Compiladores

Linguagem *ADV*

Departamento de Eletrónica, Telecomunicações e Informática
Universidade de Aveiro

abril de 2023

Objetivos

O objetivo geral deste trabalho é o desenvolvimento de um ambiente de programação, consituído pela linguagem de programação ADV (*automata description and visualization*) e correspondentes ferramentas de compilação, que permita a criação de programas numa linguagem de programação genérica (Java, C++, Python, ...) que, quando executados, permitam a visualização de autómatos finitos com animação e interação. Nas figuras 1 e 2, apresentam-se duas animações, cada uma constituída por uma sequência de 6 imagens, que ilustram a utilização da linguagem ADV. A primeira representa a construção de um autómato finito determinista completo; a segunda mostra o procedimento de reconhecimento da palavra `abcab` pelo autómato da figura 1.

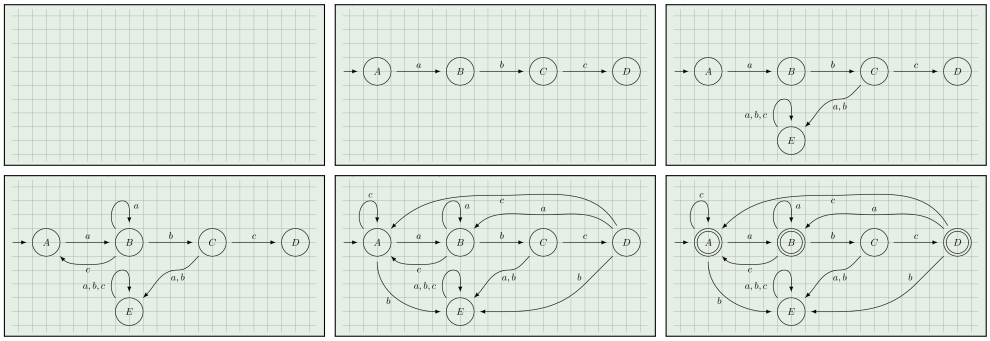

Figura 1: Animação em 6 passos para ilustração da construção de um autómato finito determinista completo.

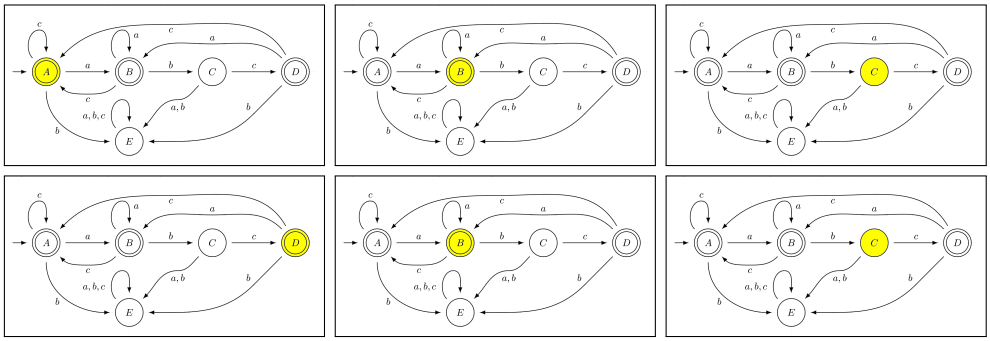

Figura 2: Animação em 6 passos para ilustração o processo de reconhecimento da palavra `abcab` pelo autómato da figura 1. A palavra é rejeitada porque não termina num estado de aceitação.

A concretização do objetivo geral deste projeto pode ser decomposto em:

- Definição da linguagem principal ADV, associada a ficheiros com a extensão `adv`, que permita a definição de automátos finitos, a definição de vistas desses automátos e a sua visualização com animação e interação. O resultado desta tarefa é a descrição em ANTLR de uma gramática adequada à análise sintática de descrições em ADV.
- Construção de um ou mais analisadores semânticos que detetem e sinalizem erros de descrição ADV que escapam à análise sintática. Compete(m) a este(s) analisador(es) verificar se as variáveis são declaradas adequadamente antes de ser utilizadas, se há repetições na declarações de variáveis, se as atribuições e operações são válidas, se os autómatos estão devidamente definidos, se as vistas desses autómatos estão devidamente definidos, etc. O resultado desta tarefa é o código fonte, na linguagem de trabalho escolhida, do(s) analisador(es) semântico(s) desenvolvido(s), assim como a sua adequada invocação no programa principal.
- Construção de um compilador que permite transformar uma descrição ADV num programa equivalente na linguagem destino escolhida e que, quando executado, produza a visualização desejada. Encoraja-se e valoriza-se o uso de *string templates*. O resultado desta tarefa é o código fonte, na linguagem de trabalho escolhida, do compilador (provavelmente um *visitor*), assim como a sua adequada invocação no programa principal, além do ficheiro dos *string templates*, se estes forem utilizados.
- Pretende-se que a linguagem principal permita a importação de elementos auxiliares através de descrições em linguagens secundárias, cujos elementos, em *runtime*, possam ser carregados pelo programa final. Uma das tarefas deste projeto é a definição de uma linguagem secundária, aqui designada por xADV, associado a ficheiros com a extensão `xadv`, que permita auxilar a linguagem ADV de alguma maneira (por exemplo, definindo estilos, tais como a forma dos estados ou o tipo e tamanho de fonte das etiquetas dos estados e das transições). O resultado desta tarefa é a gramática (ANTLR) da linguagem xADV, a construção gramatical na linguagem ADV que permite fazer a importação de descrições em xADV, o interpretador que permite fazer o *parsing* de descrições xADV, e exemplos ilustrando a sua utilização. Note que o interpretador tem de ser desenvolvido na linguagem destino e não na linguagem de trabalho, uma vez que a interpretação será feita em tempo de execução (*runtime*).

Características da solução

Uma descrição em ADV permite descrever um ou mais autómatos finitos que partilham um alfabeto comum. Para cada autómato, podem definir-se uma ou mais vistas, que representam formas visuais de o mostrar. Envolvendo uma ou mais vistas, pertencentes a um ou mais autómatos, podem definir-se animações que permitem apresentar os autómatos como uma sequência termporal de imagens, cuja evolução depende de interação com o utilizador. Finalmente, podem mandar-se executar essas animações.

Apresentam-se a seguir um conjunto de características que a solução desenvolvida pode ou deve contemplar. Essas características estão classificadas a 3 níveis:

- nível mínimo – características que a solução tem obrigatoriamente que implementar;
- nível desejável – características não obrigatórias, mas fortemente desejáveis que sejam implementadas pela solução;
- nível adicional – características apenas consideradas para avaliação se as mínimas e as médias tiverem sido contempladas na solução.

Nível mínimo

Os ficheiros `ex01.adv`, `ex02.adv` e `ex03.adv` representam descrições `ADV` que a solução de nível mínimo deve aceitar e compilar. Os ficheiros têm comentários que permitem perceber o significado de cada nova construção gramatical introduzida.

A linguagem `ADV`, na sua versão de nível mínimo, deve incorporar:

- Construção gramatical que permita definir o alfabeto. O alfabeto é um conjunto não vazio de símbolos. Os símbolos são representadas por caracteres entre plicas, limitados às letras e aos algarismos decimais.
- Construções gramaticais que permitam definir autómatos finitos dos tipos não-determinista (`ex01.adv`), determinista (`ex02.adv`) e determinista completo (`ex03.adv`). A validação do tipo deve ser feita semanticamente. No contexto de definição de autómatos, e apenas nele, devem existir instruções que permitam definir os estados e as transições que constituem os autómatos. Os exemplos mostram a sintaxe dessas instruções.
- Construção gramatical que permita definir a vista de um autómato. Assume-se que o autómato é desenhado numa tela (*canvas*) sem limites em termos de dimensões. No contexto de definição de vistas, e apenas nele, devem existir instruções que permitam posicionar os estados na tela, redefinir as setas que representam as transições e reposicionar as etiquetas dessas transições. Neste nível, as setas das transições podem ser implementadas por linhas poligonais.
- Construção gramatical que permita definir uma animação. As animações vão acontecer em janelas (`viewports`), que têm dimensões bem definidas em coordenadas de ecrã, e correspondem a sucessões de imagens cujo avanço é controlado pelo utilizador ou por tempo. No contexto de definição de animações, e apenas nele, devem existir instruções que permitam mostrar/esconder elementos (estados, transições) das vistas dos autómatos, assim como instruções para interação com o utilizador (por exemplo, esperar por uma tecla ou botão de rato antes de avançar para a próxima imagem).
- Instrução para a execução (`play`) de uma animação.
- Construção gramatical (lista de pares chave-valor entre parêntesis retos) para alteração das propriedades dos elementos intrínsicos (estado, transição) de um autómato. As propriedades podem estar relacionadas com a definição dos autómatos (por exemplo, a indicação do estado inicial), com as vistas (por exemplo, alinhamento da etiqueta de uma transição) e com as animações (por exemplo, mostrar um estado em realce).

- Os tipos de dados número (`number`), ponto (`point`) e lista, assim como álgebras que permitam a sua manipulação (ver exemplos).
- Instrução de iteração (`for .. in`) sobre os elementos de uma lista.
- Verificação semântica no uso de variáveis e de expressões.
- Verificação semântica da correção dos autómatos definidos.
- Verificação semântica da alteração de propriedades.

Nível desejável

O ficheiro `ex04.adv` representa, de forma incompleta, uma descrição ADV, que introduz alguns elementos gramaticais novos. Representa aproximadamente o exemplo apresentado na figura 2.

Desejável e adicionalmente à versão de nível mínimo, a linguagem ADV deve incorporar:

- Possibilidade de etiquetar uma transição com a palavra vazia. Nos autómatos finitos não-deterministas, além dos símbolos do alfabeto, é possível usar-se a palavra vazia como etiqueta de uma transição.
- Leitura de texto do terminal (ver exemplo).
- Instrução condicional, operando sobre expressões booleanas.
- Instrução repetitiva, controlada por expressão booleana.
- Expressões booleanas contendo pelo menos as operações de conjunção, disjunção e negação, com as precedências habituais.
- Na definição do alfabeto, possibilidade de se usar uma construção gramatical que represente uma sequência de símbolos (por exemplo, '1' – '7' para representar os algarismos entre '0' e '7'.

Nível adicional

Chama-se de novo a atenção de que acrescentos a este nível apenas serão avaliados se as características de níveis mínimo e desejável estiverem praticamente a 100%.

Adicionalmente a versão que cubra os níveis mínimo e desejável, a linguagem ADV pode incorporar:

- Existência de uma linguagem auxiliar, interpretada em tempo de execução.
- Representação das setas das transições por linhas curvas, tendo em consideração a propriedade `slope`.

- Representação das máquinas de Moore e das máquinas de Mealy. Relativamente aos autómatos finitos considerados até ao momento, estas máquinas não possuem um conjunto de estados de aceitação mas possuem uma função de saída definida sobre uma alfabeto de saída.
- Interpretação e visualização automática da evolução de um autómato ou máquina. No exemplo `ex04.adv`, a evolução do autómato face a uma palavra à entrada pode ser feita explicitamente no programa. É esse o propósito no nível desejável, de modo a introduzir a instrução condicional. Mas, dada a definição de autómato, é possível fazê-lo automaticamente.
- Para além de estados e transições, representação de outros elementos gráficos, como seja texto, apontadores, etc. Neste contexto, o tipo de dados string terá de ser considerado.
- Definição e invocação de funções.

Desafios

- Pretende-se criar uma animação:
  - onde se definam vários autómatos, deterministas e não deterministas, que se mostram ao utilizador numa grelha;
  - que permita ao utilizador escolher um deles, que passa a ocupar toda a tela;
  - que, a seguir, permita inserir uma palavra, que é executada passo a passo, controlado por pausa ou automaticamente com uma determinada cadência temporal, por escolha do utilizador, realçando o(s) estado(s) corrente(s) à medida que os símbolos da palavra forem sendo consumidos pelo autómato;
  - que permita voltar a inserir outra palavra ou voltar à grelha de seleção do autómato.

Exemplos

Exemplo ex01

```
/*
 * ex01: a quite simple example, that must be covered by the minimum version.
 */

// this is a single line comment

/*
 * This is a multi-line comment
 */

/*
 * All automata in an ADV description share the same alphabet,
 * which must be the first thing to be declared.
 *
 * It is a set of symbols (ASCII characters), symbols being specified between single quotes.
 *
 * In this example, the set is specified by a comma-separated list of items, enclosed in braces.
 */
alphabet { 'a', 'b', 'c' }

/*
 * An ADV section is devoted to define an automaton.
 * It is based on different keywords, depending on the type of automaton wanted.
 *
 * Keyword 'NFA' defines a non-determinist finite automaton,
 */
NFA a1 <<<

    /*
     * keyword 'state' allows to define states.
     */
    state A, B;

    /*
     * States have intrinsic properties.
     * Properties are represented by key-value pairs and always have default values.
     * From the automaton definition point of view, the possible properties for states are:
     * - 'initial', a boolean value indicating if the state is the initial one
     *   (only one state in an automaton can be the initial);
     * - 'accepting', a boolean value indicating if the state is an accepting one.
     *
     * Changing a property is specified with an assignment between square brackets.
     */
    A [initial = true];  // state A as the initial one
    B [accepting = true];  // state B as an accepting one

    /*
     * A set of symbols of the alphabet is associated to every ordered pair of states.
     * By default, the set is empty, meaning there is no transition between the
     * corresponding pair of states.
     *
     * Keyword 'transition' allows to add a transition.
     * Construction 'state 1 -> symbol(s) -> state 2' allows to add symbol(s) to the set of
     * transitions between state 1 and state 2.
     */
    transition
        A -> 'a','b' -> B,
        A -> 'a','b','c' -> A;
>>>

/*
 * The 'view .. of' construction allows to define
 * how a given automaton appears on a viewport.
 * The automaton itself must be already defined.
 *
 * A state is represented by a shape and a label.
 * - The default shape is a cicle with radius 0.5
 *   and the label appearing in the center.
 * - Accepting states include an additional circle with radius 0.4.
 * - Initial state includes an arrow, pointing to it.
 * - A set of properties may be used to change the default view of states.
 *
 * A transition is represented by an arrow connecting two states and a label.
 * - By default, there are default shapes for the arrows.
 *   If start and ending states are differente,
 *   the arrow is drawn over the straight line connecting the origins of the states,
 *   starting at a distance of radius plus 10% of the starting state
 *   and ending at a distance of radius plus 10% of the destination state.
 *   If they are the same, the arrow have a triangular shape.
 * - By default, the transition label is written in the middle of the arrow,
 *   slightly above it.
 * - A set of properties may be used to change the default view of transitions.
 *
```

```
 * In view of the above, the minimum requirements for a view is
 * the placement of the states in the viewport.
 */

/*
 * A view is necessarely associated to an automaton
 * The automaton's items (states and transitions) are
 * accessible inside the view.
 */
view v1 of a1 <<<
    /*
     * The 'place .. at' contruction allows to place states in the viewport
     * All states in the automaton must be placed.
     */
    place A at (2,1), B at (5,1);

>>>

/*
 * The 'animation' keyword allows to define an animation
 * It can include one or more viewports, one per automaton view.
 */

animation m1 <<<
    /*
     * create a viewport to show the view v1 of automaton a1,
     * with the upper-left corner at the given point
     */
    viewport vp1 for v1 at (10,10) -- ++(500,500);

    /*
     * Inside a viewport, the items of the associated view/automaton
     * are accessible
     */
    on vp1 <<<
        show A, B [accepting = false];
        pause;
        /*
         * Construction '<state 1,state 2>' represents all transitions from state 1 to state 2
         */
        show <A,B>;
        pause;
        show <A,A>;
        pause;
        show B [accepting = true];
        pause;
    >>>
>>>

play m1;
```

Exemplo ex02

```
/*
 * ex02: a quite simple example, that must be also covered by the minimum version.
 *
 * It corresponds to a DFA, equivalent to ex01
 *
 * In relation to example ex01, it introduces:
 * - the automaton type DFA;
 * - the point data type and operations on it;
 * - the manipulation of transitions in a view.
 */

/*
 * The alphabet
 */
alphabet { 'a', 'b', 'c' }

/*
 * Keyword 'DFA' defines a determinist finite automaton
 */
DFA a2 <<<

    /*
     * The states
     */
    state A, B;
    A [initial = true];  // state A as the initial one
    B [accepting = true];  // state B as an accepting one

    /*
     * The transitions
     */
    transition
        A -> 'a','b' -> B,
        A -> 'c' -> A,
        B -> 'c' -> A,
        B -> 'a','b' -> B;
>>>

/*
 * The view of a2
 */
view v2 of a2 <<<
    /*
     * Place states
     */
    place A at (2,1), B at (5,1);

    /*
     * The default view for transitions <A,B> and <B,A> overlap them.
     * Let modify the view of transition <B,A>,
     * which allows to introduce other ADV constructions.
     */

    /*
     * keyword 'point' allows to define a point
     */
    point p1;

    /*
     * A state reference within parentesis represents a point (its origin)
     */
    p1 = (B);

    /*
     * A point can be defined in both
     * cartesian (x,y) coordinates or polar (angle:norm) coordinates
     */
    point x1 = (200:0.6);

    /*
     * algebra on points is possible
     */
    p1 = p1 + x1;

    point p2 = (A) + (-20:0.6);
    point pm = (p1+p2)/2 + (0,0.2);

    /*
     * Transition arrows can be totally redefined, using points and slopes.
     * The result can be a polyline or a curse
     * Construction 'as .. --' allows to do that.
    */
    <B,A> as p1 -- pm -- p2;

    /*
     * The label of a transition is referenced with the construction '<.,.>#label'.
     *
     * The 'place .. at' contruction also allows to place labels of transitions
```

```
     */
    place <B,A>#label [align = below] at pa;
>>>

/*
 * The animation
 */

animation m2 <<<
    /*
     * create a viewport to show the view v2 of automaton a2.
     * It corresponds to a window with the upper-left corner at ((10,10),
     * and with width 500 and height 300.
     */
    viewport vp2 for v2 at (10,10) -- ++(500,300);

    /*
     * Inside a viewport, the items of the associated view/automaton are accessible
     */
    on vp2 <<<
        show A;
        pause;
        show <A,A>;
        pause;
        show B;
        show <A,B>;
        pause;
        show <B,B>;
        pause;
        show <B,A>;
        pause;
    >>>
>>>

play m1;
```

Exemplo ex03

```
/*
 * ex03: another example, that must be also covered by the minimum version.
 *
 * It corresponds to a complete DFA.
 *
 * In relation to example ex01 and ex02, it introduces:
 * - the automaton type complete DFA;
 * - the 'foreach' construction.
 */

/*
 * The alphabet
 */
alphabet { 'a', 'b', 'c' }

/*
 * Keywords 'complete' 'DFA' defines a determinist finite automaton,
 * in which all states and transitions are explicitly represented
 */
complete DFA a3 <<<

    /*
     * The states
     */
    state A, B, C, D, E;
    A [initial = true];  // state A as the initial one

    /*
     * A 'foreach' construction applied to sets exist.
     */
    for s in {{ A, B, D }}
        s [accepting = true];

    /*
     * The transitions
     */
    transition
        A -> 'a' -> B,
        B -> 'b' -> C,
        C -> 'c' -> D,
        C -> 'a','b' -> E,
        E -> 'a','b','c' -> E,
        B -> 'a' -> B,
        B -> 'c' -> A,
        A -> 'c' -> A,
        D -> 'c' -> D,
        D -> 'a' -> B,
        D -> 'b' -> E,
        A -> 'b' -> E;
>>>

/*
 * The view of a3
 */

/*
 * A view is necessarely associated to an automaton
 * The automaton's items (states and transitions) are
 * accessible inside the view.
 */
view v3 of a3 <<<
    /*
     * The 'grid' construction allows to draw a grid.
     * A grid has several properties
     */
    grid g3 (21,10) [  // width = 21; height = 10
        step = 0.5,        // cell size
        margin = 0.25,     // external margin (must be lower than step)
        color = gray,      // stroke color
        line = solid       // may also be dotted or dashed
    ];

    /*
     * Place states
     */
    place A at (2,1), B at (5,1), C at (7,1), D at (10,1);
    place E at (4.5,4);

    /*
     * Redefine some of the transitions, including their label positions
     */
    point p1, p2, pm;

    p1 = (B) + (200:0.7);
    p2 = (A) + (20:0.7);
    pm = ((A) + (B)) / 2 + (0,0.5);
    <B,A> as p1 -- pm -- p2;
    place <B,A>#label [align=above] at pm;
```

```
    p1 = (D) + (200:0.7);
    p2 = (B) + (20:0.7);
    pm = ((D) + (B)) / 2 + (0,1.0);
    <D,B> as p1 -- pm -- p2;
    place <D,B>#label [align=above] at pm;

    /*
     * Change the location of some transition labels
     */
    <A,E>#label [align=below left];
    <D,E>#label [align=below right];
    <C,E>#label [align=right];
    <E,E>#label [align=left];
>>>

/*
 * The animation
 */

animation m3 <<<
    /*
     * create a viewport to show view v3 of automaton a3,
     */
    viewport vp3 for v3 at (10,10) -- ++(500,300);

    /*
     * Inside a canvas, the items of the associated view/automaton are accessible
     */
    on vp3 <<<
        show g3;
        pause;
        for i in {{ A, B, D }} <<<
            show i [accepting = false];
        >>>
        show C, <A,B>, <B,C>, <C,D>;
        pause;
        show E, <C,E>, <E,E>;
        pause;
        show <B,B>, <B,A>;
        pause;
        show <A,A>, <A,E>, <D,D>, <D,E>, <D,B>;
        pause;
        for i in {{ A, B, D }} <<<
            show i [accepting = true];
        >>>
        pause;
    >>>
>>>

play m1;
```

Exemplo ex04

```
/*
 * ex04: This example corresponds roughly to the one in figure 2 of the ADV description PDF document.
 *
 * It illustrates the acceptance or rejection procedure of an input word by a complete DFA.
 *
 * In relation to previous examples, it introduces:
 * - the 'slope' property for transitions;
 * - the 'highlighted' property for states;
 * - the read instruction
 */

/*
 * The alphabet
 */
alphabet { 'a', 'b', 'c' }

/*
 * Keywords 'complete' 'DFA' defines a determinist finite automaton,
 * in which all states and transitions are explicitly represented
 */
complete DFA a4 <<<

    /*
     * The states
     */
    state A, B, C, D, E;
    A [initial = true];  // state A as the initial one

    /*
     * A 'foreach' construction applied to sets exist.
     */
    for s in {{ A, B, D }}
        s [accepting = true];

    /*
     * The transitions
     */
    transition
        A -> 'a' -> B,
        B -> 'b' -> C,
        C -> 'c' -> D,
        C -> 'a','b' -> E,
        E -> 'a','b','c' -> E,
        B -> 'a' -> B,
        B -> 'c' -> A,
        A -> 'c' -> A,
        D -> 'c' -> A,
        D -> 'a' -> B,
        D -> 'b' -> E,
        A -> 'b' -> E;
>>>

/*
 * The view of a4
 */
view v4 of a4 <<<
    /*
     * Place states
     */
    place A at (2,1), B at (5,1), C at (7,1), D at (10,1);
    place E at (5,4);

    /*
     * Redefine some of the transitions, including their label positions
     */
    point p1, p2, pm;

    p1 = (B) + (235:0.7);
    p2 = (A) + (45:0.7);
    pm = ((A) + (B)) / 2 + (0,-0.5);
    <B,A> as p1 [slope=235] -- pm [slope=0] -- p2 [slope=45];
    place <B,A>#label [align=below] at pm;

    p1 = (D) + (150:0.7);
    p2 = (B) + (30:0.7);
    pm = ((D) + (B)) / 2 + (0,1.0);
    <D,B> as p1 [slope=150] -- pm [slope=0] -- p2 [slope=30] ;
    place <D,B>#label [align=above] at pm;

    p1 = (D) + (120:0.7);
    p2 = (A) + (60:0.7);
    pm = ((D) + (A)) / 2 + (0,1.5);
    <D,A> as p1 [slope=120] -- pm [slope=0] -- p2 [slope=60] ;
    place <D,A>#label [align=above] at pm;

    /*
     * Change the location of some transition labels
     */
```

```
    <A,E>#label [align=below left];
    <D,E>#label [align=below right];
    <C,E>#label [align=right];
    <E,E>#label [align=left];
>>>

/*
 * The animation
 */
animation m4 <<<
    /*
     * create a viewport to show view v4 of automaton a4,
     */
    viewport vp4 for v4 at (10,10) -- ++(500,300);

    on vp4 <<<
        /*
         * The show instruction without arguments applies to the whole view
         */
        show;

        /*
         * ask user for a word, which is assumed to be composed only of symbols from the alphabet
         */
        string word = read [prompt="Insira uma palavra: "];

        /*
         * execute automaton with word
         */
        state cs = A;
        show cs [ highlighted = true ];
        pause;
        for l in word <<<
            /* depending on the value of l, highlight the appropriate states */
            pause;
        >>>
    >>>
>>>

play m1;
```