefaultPalette.arrowAngle,
124         textFontFamily: edgeDefaultPalette.textFontFamily,
125         textFontSize: edgeDefaultPalette.textFontSize,
126         textFontColor: edgeDefaultPalette.textFontColor
127     };
128
129     export const acceptedTestCaseColor = "green";
130     export const rejectedTestCaseColor = "red";
131
```

```
132     export const shortcuts = {
133         // File-related controls
134         save: ["ctrl", "S"],
135         open: ["ctrl", "O"],
136         // Automaton-related controls
137         toggleInitial: ["I"],
138         toggleFinal: ["F"],
139         dimSelection: ["ESC"],
140         deleteEntity: ["DELETE"],
141         clearMachine: ["C"],
142         left: ["LEFT"],
143         right: ["RIGHT"],
144         up: ["UP"],
145         down: ["DOWN"],
146         undo: ["ctrl", "Z"],
147         redo: ["ctrl", "Y"],
148         // Recognition-related controls
149         focusTestCase: ["ctrl", "I"],
150         dimTestCase: ["ENTER"], // must be unitary since it's not bound via bindShortcut
151         fastForward: ["R"], // "R"ecognize (is there a better alternative?)
152         step: ["ENTER"],
153         stop: ["ESC"],
154     };
155 
156     export const languages: {[moduleName: string]: any} = lang;
157 
158     export const Machine = automata.Machine;
159 
160     export let language = lang.english;
161 
162     export type Language = typeof language;
163     type LanguageLabel = keyof typeof language.strings;
164 
165     // The current machine being operated on. Defaults to the first machine
166     // of the Machine enum (unless changed, that means FA)
167     export let currentMachine = 0;
168 
169     export let machines: {[m: number]: MachineTraits} = {};
170 
171     // Helper method to get the current controller
172     export function controller(): Controller {
173         return machines[currentMachine].controller;
174     }
175 
176     // Helper method to get the supported operations
177     // of the current controller
178     export function supportedOperations(): OperationMap {
179         return machines[currentMachine].operations;
```

```
180     }
181
182     let customSettings: [string, Element][] = [];
183     export function addCustomSetting(name: string, element:
        Element): void {
184         customSettings.push([name, element]);
185     }
186
187     export function getCustomSettings(): [string, Element][]
         {
188         return customSettings;
189     }
190
191     let controllerMap: {[m: number]: Controller} = {};
192     let initializerMap: {[m: number]: Initializable} = {};
193     let operationMap: {[m: number]: OperationMap} = {};
194
195     function getController(name: MachineName): Controller {
196         return new (<any> controllers)[name + "Controller"
        ]();
197     }
198
199     function getInitializable(name: MachineName):
        Initializable {
200         return new (<any> init)["init" + name]();
201     }
202
203     function buildOperationMap(map: Map<OperationDefinition
        >): OperationMap {
204         let result: OperationMap = {};
205         utils.foreach(map, function(filename, definition) {
206             result[definition.name] = definition.command;
207         });
208
209         return result;
210     }
211
212     let firstUpdate = true;
213     export function update(): void {
214         if (firstUpdate) {
215             for (let index in Machine) {
216                 if (Machine.hasOwnProperty(index) && !isNaN(
        parseInt(index))) {
217                     // controllerMap[index] = new
        controllers[Machine[index] + "Controller"]();
218                     // initializerMap[index] = new init["
        init" + Machine[index]]();
219                     let machineName = <MachineName> Machine[
        index];
220                     controllerMap[index] = getController(
        machineName);
```

```
221                         initializerMap[index] = getInitializable
    (machineName);
222                         operationMap[index] = buildOperationMap(
    operations[machineName]);
223                     }
224                 }
225             }
226
227             let machineList: typeof machines = {};
228             for (let index in Machine) {
229                 if (Machine.hasOwnProperty(index) && !isNaN(
    parseInt(index))) {
230                     // Stores the traits of this machine. Note
    that the
231                     // "sidebar" property is filled by the init*
     classes.
232                     machineList[index] = {
233                         name: language.strings[<LanguageLabel>
    Machine[index]],
234                         abbreviatedName: Machine[index],
235                         sidebar: [],
236                         controller: controllerMap[index],
237                         initializer: initializerMap[index],
238                         operations: operationMap[index]
239                     };
240                 }
241             }
242
243             utils.foreach(machineList, function(key, value) {
244                 machines[parseInt(key)] = value;
245             });
246
247             Initializer.exec(initializerMap);
248
249             if (firstUpdate) {
250                 machines[currentMachine].initializer.onEnter();
251             }
252
253             firstUpdate = false;
254         }
255
256         export function changeLanguage(newLanguage: Language):
    void {
257             customSettings = [];
258             language = newLanguage;
259             Strings = language.strings;
260             update();
261         }
262
263         export function changeMachine(machineIndex: number):
    void {
264             machines[currentMachine].initializer.onExit();
```

```
265             currentMachine = machineIndex;
266             machines[currentMachine].initializer.onEnter();
267         }
268
269         // This can be customized to allow other kinds of
            resources.
270         // Currently, every resource is an image.
271         export function getResourcePath(name: string): string {
272             return imageFolder + name;
273         }
274     }
275
276     export let Strings = Settings.language.strings;
277
278     // Settings.update();
279     // Initializer.exec();
```

Listing B.6: scripts/MainController.ts

```
1   /// <reference path="types.ts" />
2
3   import {AutomatonRenderer} from "./interface/
        AutomatonRenderer"
4   import {EdgeUtils} from "./EdgeUtils"
5   import {Memento} from "./Memento"
6   import {PersistenceHandler} from "./persistence/
        PersistenceHandler"
7   import {Settings, Strings} from "./Settings"
8   import {Signal, SignalEmitter, SignalResponse} from "./
        SignalEmitter"
9   import {System} from "./System"
10  import {utils} from "./Utils"
11
12  /**
13   * Controls the main area of the application. Interacts with
         a renderer
14   * (which is an AutomatonRenderer), handles persistence,
        enables undo/redo
15   * of actions and interacts with a Controller.
16   */
17  export class MainController {
18      constructor(renderer: AutomatonRenderer, memento:
        Memento<string>,
19                  persistenceHandler: PersistenceHandler) {
20
21          this.renderer = renderer;
22          this.memento = memento;
23          this.persistenceHandler = persistenceHandler;
24
25          renderer.setController(this);
26          SignalEmitter.addSignalObserver(this);
```

```
27
28          System.addLanguageChangeObserver({
29              onLanguageChange: () => {
30                  renderer.onLanguageChange();
31              }
32          });
33
34          System.addMachineChangeObserver({
35              onMachineChange: () => {
36                  renderer.onMachineChange();
37              }
38          });
39      }
40
41      public receiveSignal(signal: Signal): SignalResponse|
        null {
42          if (signal.targetID == Settings.
        mainControllerSignalID) {
43              let methodName = <keyof this> signal.identifier;
44              let method = <Function> <any> this[methodName];
45
46              return {
47                  reacted: true,
48                  response: method.apply(this, signal.data)
49              };
50          }
51
52          return null;
53      }
54
55      public clear(): void {
56          this.stateList = {};
57          this.edgeList = {};
58          this.initialState = null;
59
60          this.renderer.clear();
61          Settings.controller().clear();
62      }
63
64      public empty(): boolean {
65          // Doesn't need to check for edgeList.length since
        edges
66          // can't exist without states.
67          return Object.keys(this.stateList).length == 0;
68      }
69
70      public save(): string {
71          return this.persistenceHandler.save(this.stateList,
72                      this.edgeList, this.initialState);
73      }
74
75      public load(content: string): void {
```

```
76          this.internalLoad(content);
77          this.pushState();
78      }
79
80      public undo(): void {
81          let data = this.memento.undo();
82          if (data) {
83              this.clearAndLoad(data);
84          }
85      }
86
87      public redo(): void {
88          let data = this.memento.redo();
89          if (data) {
90              this.clearAndLoad(data);
91          }
92      }
93
94      // ---------------------- Forwarders ----------------------
95      public recognitionHighlight(states: string[]): void {
96          this.renderer.recognitionHighlight(states);
97      }
98
99      public recognitionDim(): void {
100         this.renderer.recognitionDim();
101     }
102
103     public lock(): void {
104         this.renderer.lock();
105     }
106
107     public unlock(): void {
108         this.renderer.unlock();
109     }
110
111     public stateManualCreation(): void {
112         this.renderer.stateManualCreation();
113     }
114
115     public edgeManualCreation(): void {
116         this.renderer.edgeManualCreation();
117     }
118
119     // ------------------------ Creation ------------------------
120     public createState(externalState: State): void {
121         let state = this.cleanup(externalState);
122
123         if (this.empty()) {
124             // The first state should be initial
125             state.initial = true;
126             this.initialState = state;
127         }
```

```
128
129             this.stateList[state.name] = state;
130             this.renderer.createState(state);
131             Settings.controller().createState(state);
132         }
133
134         public createEdge<T extends State>(externalEdge: Edge<T
        >): void {
135             this.createMergedTransition(externalEdge);
136             this.renderer.createEdge(externalEdge);
137         }
138
139         // Creates a transition both internally and remotely,
140         // but does not render it
141         public createMergedTransition<T extends State>(
        externalEdge: Edge<T>): void {
142             let edge = this.cleanup(externalEdge);
143             this.internalCreateEdge(edge);
144
145             let {origin, target} = edge;
146             for (let dataList of edge.dataList) {
147                 this.remoteCreateTransition(origin, target,
        dataList);
148             }
149         }
150
151         public remoteCreateTransition(origin: State, target:
        State, data: string[]): void {
152             Settings.controller().createTransition(origin,
        target, data);
153         }
154
155         // ---------------------- Edition: states
        -----------------------
156         public renameState(externalState: State, newName: string
        ): boolean {
157             if (this.stateExists(newName)) {
158                 return false;
159             }
160
161             let state = this.internal(externalState);
162
163             delete this.stateList[state.name];
164             utils.rerouteEdges(this.edgeList, state.name,
        newName);
165             this.stateList[newName] = state;
166
167             state.name = newName;
168
169             // Must use the external state here, since the
        internal one
```

```
170         // has already been updated (which would cause the
    renderer
171         // to not find it)
172         Settings.controller().renameState(externalState,
    newName);
173         this.renderer.renameState(externalState, newName);
174         return true;
175     }
176
177     public toggleInitialFlag(externalState: State): void {
178         let state = this.internal(externalState);
179
180         if (state == this.initialState) {
181             state.initial = false;
182             this.initialState = null;
183         } else {
184             if (this.initialState) {
185                 this.initialState.initial = false;
186                 Settings.controller().changeInitialFlag(this
    .initialState);
187             }
188
189             state.initial = true;
190             this.initialState = state;
191         }
192
193         this.renderer.toggleInitialFlag(state);
194         Settings.controller().changeInitialFlag(state);
195     }
196
197     public toggleFinalFlag(externalState: State): void {
198         let state = this.internal(externalState);
199         state.final = !state.final;
200         this.renderer.toggleFinalFlag(state);
201         Settings.controller().changeFinalFlag(state);
202     }
203
204     // ------------------- Edition: edges/transitions
    ---------------------
205     public changeTransitionOrigin<T extends State>(
    externalEdge: Edge<T>,
206         newOrigin: State): void {
207
208         let edge = this.internal(externalEdge);
209
210         this.remoteDeleteEdge(edge);
211         this.internalDeleteEdge(edge);
212         edge.origin = this.internal(newOrigin);
213         this.internalCreateEdge(edge);
214
215         // Must use the external edge here, since the
    internal one
```

```
216             // has already been updated (which would cause the
        renderer
217             // to not find it)
218             this.renderer.changeTransitionOrigin(externalEdge,
        newOrigin);
219
220             this.rebuildEdge(edge);
221         }
222
223         public changeTransitionTarget<T extends State>(
        externalEdge: Edge<T>,
224             newTarget: State): void {
225
226             let edge = this.internal(externalEdge);
227
228             this.remoteDeleteEdge(edge);
229             this.internalDeleteEdge(edge);
230             edge.target = this.internal(newTarget);
231             this.internalCreateEdge(edge);
232
233             // Must use the external edge here, since the
        internal one
234             // has already been updated (which would cause the
        renderer
235             // to not find it)
236             this.renderer.changeTransitionTarget(externalEdge,
        newTarget);
237
238             this.rebuildEdge(edge);
239         }
240
241         public changeTransitionData<T extends State>(
        externalEdge: Edge<T>,
242             transitionIndex: number, newData: string[], newText:
         string): void {
243
244             let edge = this.internal(externalEdge);
245             let {origin, target, dataList, textList} = edge;
246
247             let controller = Settings.controller();
248             controller.deleteTransition(origin, target, dataList
        [transitionIndex]);
249
250             dataList[transitionIndex] = utils.clone(newData);
251             textList[transitionIndex] = newText;
252             controller.createTransition(origin, target, newData)
        ;
253
254             this.renderer.changeTransitionData(edge,
        transitionIndex,
255                 newData, newText);
256         }
```

```
257
258     // ---------------------- Deletion -----------------------
259     public deleteState(externalState: State): void {
260         if (!this.stateExists(externalState.name)) {
261             return;
262         }
263
264         let state = this.internal(externalState);
265
266         this.removeEdgesOfState(state);
267
268         delete this.stateList[state.name];
269
270         this.renderer.deleteState(state);
271         Settings.controller().deleteState(state);
272     }
273
274     public deleteTransition<T extends State>(externalEdge:
    Edge<T>,
275         transitionIndex: number): void {
276
277         let edge = this.internal(externalEdge);
278         let {origin, target, dataList, textList} = edge;
279
280         let controller = Settings.controller();
281         controller.deleteTransition(origin, target, dataList
    [transitionIndex]);
282
283         dataList.splice(transitionIndex, 1);
284         textList.splice(transitionIndex, 1);
285
286         if (dataList.length == 0) {
287             this.deleteEdge(edge);
288         } else {
289             this.renderer.deleteTransition(edge,
    transitionIndex);
290         }
291     }
292
293     public deleteEdge<T extends State>(externalEdge: Edge<T
    >): void {
294         let edge = this.internal(externalEdge);
295
296         this.internalDeleteEdge(edge);
297         this.renderer.deleteEdge(edge);
298
299         let {origin, target, dataList} = edge;
300         let controller = Settings.controller();
301
302         for (let data of dataList) {
303             controller.deleteTransition(origin, target, data
    );
```

```
304             }
305         }
306
307         public internalDeleteEdge<T extends State>(edge: Edge<T
            >): void {
308             if (!this.edgeList.hasOwnProperty(edge.origin.name))
             {
309                 return;
310             }
311
312             delete this.edgeList[edge.origin.name][edge.target.
            name];
313         }
314
315         // ---------------------- Event listeners
            ------------------------
316         /**
317          * Returns a function that is called when the formal
            definition
318          * of the current machine changes (i.e. when it's edited
            ). If it
319          * returns false, the renderer ignores this event.
320          * @return {Generator<boolean>} the listener function
321          */
322         public getFormalDefinitionCallback(): Generator<boolean>
             {
323             return () => {
324                 if (this.loadingMode) {
325                     return false;
326                 }
327
328                 if (!this.frozenMemento) {
329                     this.pushState();
330                 }
331
332                 return true;
333             };
334         }
335
336         /**
337          * Called after a state stops being dragged.
338          */
339         public onStateDrag(externalState: State): void {
340             let state = this.internal(externalState);
341
342             state.x = externalState.x;
343             state.y = externalState.y;
344
345             // Saves the post-drag state to the memento
346             // to allow the user to undo it
347             this.pushState();
348         }
```

```
349
350
351
352     private removeEdgesOfState(state: State): void {
353         EdgeUtils.edgeIteration(this.edgeList, (edge) => {
354             let {origin, target} = edge;
355
356             if (origin == state || target == state) {
357                 this.internalDeleteEdge(edge);
358             }
359         });
360     }
361
362     private internalCreateEdge<T extends State>(edge: Edge<T
    >): void {
363         let {origin, target} = edge;
364
365         if (!this.edgeList.hasOwnProperty(origin.name)) {
366             this.edgeList[origin.name] = {};
367         }
368
369         if (!this.edgeList[origin.name].hasOwnProperty(
    target.name)) {
370             this.edgeList[origin.name][target.name] = edge;
371             return;
372         }
373
374         let parallelEdge = this.edgeList[origin.name][target
    .name];
375         parallelEdge.dataList.push(...edge.dataList);
376         parallelEdge.textList.push(...edge.textList);
377     }
378
379     private remoteDeleteEdge<T extends State>(edge: Edge<T>)
    : void {
380         let controller = Settings.controller();
381
382         for (let data of edge.dataList) {
383             controller.deleteTransition(edge.origin, edge.
    target, data);
384         }
385     }
386
387     private rebuildEdge<T extends State>(edge: Edge<T>):
    void {
388         for (let data of edge.dataList) {
389             this.remoteCreateTransition(edge.origin, edge.
    target, data);
390         }
391     }
392
393     private pushState(): void {
```

```
394             this.memento.push(this.save());
395         }
396
397         private clearAndLoad(data: string): void {
398             this.frozenMemento = true;
399             this.clear();
400             this.frozenMemento = false;
401
402             this.internalLoad(data);
403         }
404
405         private stateExists(name: string): boolean {
406             return this.stateList.hasOwnProperty(name);
407         }
408
409         private internalLoad(content: string): void {
410             // Blocks formal definition change events
411             this.loadingMode = true;
412
413             // Blocks changes to the memento until the load
        process is complete
414             this.frozenMemento = true;
415
416             Settings.controller().clear();
417
418             let loadedData = this.persistenceHandler.load(
        content);
419             if (loadedData.error) {
420                 alert(Strings.INVALID_FILE);
421                 this.loadingMode = false;
422                 this.frozenMemento = false;
423                 return;
424             }
425
426             if (loadedData.aborted) {
427                 this.loadingMode = false;
428                 this.frozenMemento = false;
429                 return;
430             }
431
432             this.stateList = loadedData.stateList;
433             this.edgeList = loadedData.edgeList;
434
435             this.initialState = loadedData.initialState;
436
437             // We shouldn't render states and edges during
        creation because,
438             // if the automaton is big enough and there's an
        error in the
439             // source file, the user would see states and edges
        appearing
```

```
440             // and then vanishing, then an error message.
    Rendering everything
441             // after processing makes it so that nothing appears
     (except the
442             // error message) if there's an error.
443             this.renderer.setStateList(this.stateList);
444             this.renderer.setEdgeList(this.edgeList);
445 
446             this.loadingMode = false;
447             this.renderer.triggerFormalDefinitionChange();
448             this.frozenMemento = false;
449         }
450 
451         private internal(state: State): State;
452         private internal<T extends State>(edge: Edge<T>): Edge<
        State>;
453         private internal<T extends State>(entity: State|Edge<T>)
        : State|Edge<State>;
454         private internal(entity: any): any {
455             if (entity.type == "state") {
456                 return this.stateList[entity.name];
457             } else {
458                 return this.edgeList[entity.origin.name][entity.
        target.name];
459             }
460         }
461 
462         // Removes all unnecessary properties
463         private cleanup(state: State): State;
464         private cleanup<T extends State>(state: Edge<T>): Edge<T
        >;
465         private cleanup<T extends State>(entity: State|Edge<T>):
         any {
466             if (entity.type == "state") {
467                 return {
468                     x: entity.x,
469                     y: entity.y,
470                     initial: entity.initial,
471                     final: entity.final,
472                     name: entity.name,
473                     type: entity.type
474                 };
475             } else {
476                 return {
477                     origin: this.internal(entity.origin),
478                     target: this.internal(entity.target),
479                     textList: utils.clone(entity.textList),
480                     dataList: utils.clone(entity.dataList),
481                     type: entity.type
482                 };
483             }
484         }
```

```
485
486     private memento: Memento<string>;
487     private persistenceHandler: PersistenceHandler;
488     private renderer: AutomatonRenderer;
489
490     // Internal automaton structures
491     private stateList: Map<State> = {};
492     private edgeList: IndexedEdgeGroup<Edge<State>> = {};
493
494     private initialState: State|null = null;
495
496     private frozenMemento: boolean = false;
497     private loadingMode: boolean = false;
498 }
```

Listing B.7: scripts/Controller.ts

```
1  /// <reference path="types.ts" />
2
3  import {Prompt} from "./Prompt"
4
5  export interface FormalDefinition {
6      // Order of the parameters displayed in M = (...)
7      tupleSequence: string[]; // e.g [Q, sigma, delta, q0, F]
8
9      // Order of the parameters displayed below M = (...)
10     parameterSequence: string[];
11
12     // Values of each parameter
13     parameterValues: {[p: string]: any};
14 }
15
16 export interface TransitionTable {
17     domain: string;
18     codomain: string;
19     header: string[];
20     list: string[][];
21     metadata: [string, string, string[]][];
22 }
23
24 export type Operation = (...args: any[]) => any;
25
26 /**
27  * Generic interface that specifies the mandatory methods of
        a controller.
28  */
29 export interface Controller {
30     // Interface-related edge manipulation
31     edgePrompt(callback: (data: string[], text: string) =>
       void,
32                fallback?: () => void): Prompt;
```

```
33
34      edgeDataToText(data: string[]): string;
35
36      // Edition-related methods
37      createState(state: State): void;
38      createTransition(origin: State, target: State, data:
        string[]): void;
39      changeInitialFlag(state: State): void;
40      changeFinalFlag(state: State): void;
41      renameState(state: State, newName: string): void;
42      deleteState(state: State): void;
43      deleteTransition(origin: State, target: State, data:
        string[]): void;
44      clear(): void;
45
46      // Recognition-related methods
47      fastForward(input: string): void;
48      step(input: string): void;
49      stop(): void;
50      reset(): void;
51      finished(input: string): boolean;
52      isStopped(): boolean;
53      stepPosition(): number;
54
55      // Editing-related callback methods (useful for updating
         formal
56      // definitions in the interface)
57      // Should be called whenever an editing-related method
        is called.
58      setEditingCallback(callback: () => void): void;
59
60      // Useful getters
61      currentStates(): string[];
62      accepts(): boolean;
63      formalDefinition(): FormalDefinition;
64
65      // Support to machine operations
66      applyOperation(operation: Operation): void;
67  }
```

Listing B.8: scripts/machines/PDA/PDA.ts

```
1  import {UnorderedSet} from "../../datastructures/
       UnorderedSet"
2  import {utils} from "../../Utils"
3
4  type State = string;
5  type Index = number;
6  type Alphabet = {[i: string]: number};
7  type GammaClosure = string;
```

```
type InternalTransitionInformation = [Index, GammaClosure][];
export type TransitionInformation = [State, GammaClosure];

type SymbolLocation = "inputAlphabet" | "stackAlphabet";

interface BaseAction {
    stepIndex: number;
    currentInput: string;
    currentStack: string[];
    inputRead: string;
    stackWrite: string;
}

interface Action extends BaseAction {
    targetState: Index;
}

export interface ActionInformation extends BaseAction {
    targetState: State;
}

export enum AcceptingHeuristic {
    NEVER = 0,
    ACCEPTING_STATE = 1,
    EMPTY_STACK = 2,
    BOTH = ACCEPTING_STATE | EMPTY_STACK
}

const EPSILON_KEY = "";

export class PDA {
    public setAcceptingHeuristic(heuristic: AcceptingHeuristic): void {
        this.acceptingHeuristic = heuristic;
    }

    public getAcceptingHeuristic(): AcceptingHeuristic {
        return this.acceptingHeuristic;
    }

    // Adds a state to this PDA, marking it as the initial state
    // if there are no other states in this PDA.
    public addState(name: State): Index {
        this.stateList.push(name);
        // Cannot use this.numStates() here because
        // removed states are kept in the list
        let index = this.realNumStates() - 1;
        this.transitions[index] = {};
        if (this.initialState == -1) {
            this.initialState = index;
```

```
57              this.reset();
58          }
59          return index;
60      }
61
62      // Removes a state from this PDA.
63      public removeState(index: Index): void {
64          this.removeEdgesOfState(index);
65
66          if (this.initialState == index) {
67              this.unsetInitialState();
68          }
69
70          this.finalStates.erase(index);
71
72          this.stateList[index] = undefined;
73          delete this.transitions[index];
74          this.numRemovedStates++;
75      }
76
77      // Renames a state of this PDA.
78      public renameState(index: Index, newName: State): void {
79          this.stateList[index] = newName;
80      }
81
82      // Adds a transition to this PDA.
83      public addTransition(source: Index, target: Index, data:
         string[]): void {
84          let transitions = this.transitions[source];
85          let input = data[0];
86          let stackRead = data[1];
87          let stackWrite = data[2].split("").reverse().join("")
        );
88
89          if (!transitions.hasOwnProperty(input)) {
90              transitions[input] = {};
91          }
92
93          if (!transitions[input].hasOwnProperty(stackRead)) {
94              transitions[input][stackRead] = [];
95          }
96
97          transitions[input][stackRead].push([target,
        stackWrite]);
98
99          this.addInputSymbol(input);
100         this.addStackSymbol(stackRead);
101
102         for (let i = 0; i < stackWrite.length; i++) {
103             this.addStackSymbol(stackWrite[i]);
104         }
105     }
```

```
106
107     // Removes a transition from this PDA.
108     public removeTransition(source: Index, target: Index,
        data: string[]): void {
109         let transitions = this.transitions[source];
110         let input = data[0];
111         let stackRead = data[1];
112         let stackWrite = data[2].split("").reverse().join("")
        );
113
114         if (!transitions.hasOwnProperty(input)) {
115             return;
116         }
117
118         if (!transitions[input].hasOwnProperty(stackRead)) {
119             return;
120         }
121
122         let properties = transitions[input][stackRead];
123         for (let i = 0; i < properties.length; i++) {
124             let group = properties[i];
125             if (group[0] == target && group[1] == stackWrite
        ) {
126                 // properties.splice(i, 1);
127                 this.uncheckedRemoveTransition(source, input
        , stackRead, i);
128                 break;
129             }
130         }
131     }
132
133     // Sets the initial state of this PDA.
134     public setInitialState(index: Index): void {
135         if (index < this.realNumStates()) {
136             this.initialState = index;
137         }
138     }
139
140     // Unsets the initial state of this PDA.
141     public unsetInitialState(): void {
142         this.initialState = -1;
143     }
144
145     // Returns the name of the initial state.
146     public getInitialState(): State|undefined {
147         return this.stateList[this.initialState];
148     }
149
150     // Marks a state as final.
151     public addAcceptingState(index: Index): void {
152         this.finalStates.insert(index);
153     }
```

```
154
155     // Marks a state as non-final.
156     public removeAcceptingState(index: Index): void {
157         this.finalStates.erase(index);
158     }
159
160     // Returns all accepting states
161     public getAcceptingStates(): State[] {
162         let result: State[] = [];
163         let self = this;
164         this.finalStates.forEach(function(index) {
165             result.push(self.stateList[index]!);
166         });
167         return result;
168     }
169
170     // Returns the current state of this PDA.
171     public getCurrentState(): State|undefined {
172         if (this.currentState === null) {
173             return undefined;
174         }
175
176         return this.stateList[this.currentState];
177     }
178
179     // Returns a list containing all the states of this PDA.
180     public getStates(): State[] {
181         return (<State[]> this.stateList).filter(function(
    value) {
182             return value !== undefined;
183         });
184     }
185
186     // Executes a callback function for every transition of
        this PDA.
187     public transitionIteration(callback: (source: State,
188         data: TransitionInformation, input: string,
189         stackRead: string) => void): void {
190
191         let self = this;
192         utils.foreach(this.transitions, function(index,
    stateTransitions) {
193             utils.foreach(stateTransitions, function(input,
    indexedByStack) {
194                 utils.foreach(indexedByStack, function(
    stackRead, info) {
195                     let sourceState = self.stateList[
    parseInt(index)]!;
196                     for (let group of info) {
197                         let targetState = self.stateList[
    group[0]]!;
```

```
198                         let stackWrite = group[1].split("").
    reverse().join("");
199                         callback(sourceState, [targetState,
    stackWrite], input, stackRead);
200                     }
201                 });
202             });
203         });
204     }
205
206     // Returns the input alphabet of this PDA.
207     public getInputAlphabet(): string[] {
208         let result: string[] = [];
209         for (let member in this.inputAlphabet) {
210             if (this.inputAlphabet.hasOwnProperty(member)) {
211                 result.push(member);
212             }
213         }
214         return result;
215     }
216
217     // Returns the stack alphabet of this PDA.
218     public getStackAlphabet(): string[] {
219         let result: string[] = [];
220         for (let member in this.stackAlphabet) {
221             if (this.stackAlphabet.hasOwnProperty(member)) {
222                 result.push(member);
223             }
224         }
225         return result;
226     }
227
228     public getStackContent(): string[] {
229         return this.stack;
230     }
231
232     public setInput(input: string): void {
233         this.input = input;
234         this.stack = ["$"];
235         this.stepIndex = 0;
236         this.actionTree = this.getPossibleActions();
237         this.halt = false;
238     }
239
240     public getCurrentInput(): string {
241         return this.input;
242     }
243
244     public getActionTree(): ActionInformation[] {
245         let result: ActionInformation[] = [];
246         for (let action of this.actionTree) {
247             result.push({
```

```
248                 stepIndex: action.stepIndex,
249                 currentInput: action.currentInput,
250                 currentStack: action.currentStack,
251                 inputRead: action.inputRead,
252                 stackWrite: action.stackWrite,
253                 targetState: this.stateList[action.
    targetState]!
254             });
255         }
256
257         return result;
258     }
259
260     // Reads a character from the input, triggering state
    changes to this PDA.
261     public read(): void {
262         if (this.error()) {
263             this.halt = true;
264             return;
265         }
266
267         let actionTree = this.actionTree;
268
269         // makes it possible for PDAs to accept empty
    strings
270         if (this.input.length == 0 && this.accepts()) {
271             this.halt = true;
272             return;
273         }
274
275         if (actionTree.length == 0) {
276             // goes to the error state
277             this.currentState = null;
278             this.input = "";
279             this.halt = true;
280             return;
281         }
282
283         let nextAction = actionTree[actionTree.length - 1];
284
285         // if (nextAction.stepIndex <= this.stepIndex) {
286         //  console.log("[BACKTRACK]");
287         // }
288
289         this.processAction(nextAction);
290         this.actionTree.pop();
291         this.halt = false;
292
293         let possibleActions = this.getPossibleActions();
294         this.actionTree.push(...possibleActions);
295
```

```
296             let imminentBacktracking = (possibleActions.length
    == 0);
297             if (imminentBacktracking && this.input.length == 0
    && this.accepts()) {
298                 // prevent backtracking on the next call to read
    ()
299                 // since we found an accepting branch
300                 this.halt = true;
301             }
302         }
303
304         public halted(): boolean {
305             return this.halt;
306         }
307
308         // Resets this PDA, making it return to its initial
    state and
309         // clearing its stack.
310         public reset(): void {
311             if (this.initialState == -1) {
312                 this.currentState = null;
313             } else {
314                 this.currentState = this.initialState;
315             }
316
317             this.stack = [];
318             this.stepIndex = 0;
319             this.actionTree = [];
320             this.halt = true;
321         }
322
323         // Clears this PDA, making it effectively equal to new
    PDA().
324         public clear(): void {
325             this.stateList = [];
326             this.inputAlphabet = {};
327             this.stackAlphabet = {};
328             this.transitions = {};
329             this.unsetInitialState();
330             this.finalStates.clear();
331             this.numRemovedStates = 0;
332             this.currentState = null;
333             this.stack = [];
334             this.stepIndex = 0;
335             this.actionTree = [];
336             this.halt = true;
337         }
338
339         // Checks if this PDA accepts in its current state.
340         public accepts(): boolean {
341             if (this.currentState === null) {
342                 return false;
```

```
343         }
344
345         let result: boolean = false;
346         if (this.acceptingHeuristic & AcceptingHeuristic.
    ACCEPTING_STATE) {
347             result = result || this.finalStates.contains(
    this.currentState);
348         }
349
350         if (this.acceptingHeuristic & AcceptingHeuristic.
    EMPTY_STACK) {
351             result = result || (this.stack.length == 0);
352         }
353
354         return result;
355     }
356
357     public acceptedHeuristic(): AcceptingHeuristic|null {
358         if (this.currentState === null) {
359             return null;
360         }
361
362         let result: AcceptingHeuristic = AcceptingHeuristic.
    NEVER;
363         if (this.acceptingHeuristic & AcceptingHeuristic.
    ACCEPTING_STATE) {
364             if (this.finalStates.contains(this.currentState)
    ) {
365                 result |= AcceptingHeuristic.ACCEPTING_STATE
    ;
366             }
367         }
368
369         if (this.acceptingHeuristic & AcceptingHeuristic.
    EMPTY_STACK) {
370             if (this.stack.length == 0) {
371                 result |= AcceptingHeuristic.EMPTY_STACK;
372             }
373         }
374
375         return result == AcceptingHeuristic.NEVER ? null :
    result;
376     }
377
378     // Checks if this PDA is in an error state, i.e. isn't
    in any state.
379     public error(): boolean {
380         return this.currentState === null;
381     }
382
383     // Returns the number of states of this PDA.
384     public numStates(): number {
```

```
385         return this.stateList.length - this.numRemovedStates
        ;
386     }
387
388     private uncheckedRemoveTransition(source: Index, input:
        string,
389         stackRead: string, index: number): void {
390
391         let groups = this.transitions[source][input][
        stackRead];
392         let stackWrite = groups[index][1];
393
394         this.removeInputSymbol(input);
395         this.removeStackSymbol(stackRead);
396
397         for (let i = 0; i < stackWrite.length; i++) {
398             this.removeStackSymbol(stackWrite[i]);
399         }
400
401         groups.splice(index, 1);
402     }
403
404     private removeEdgesOfState(index: Index): void {
405         utils.foreach(this.transitions, (originIndex,
        transitions) => {
406             let origin = parseInt(originIndex);
407
408             utils.foreach(transitions, (input,
        indexedByStack) => {
409                 utils.foreach(indexedByStack, (stackRead,
        group) => {
410                     let i = 0;
411                     while (i < group.length) {
412                         if (origin == index || group[i][0]
        == index) {
413                             this.uncheckedRemoveTransition(
        origin, input, stackRead, i);
414                             // group.splice(i, 1);
415                         } else {
416                             i++;
417                         }
418                     }
419                 });
420             });
421         });
422     }
423
424     private getPossibleActions(): Action[] {
425         let result: Action[] = [];
426
427         if (this.input.length > 0) {
428             this.handleInputSymbol(this.input[0], result);
```

```
429         }
430
431         this.handleInputSymbol(EPSILON_KEY, result);
432
433         return result;
434     }
435
436     private handleInputSymbol(inputSymbol: string, buffer:
    Action[]): void {
437         if (this.currentState === null) {
438             return;
439         }
440
441         let availableTransitions = this.transitions[this.
    currentState];
442         if (!availableTransitions.hasOwnProperty(inputSymbol
    )) {
443             return;
444         }
445
446         let indexedByStack = availableTransitions[
    inputSymbol];
447         let stackTop = this.stack[this.stack.length - 1];
448         if (!indexedByStack.hasOwnProperty(stackTop)) {
449             return;
450         }
451
452         let groups = indexedByStack[stackTop];
453         for (let group of groups) {
454             buffer.push({
455                 stepIndex: this.stepIndex + 1,
456                 currentInput: this.input,
457                 currentStack: utils.clone(this.stack),
458                 inputRead: inputSymbol,
459                 stackWrite: group[1],
460                 targetState: group[0]
461             });
462         }
463     }
464
465     private processAction(action: Action): void {
466         this.stepIndex = action.stepIndex;
467         this.input = action.currentInput;
468         this.stack = action.currentStack;
469
470         if (action.inputRead != EPSILON_KEY) {
471             this.input = this.input.slice(1);
472         }
473
474         this.stack.pop();
475
476         for (let i = 0; i < action.stackWrite.length; i++) {
```

```
477                 this.stack.push(action.stackWrite[i]);
478             }
479
480             this.currentState = action.targetState;
481         }
482
483         private addSymbol(location: SymbolLocation, symbol:
            string): void {
484             if (symbol.length > 0) {
485                 if (!this[location].hasOwnProperty(symbol)) {
486                     this[location][symbol] = 0;
487                 }
488                 this[location][symbol]++;
489             }
490         }
491
492         private addInputSymbol(symbol: string): void {
493             this.addSymbol("inputAlphabet", symbol);
494         }
495
496         private addStackSymbol(symbol: string): void {
497             this.addSymbol("stackAlphabet", symbol);
498         }
499
500         public removeSymbol(location: SymbolLocation, symbol:
            string): void {
501             if (symbol.length > 0) {
502                 this[location][symbol]--;
503                 if (this[location][symbol] == 0) {
504                     delete this[location][symbol];
505                 }
506             }
507         }
508
509         private removeInputSymbol(symbol: string): void {
510             this.removeSymbol("inputAlphabet", symbol);
511         }
512
513         private removeStackSymbol(symbol: string): void {
514             this.removeSymbol("stackAlphabet", symbol);
515         }
516
517         // Returns the number of states that are being stored
            inside
518         // this PDA (which counts removed states)
519         private realNumStates(): number {
520             return this.stateList.length;
521         }
522
523         private stateList: (State|undefined)[] = []; // Q
524         private inputAlphabet: Alphabet = {}; // sigma
525         private stackAlphabet: Alphabet = {}; // gamma
```

```
526     private transitions: {
527         [index: number]: {
528             [inputSymbol: string]: {
529                 [stackSymbol: string]:
    InternalTransitionInformation
530             }
531         }
532     } = {}; // delta (Q x sigma x gamma -> (Q x gamma*)+)
533     private initialState: Index = -1; // q0
534     private finalStates = new UnorderedSet<Index>(); // F
535 
536     private numRemovedStates: number = 0;
537 
538     // Instantaneous configuration-related attributes
539     private currentState: Index|null = null;
540     private stack: string[] = [];
541 
542     private input: string;
543     private stepIndex: number;
544     private actionTree: Action[] = [];
545     private halt: boolean = true;
546 
547     private acceptingHeuristic = AcceptingHeuristic.
    ACCEPTING_STATE;
548 }
```

Listing B.9: scripts/persistence/PersistenceHandler.ts

```
1  /// <reference path="../types.ts" />
2  
3  export interface AutomatonSummary {
4      aborted: boolean,
5      error: boolean,
6      initialState: State|null,
7      stateList: Map<State>,
8      edgeList: IndexedEdgeGroup<Edge<State>>
9  }
10 
11 /**
12  * Generic interface for classes that handle persistence.
13  * Any class that implements this interface can be used in
        AutomatonRenderer.
14  */
15 export interface PersistenceHandler {
16     save(stateList: Map<State>, edgeList: IndexedEdgeGroup<
    Edge<State>>,
17         initialState: State|null): string;
18     load(content: string): AutomatonSummary;
19 }
```

Listing B.10: scripts/System.ts

```
import {Keyboard} from "./Keyboard"
import {Settings} from "./Settings"

interface KeyboardObserver {
    keys: string[];
    callback: () => void;
    group?: string;
}

interface SpecialKeyMapping {
    altKey: boolean;
    ctrlKey: boolean;
    shiftKey: boolean;
}

type SpecialKey = keyof SpecialKeyMapping;

interface KeyboardKeyPress extends SpecialKeyMapping {
    keyCode: number;
    preventDefault: () => void;
}

interface LanguageChangeObserver {
    onLanguageChange: () => void;
}

interface MachineChangeObserver {
    onMachineChange: () => void;
}

const modifiers = ["alt", "ctrl", "shift"];

function propertyName(type: string) {
    return type + "Key";
}

/**
 * Handles keybinding management and a publish-subscribe system
 * for the main events of the application (i.e language-change
 * and machine-change).
 */
export class System {
    /**
     * Changes the system language and then notifies all
     * the language change observers.
     */
    static changeLanguage(language: Settings.Language): void {
```

```
48          Settings.changeLanguage(language);
49          for (let listener of this.languageChangeObservers) {
50              listener.onLanguageChange();
51          }
52      }
53
54      /**
55       * Changes the current machine and then notifies all
56       * the machine change observers.
57       */
58      static changeMachine(type: number): void {
59          Settings.changeMachine(type);
60          for (let listener of this.machineChangeObservers) {
61              listener.onMachineChange();
62          }
63      }
64
65      /**
66       * Registers a new language change observer, which will
         be notified
67       * when the system language changes.
68       */
69      static addLanguageChangeObserver(observer:
        LanguageChangeObserver): void {
70          this.languageChangeObservers.push(observer);
71      }
72
73      /**
74       * Registers a new machine change observer, which will
         be notified
75       * when the system machine changes.
76       */
77      static addMachineChangeObserver(observer:
        MachineChangeObserver): void {
78          this.machineChangeObservers.push(observer);
79      }
80
81      /**
82       * Triggers a key event as if the user himself
83       * had pressed the corresponding keys.
84       */
85      static emitKeyEvent(keys: string[]): void {
86          let event: Partial<KeyboardKeyPress> = {
87              preventDefault: function() {}
88          };
89
90          for (let modifier of modifiers) {
91              event[<SpecialKey> propertyName(modifier)] =
        false;
92          }
93
94          for (let key of keys) {
```

```
95              if (modifiers.indexOf(key) >= 0) {
96                  event[<SpecialKey> propertyName(key)] = true
    ;
97              } else {
98                  event.keyCode = Keyboard.keys[<Keyboard.Key>
     key.toUpperCase()];
99              }
100         }
101
102         this.keyEvent(<KeyboardKeyPress> event);
103     }
104
105     /**
106      * Notifies every non-locked keyboard observer that is '
    interested'
107      * in the triggered keyboard event.
108      */
109     static keyEvent(event: KeyboardKeyPress): boolean {
110         let triggered = false;
111         for (let observer of this.keyboardObservers) {
112             let keys = observer.keys;
113             if (!this.locked(observer) && this.
    shortcutMatches(event, keys)) {
114                 observer.callback();
115                 triggered = true;
116             }
117         }
118
119         if (triggered) {
120             event.preventDefault();
121             return false;
122         }
123         return true;
124     }
125
126     /**
127      * Binds a keyboard shortcut to the page.
128      * @param {string[]} keys  the keys that compose the new
     shortcut
129      * @param {() => void} callback a function to be called
    when the shortcut is activated
130      * @param {string}   group the group that this shortcut
    belongs to
131      */
132     static bindShortcut(keys: string[], callback: () => void
    , group?: string): void {
133         this.keyboardObservers.push({
134             keys: keys,
135             callback: callback,
136             group: group
137         });
138     }
```

```
139
140         /**
141          * Disables all shortcuts in a given shortcut group.
142          * @param {string} group the name of the shortcut group
143          */
144         static lockShortcutGroup(group: string): void {
145             this.lockedGroups[group] = true;
146         }
147
148         /**
149          * Enables all shortcuts in a given shortcut group.
150          * @param {string} group the name of the shortcut group
151          */
152         static unlockShortcutGroup(group: string): void {
153             delete this.lockedGroups[group];
154         }
155
156         /**
157          * Sets a global lock for keyboard shortcuts.
158          */
159         static blockEvents(): void {
160             this.eventBlock = true;
161         }
162
163         /**
164          * Unsets the keyboard shortcuts global lock.
165          */
166         static unblockEvents(): void {
167             this.eventBlock = false;
168         }
169
170         // Checks if a given keyboard event matches a given
        group of keys.
171         private static shortcutMatches(event: KeyboardKeyPress,
        keys: string[]): boolean {
172             if (this.eventBlock) {
173                 // Ignore all keyboard events if there's an
        active event block.
174                 return false;
175             }
176
177             let expectedModifiers: string[] = [];
178             for (let key of keys) {
179                 if (modifiers.indexOf(key) >= 0) {
180                     expectedModifiers.push(key);
181                     if (!event[<SpecialKey> propertyName(key)])
        {
182                         return false;
183                     }
184                 } else if (event.keyCode != Keyboard.keys[<
        Keyboard.Key> key]) {
185                     return false;
```

```
186             }
187         }
188
189         // Ignores the key combination if there are extra
    modifiers being pressed
190         for (let modifier of modifiers) {
191             if (expectedModifiers.indexOf(modifier) == -1) {
192                 if (event[<SpecialKey> propertyName(modifier
    )]) {
193                     return false;
194                 }
195             }
196         }
197
198         return true;
199     }
200
201     // Checks if a given keyboard observer is locked.
202     private static locked(observer: KeyboardObserver):
    boolean {
203         if (!observer.group) {
204             return false;
205         }
206
207         return this.lockedGroups.hasOwnProperty(observer.
    group);
208     }
209
210     private static keyboardObservers: KeyboardObserver[] =
    [];
211     private static languageChangeObservers:
    LanguageChangeObserver[] = [];
212     private static machineChangeObservers:
    MachineChangeObserver[] = [];
213     private static eventBlock: boolean = false;
214     private static lockedGroups: {[g: string]: boolean} =
    {};
215 }
```