HaLeX: A Haskell Library to Model, Manipulate and Animate Regular Languages

João Saraiva

Department of Computer Science,
University of Minho, Braga, Portugal
jas@di.uminho.pt

Abstract. *HaLeX* is a library of datatypes and functions implemented in HASKELL that allows us to model, manipulate and animate regular languages. This paper briefly summarizes the design of *HaLeX*. It presents some examples constructed with the library and discusses the use of *HaLeX* in education.

1 Introduction

This paper presents the HASKELL library *HaLeX* that enables us to model, manipulate and animate regular languages. This library was developed in the context of a programming methodology course for undergraduate students, and as a consequence, it was defined mainly for educational purposes. Indeed, it provides a clear, efficient and concise way to define, to understand and to manipulate regular languages in HASKELL. Furthermore, the construction of the complete library has been proposed as assignment projects to the students following the course. *HaLeX* is now being used to support this course.

This library introduces a number of HASKELL datatypes and the respective functions that manipulate them, providing the following facilities:

- The definition of deterministic finite automata, non-deterministic finite automata, and regular expressions directly and straightforwardly in HASKELL.
- The definition of the acceptance functions for all those models.
- The transformation from regular expressions into non-deterministic finite automata (NDFA) and from NDFA into deterministic finite automata (DFA).
- The minimization of the number of states of deterministic finite automata.
- The equivalence of regular expressions and finite automata.
- The graphical representation of finite automata.
- The definition of Lex-like facilities, for example, it generates from a regular expression an efficient recognizer for the language: a HASKELL program based on the equivalent minimized DFA.
- The definition of reactive finite automata. In order to allow that an automaton reacts while accepting an input sentence, the library defines a monadic finite automaton. The reactions are defined straightforwardly as monadic functions, that are easily embedded in the automaton definition.
- The automatic animation of the acceptance function of finite automata.

In this paper I summarize how this library is implemented in HASKELL and I briefly discuss its use in education. *HaLeX* is available at the following `http` address:

`http://www.di.uminho.pt/~jas/Research/HaLeX/HaLeX.html`

2 The HaLeX Library

The *HaLeX* library is a small extension to [Tho00,JS99]. Basically, it extends definitions in these papers with the graphic visualization of finite automata and the definition of reactive finite automata.

2.1 Regular Expressions and Finite Automata

Regular expressions (RE) are an effective and concise notation for defining regular languages. In order to define regular expressions directly in HASKELL we introduce a new algebraic datatype *RegExp*. With this datatype we associate type constructor functions. Such functions allow us to construct values of type *RegExp*. Thus, we associate a *RegExp* data constructor to each of the five sorts of regular expressions (we omit the constructor of the empty language).

```
data RegExp = Epsilon                      -- ϵ
            | Literal Char                 -- a
            | Or      RegExp  RegExp       -- p + q
            | Then    RegExp  RegExp       -- pq
            | Star    RegExp               -- p*
```

Using these data constructors we are able to write regular expressions in HASKELL. For example, the regular expression $p = a + ba^*$ is written as:

```
p  = Or a (Then b (Star a))
where a = Literal 'a'
      b = Literal 'b'
```

In HASKELL we may write functions between their (two) arguments (*i.e.*, in infix form) by enclosing the function in back quotes. So the above RE becomes a ‘OR‘ (b ‘THEN‘ (STAR a)). To write RE in concrete notation we constructed a parser, via parser combinators, for a Unix-like RE notation.

A finite automaton is a computational machine that either accepts or rejects sequences of symbols of an alphabet. Informally, a Finite Automaton (FA) is a tuple of five elements: the alphabet, the finite set of states, the start state(s), the final (or accepting) states and the transition function. Next, we present the HASKELL-definition of finite automata we included in *HaLeX*: deterministic finite automata (left) and non-deterministic finite automata (right). Although tuples are built-in in HASKELL, and, thus, could be used to represent five-tuple, we prefer to follow the approach of [Tho00], and for readability purposes only, we introduce a new HASKELL datatype *Dfa* (*Ndfa*), with a single constructor with the same name, to model, respectively, deterministic and non-deterministic FA. It is parameterized with the types of states and symbols. To model sets (of states) we use the HASKELL built-in lists and respective set-operations on lists.

```
data Dfa st sy                  data Ndfa  st  sy
    = Dfa [sy]                      = Ndfa [sy]                       -- Alphabet
          [st]                             [st]                       -- Set of states
          st                               [st]                       -- Start state(s)
          [st]                             [st]                       -- Final states
          (st -> sy -> st)                 (st -> Maybe sy -> [st])   -- Transition function
```

We define non-deterministic finite automata with ϵ-transitions. To model such transitions we use the pre-defined datatype `data` *Maybe* a = NOTHING | JUST a, where the constructor NOTHING models ϵ-transitions, while the constructor JUST models transitions labeled by its argument symbol. Having introduced the datatypes to model FA, we can now define DFA and NDFA straightforwardly in HASKELL:

```
dfa = Dfa ['a','b'] [1,2,3,4] 1 [2,3] delta
  where  delta 1 'a' = 2
         delta 1 'b' = 3
         delta 2 'a' = 4
         delta 2 'b' = 4
         delta 3 'a' = 3
         delta 3 'b' = 4
         delta 4 'a' = 4
         delta 4 'b' = 4
```

```
ndfa = Ndfa ['a','b'] [1,2,3,4]
            [1] [4] delta
  where  delta 1 (Just 'a') = [4]
         delta 1 (Just 'b') = [2]
         delta 2 (Just 'a') = [3]
         delta 2 Nothing    = [3]
         delta 3 Nothing    = [2,4]
         delta _  _         = []
```

(a graphic representation of the automaton `dfa`, on the left, is given in section 2.2)

The acceptance function of a DFA expresses the recursion pattern captured by the HASKELL prelude *foldl* function. Thus, the function *accept* is concisely defined as follows:

$$\mathit{accept} :: \mathit{Eq}\ \ \mathit{st} \ \Rightarrow \ (\text{DFA}\ \mathit{st}\ \mathit{sy}) \ \rightarrow \ [\mathit{sy}] \ \rightarrow \ \mathit{Bool}$$
$$\mathit{accept}\ (\text{DFA}\ \ v\ \ q\ \ s\ \ z\ \ d)\ \ \mathit{sy} = (\mathit{foldl}\ \ d\ \ s\ \ \mathit{sy})\ \ \text{`}\mathit{elem}\text{`}\ \ z$$

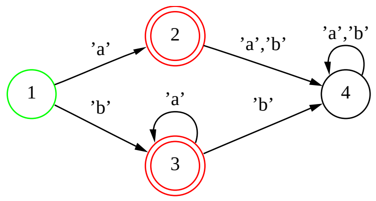
The set of strings accepted by a FA, say $\mathcal{A}$, is the *language* of $\mathcal{A}$ and it is denoted by $\mathcal{L}_{\mathcal{A}}$.

2.2 Graphic Representation of Finite Automata

Finite automata are more comprehensible in a graphical representation. The following representation is usually used: states are depicted as the nodes of a graph, final states get a double circle, and the initial states are explicitly mentioned usually with an incoming arrow. The transition function induces arrows connecting the nodes of the graph: for each $d\ q_i\ s\ =\ q_j$ there is an arrow labeled by s from q_i to q_j.

Drawing graphs is a complex task and an intensive area of research. Thus, instead of defining a graph visualization system from scratch, and reinventing the wheel, *HaLeX* synthesizes a graph representation for an existing high quality graph visualization system: the GraphViz system, developed and available at *AT&T* Labs [GN99]. GraphViz provides a collection of tools for manipulating graph structures and generating graph layouts. The input is a description of the graph in the *Dot* language. Next, we present an example of the input *Dot* text (left) and the outputed graph as displayed by one of GraphViz tools, the `dot` tool (right).

```
digraph HaLeX {
 rankdir = LR ;
 "1" [shape=circle , color=green];
 "2" [shape=doublecircle , color=red];
 "3" [shape=doublecircle , color=red];
 node [shape=circle , color=black];
 "1" -> "2" [label = "'a'"];
 "1" -> "3" [label = "'b'"];
 "2" -> "4" [label = "'a','b'"];
 "3" -> "3" [label = "'a'"];
 "3" -> "4" [label = "'b'"];
 "4" -> "4" [label = "'a','b'"];
}
```



The *HaLeX* library includes a *pretty-printing* function, named *toGraphViz* that, given a FA, produces its (textual) graph representation in the *Dot* language. In order to "*beautify*" the graphical representation, we have included options to ignore the *sink* states and to fuse transitions with the same origin and destination. We omit here its trivial implementation. The above *Dot* text is the result of *toGraphViz* given RE p as argument (section 2.1). Colors are used to mark the initial state(s) (green) and the final ones (red).

2.3 Manipulating Regular Expressions and Finite Automata

The *HaLeX* library includes several functions that manipulate finite automata and regular expressions. For FA we have defined functions to implement the usual language operations, namely the exponentiation, concatenation, union, and Kleene and positive closure. For example, the function *union* (*concat*) performs the union (concatenation) of two FAs, *i.e.*, it takes two FAs as arguments, say $\mathcal{A}_1$ and $\mathcal{A}_2$, and returns a NDFA, say $\mathcal{A}_3$, accepting the language $\mathcal{L}_{\mathcal{A}_3} = \mathcal{L}_{\mathcal{A}_1} \cup \mathcal{L}_{\mathcal{A}_2}$ ($\mathcal{L}_{\mathcal{A}_3} = \mathcal{L}_{\mathcal{A}_1}\mathcal{L}_{\mathcal{A}_2}$), respectively. The functions *exp*, *star* and *plus* take a FA $\mathcal{A}$ and return a NDFA accepting the languages $\mathcal{L}^i_{\mathcal{A}}$ (i is an extra argument of *exp*), $\cup_{i=0}^{\infty}\mathcal{L}^i_{\mathcal{A}}$ and $\cup_{i=1}^{\infty}\mathcal{L}^i_{\mathcal{A}}$, respectively. The function *equiv* takes two FAs, say $\mathcal{A}_1$ and $\mathcal{A}_2$ and checks whether they are equivalent or not, *i.e.*, $\mathcal{L}_{\mathcal{A}_1} \equiv \mathcal{L}_{\mathcal{A}_2}$.

In order to show how such functions are nicely and concisely constructed in a modern, polymorphic and higher-order functional language as HASKELL, we briefly present the functions that minimize the number of states of a DFA (*minimize*) and the mapping from regular expressions into NDFA (*re2ndfa*). We start by defining a function *reverse* that accepts the reverse language of its argument DFA.

```
reverse :: Eq st => Dfa st sy -> Ndfa st sy
reverse (Dfa v q s z d) = Ndfa v q z [s] d'
 where d' st (Just sy) = [ q | q <- qs , d sy == st ]
       d' st Nothing = []
```

Now, we can implement the elegant Brzozowski's algorithm [Brz62] to minimize the number of states of a DFA by reverting the FA twice[1]:

```
minimize :: (Eq sy, Ord st) => Dfa st sy -> Dfa [[st]] sy
minimize = ndfa2dfa . reverse . ndfa2dfa . reverse
```

where the function *ndfa2dfa* is the function converting a NDFA to a DFA. This function fuses states of the NDFA in order to construct an equivalent DFA. Thus, the states of the DFA are sets (lists in our case) of states of the NDFA. Because we are applying the function twice, the type of the states of the resulting DFA are lists of lists of states. Obviously, such (possible) long state identifiers may obscure the *HaLeX* graphical representation. Thus, we have a function *rename* that renames the states of an FA assigning integer numbers to the states (and starting at a giving value).

In developing the *HaLeX* library we express our recursive algorithms through standard recursion operators, rather than using explicitly recursion [JS99]. Next, we present the function *foldRE* that is induced by datatype *RegExp* (left) and we elegantly express the function *re2ndfa* which converts regular expressions into NDFA as a fold function (right).

```
foldRE (epsilon, lit, or, then, star) = fold
 where fold Epsilon = epsilon
       fold (Literal a) = lit a
       fold (Or l r) = or (fold l) (fold r)
       fold (Then l r) = then (fold l) (fold r)
       fold (Star er) = star (fold er)
```

```
re2ndfa :: RegExp -> Ndfa Int Char
re2ndfa = foldRE ( Ndfa [] [1] [1] [1] ( _ _ -> [])
                 , literal
                 , union
                 , concat
                 , star )
```

The functions *union*, *concat* and *star* are the ones discussed at the beginning of this section. We omit the trivial definition of *literal* that given a symbol, returns a NDFA that accepts that symbol only.

HaLeX includes other functions that manipulate regular expressions. For example, the functions *toHaskell* and *toGraphViz* produce, given a RE, an efficient recognizer for the language (*i.e.*, a HASKELL program based on the minimized DFA), and a graphical representation (based on a NDFA or DFA), respectively. Actually, the HASKELL-based FA shown in section 2.1 and the graph in section 2.2 were produced by such functions given the regular expression *p* (section 2.1).

The Language of Real Numbers: Let us consider a very simple language of real numbers. This language is specified by the regular expression `er` shown on Fig.1, where `d` represents the class of digits (we do not enumerate the digits to simplify the NDFA). Fig.2 shows the HASKELL-based DFA generated by *HaLeX* from `er`. Fig.3, Fig.4 and Fig.5 contain the NDFA, DFA and the minimized DFA generated by *HaLeX* and displayed by `dot`.

[1] This algorithm is little known despite its amazing simplicity. It was Doaitse Swierstra who first explained the algorithm informally to me. But, its HASKELL definition says more than one thousand words!

```
re = ('+'|'-')?d*('.')?d+
```

Fig.1: *The regular expression (in the HaLeX Unix-like notation).*

```
dfa = Dfa v q s z delta
  where  v = "+-d."
         q = [1,2,3,4,5,6]            delta 2 'd' = 3
         s = 1                        delta 2 '.' = 4
         z = [3,6]                    delta 3 'd' = 3
         delta 1 '+' = 2              delta 3 '.' = 4
         delta 1 '-' = 2              delta 4 'd' = 6
         delta 1 'd' = 3              delta 6 'd' = 6
         delta 1 '.' = 4              delta _ _   = 5
```

Fig.2: *The* HASKELL*-based minimized DFA derived from* re.

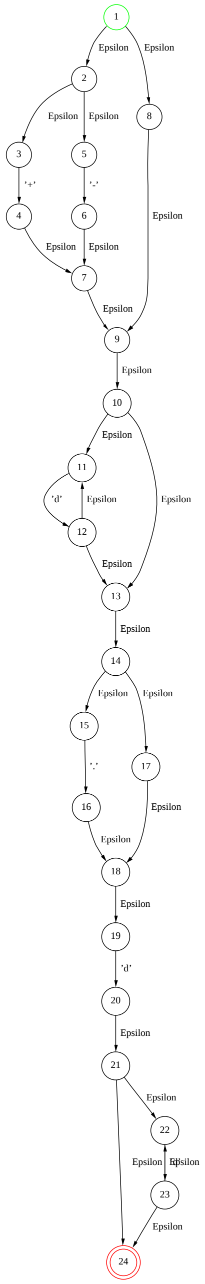


Fig.3: *NDFA (for* re*).*

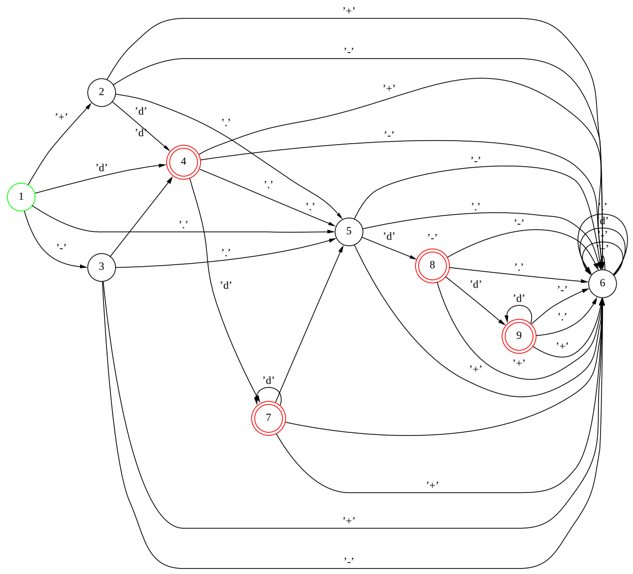


Fig.4: *The graphical representation of the DFA derived from* re.

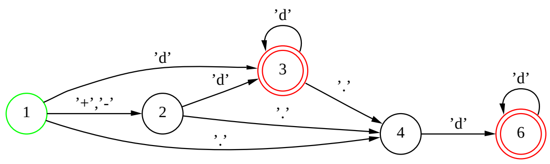


Fig.5: *The minimized and "beautified" DFA.*

3 Reactive Finite Automata

While accepting a sentence, we may wish the automaton to react. For example, we may wish to compute some trace information (for debugging purposes), or to perform some IO operations, or to compute semantic functions (in our example of real numbers, to convert the sequence of characters into its real value). A reactive finite automaton (RFA) is a machine that reacts while moving from one state to another. To add reactions to finite automata we use monads and we define monadic finite automata. Because we wish to associate different effects to finite automata, we parameterize the type of the DFA with the type of the monad. The reactions are defined in the (monadic) transition function. As a result, those functions not only have to indicate the destination state(s), but also the reactions to be performed in the transitions. Reactive deterministic finite automata are defined as follows:

```
data Dfa m st sy = Dfa [sy] [st] st [st]      -- Alphabet, states, start and final states
                       (st → sy → m st)       -- Monadic transition function
```

The acceptance function is now defined as the standard HASKELL monadic *foldM* function:

```
dfaaccept :: (Monad m , Eq st) ⇒ Dfa m st sy → [sy] → m Bool
dfaaccept (Dfa v q s z d) sent = do { st <- foldM d s sent ;
                                      return (st ‘elem‘ z) }
```

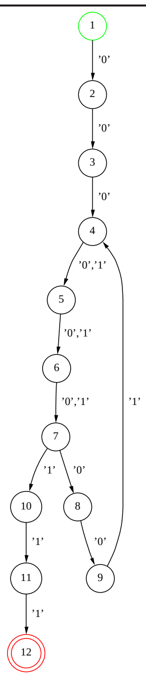
Having defined monadic DFA, different type of reactions are now easily embedded in the automata by simply defining different monadic functions. Next, we present two possible transition functions: `deltaIO` (left) performs IO operations (by sending the consumed symbol to the output), while `delta` (right) does not perform any reaction at all.

```
deltaIO 1 'A' = do putStrLn "A"                delta 1 'A' = return 2
                   return 2
```

A typical operation that we wish to model via reactive automata is to store/accumulate values in global variables for further processing. We can mimic this imperative feature by using the state monad. Next we present an example of such a RFA.

Communication protocol: We want to construct a program to implement the communication protocol between two electronic components. The communication is established by sending an initial sequence of bits `000`. Then, several 3-bit values are sent in order to configure different parameters of the components. Those values are separated by the special sequence of bits `001`. The communication ends after the sequence of bits `111` is sent. Consider also that to configure the components the program has to compute the integer values transmitted.

The protocol is specified through the DFA displayed on Fig.6. To compute the desired values we will use two “global variables”: one to accumulate the bits defining a value (type `[`*Char*`]`) and another to store the integers (type `[`*Int*`]`). Thus, the type of our state (monad) is the tuple (`[`*Char*`]`, `[`*Int*`]`). Now, we have just to define the monadic functions that manipulate this state, and to associate them to the transitions, where needed. Fig.6 shows this RFA (right).




```
ptl :: Dfa (State ([Char],[Int])) Int Char
ptl = Dfa ['1','0'] [1,2,3,4,5,6,7,8,9,10,11,12,13] 1 [12] delta
  where delta 1  '0' = return 2
        delta 2  '0' = return 3
        delta 3  '0' = return 4
        delta 4   s  = do { bits s ; return 5 }
        delta 5   s  = do { bits s ; return 6 }
        delta 6   s  = do { bits s ; return 7 }
        delta 7  '0' = do { values ; return 8 }
        delta 7  '1' = do { values ; return 10 }
        delta 8  '0' = return 9
        delta 9  '1' = return 4
        delta 10 '1' = return 11
        delta 11 '1' = return 12
        delta _   _  = return 13
        bits x   = modify (\ s -> ((fst s)++[x],snd s))
        values   = modify (\ s -> ("",snd s ++ [bits2int (fst s)]))
```

Fig.6: The reactive DFA: in the transitions with origin in states 4, 5, and 6, the RFA accumulates (in the first component of the state) the transmited bit-value. After that, *i.e.*, state 7, the function `values` converts the stored 3-bit value into its integer value (function `bits2int`), accumulates the value in the second component of the state, and initializes the first one. (`modify` is defined in the HASKELL monad library and it modifies the state)

HaLeX includes the function *runDfa* that given a monadic DFA, a sentence and an initial state, returns a pair with the result of accepting/rejecting the sentence and the final value of the state. An example of running the RFA of Fig.6 is given next.

```
>runDfa ptl "000010001111001000111" ([],[])
(True,("",[2,7,0]))
```

3.1 Animation of the Acceptance Function

Informally, we say that an automaton accepts a string s if it is possible to find a *path* from a initial state to a final one, such that the concatenation of the symbols labeling the performed transitions form s. We call this path an *accepting path* (or *rejecting path* if the string is rejected) and it consists of the sequence of nodes visited during the acceptance process.

To animate the acceptance of a string we can use this notion of path. The animations are just the step-by-step display of such a path in the graphic representation of the automaton. Thus, the sequence of visited nodes is the trace information we need to animate finite automata. This trace information can be easily computed using our reactive FA: the monadic automata just have to accumulate in the state the nodes being visited.

Fig.7 shows a fragment of the monadic DFA that uses the state monad to store the list of visited nodes. *HaLeX* derives such monadic DFA from regular expressions (function *re2MHaskell*). Fig.8 displays the animation of the DFA for the language of real numbers. We have defined a script (in `lefty`, a scripting language used by one of GraphViz tools) that, given the sequence of nodes and the graphic representation of the automaton, animates the acceptance function.

```
mdfa :: Dfa (State [Char]) Int Char
mdfa = Dfa v q s z delta
  where
    v = "+-d."
    q = [1,2,3,4,5,6]
    s = 1
    z = [3,6]
    delta 1 '+' = do accum '2'
                     return 2
    delta 1 '-' = do accum '2'
                     return 2
    delta 1 'd' = do accum '3'
                     return 3
    delta 1 '.' = do accum '4'
                     return 4
    delta 2 '+' = do accum '5'
                     return 5
    delta 2 '-' = do accum '5'
                     return 5
    delta 2 'd' = do accum '3'
                     return 3
    delta 2 '.' = do accum '4'
                     return 4
    delta 3 'd' = do accum '3'
                     return 3

    accum x = modify (\ s -> s++[x])
```

Fig.7: *Reactive DFA.* The DFA accumulates in the state monad the sequence of nodes visited.

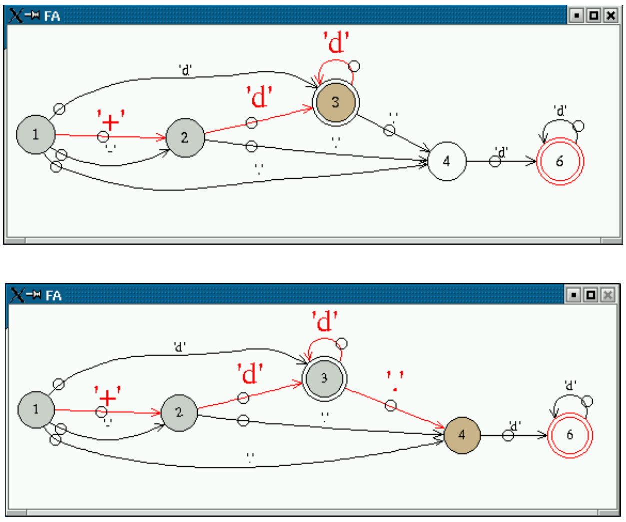


Fig.8: *Animation of Finite Automata.* The two figures display two consecutive snapshots of the "animated" acceptance function for the minimized DFA derived from `re` given the sentence `"+dd.dd"` as input. Colors are used to mark both the visited nodes (shadowed ones) and the edges used in the moves (red edges with larger fonts). Thus, the snapshots show the moments before and after accepting the character `'.'` of the input sentence. After consuming the last symbol of the input, the path from the initial to a final state, performed while accepting the sentence, is highlighted in the graph. The animation is defined by the *HaLeX* script: the graphic representation of the DFA and the visit sequence can be loaded in and animated, either in a single step or in continuous mode, forwards and backwards.

4 Using HaLeX

Having defined the *HaLeX* library, we can now define useful tools to manipulate and visualize regular languages. We have constructed the `halex` program entirely in *HaLeX*. A simple *main* function was defined and the option handling `GetOpt` (a HASKELL library) was used. The HASKELL automata and the graphical representations included in this paper have been produced with such a tool. For example, the NDFA of Fig.3 is produced by the following Unix-pipe:

```
halex -N -G -R"('+'|'-')?d*('.')?d+" | dot -Tps
```

Next, we use `halex` to check whether two regular expressions are equivalent or not (left), and we prove one law of the algebra of regular expressions (right).

```
halex -R"aa+" -R"a+"
  The REs are not equivalent
```

```
halex -R"a*" -R"(a+)?"
  The REs are equivalent
```

5 HaLeX in Education

We started developing the *HaLeX* library two years ago in the context of a third year course on programming methodology. This course has a working load of 24 hours of theoretical classes and another 24 hours of laboratory classes, running for 12 weeks (*i.e.*, a semester). The theoretical classes introduce the basic concepts of regular expressions, finite automata and context-free languages. *HaLeX* is used to support such classes. In the laboratory (a two hour class per week) the students have to solve exercises using a computer. We have defined eleven exercise sets (one per week), using literate HASKELL, that the students have to complete. Each set of exercises defines a module of the *HaLeX* library. Thus, at the end of the course the students have a complete documentation of all the exercises and topics covered in the course, and, also, of the *HaLeX* library.

Furthermore, during the semester the students have to complete two assignment projects. The construction of parts of the *HaLeX* library has been proposed as such a project. For example, in the last instance of the course the first project was the implementation of the minimization and the visualization of a DFA, while the second one was the definition of a concrete notation for regular expressions, the implementation of its parser (through parser combinators), and the construction of (a simplified version of) the halex program.

6 Conclusion

This paper summarizes the design and implementation of the *HaLeX* library. Two examples are presented, namely the visualization/animation of a language of real numbers, and the specification of a simple communication protocol through a reactive finite automata.

Acknowledgements: I would like to thank the students who followed the course *Métodos de Programação III* in the scholar years 2000/2001 and 2001/2002 for their enthusiasm and cooperation in implementing parts of the library during classes. I thank José Bacelar Almeida who gave helpful comments during the development of *HaLeX*.

References

[Brz62] J. A. Brzozowski. Canonical regular expressions and minimal state graphs for definite events. *Mathematical theory of Automata*, 12:529–561, 1962.

[GN99] Emden R. Gransner and Stephen C. North. An open graph visualization system and its applications to software engineering. *Software Practice and Experience*, 00(S1):1–29, 1999.

[JS99] Johan Jeuring and Doaitse Swierstra. Advanced Programming Concepts in a Course on Grammars and Parsing. In Matthias Felleisen, Michael Hanus, and Simon Thompson, editor, *Proceedings of the Workshop on Functional and Declarative Programming in Education*, Rice Technical Report 99-346, August 1999.

[Tho00] Simon Thompson. Regular expressions and automata using haskell. Technical Report 5-00, Computing Laboratory, University of Kent, January 2000.