Teaching Nondeterministic and Universal Automata using SCHEME

Christian Wagenknecht* and Daniel P. Friedman†
Computer Science Department
Indiana University
Bloomington, IN 47405

October 2, 1996

Abstract

A serious study of computer science should include the theory of formal languages, abstract automata, computability, and complexity. The primary advantage of delivering this material is to teach the foundations and limitations of computation. A knowledge of standard proof methods and modeling is important for the development of good problem solving skills in the young computer scientist, thus, the theoretical aspect of this material should be taught in a very deliberate manner. There is a danger that students will proficiently, yet blindly, execute transition functions of formally defined automata without ever attaining a deeper understanding of the semantics of the machines. Furthermore, many different, but equivalent, definitions of the various terms abound in the literature. This equivalence is a pleasure for the teacher, but can cause confusion for the beginner.

We describe a new way of teaching students *how to think about abstract automata*. We restrict our attention to *acceptors, i.e., transition machines* with a particular *communication unit* interacting with a local tape object. The transition behavior is the defining property of an automaton. However, the tape is updated (reconfigured) in different ways, depending on the type of machine. The communication unit is implemented in an object-oriented manner, which hides the details from the students at the beginning of the learning process. Because our approach is based on the idea of representation independence, the actual implementation of the appropriate primitives may be seamlessly changed later.

The abstract terms, *i.e.*, the vocabulary of automata theory, are described in a precise manner using the high-level language Scheme. This makes the ideas and definitions concrete and explorable. The functional representation of an automaton, as well as each of its states, enables us to focus on the important ideas of state, transition, and nondeterminism rather than the mathematical syntax. Additionally, students are provided the invaluable and unusual opportunity of experimenting with very abstract concepts.

1 Introduction

Traditionally, an automaton is defined by giving a description of each of its components, that is, an automaton is defined to be a *k-tuple*, where k depends on the particular type of automata. Indeed, there is no serious reason to choose a tuple or a list to represent a unique automaton object. The imposed ordering of the elements in a tuple, and the possibility of duplicates, are mathematical constraints that provide mechanisms for building composite objects. A practical disadvantage of this definition is that nobody is really able to remember the chosen sequence of the items in a 7-tuple. Martin [3, p.70] presents a brief look at the use of mathematical tuples to define automata.

*Supported by the Deutsche Forschungsgemeinschaft (DFG) and the HTWS Zittau/Görlitz – sabbatical from 09/19/95 through 02/17/96. wagenkn@inf-gr.htw-zittau.de
†friedman@cs.indiana.edu

The definition of an extended transition function is supplemented by a very informal description of how the particular kind of automaton operates and could accept a certain set of inputs. Following the traditional mathematical culture, Kelley [2] and other authors replace those informal descriptions by formal notations. This extends the definition significantly and leads to a more cumbersome formulation.

The main objective of this paper is to demonstrate how a convenient programming language, like Scheme, can be used to define very abstract terms with a function-oriented implementation, where programming means describing much more than code generation. This approach is based mainly on the idea of *learning by programming*. A Scheme representation gives both a clear definition of the terms and an opportunity to execute and manipulate the defined object by, for instance, applying it to different input or passing it to other procedures.

This paper is divided into five sections. After this introduction, section 2 conveys the main idea of procedural acceptor's representation. The proposed implementation is similar to the diagram representation that is usually used to illustrate the possible transitions between the states of an automaton. There is no doubt that such diagrams contribute something towards understanding how the automaton operates, but they do not completely capture its behavior. So we introduce a communication unit that provides a high-level communication with an object that might be imagined as an instance of a particular abstract data type or of a class with regard to the specialized type of automaton. An object-oriented implementation of all types of communication units is presented in section 3. To present readable code we prepared some procedures shown in the appendix. Section 4 contains several examples illustrating the application of the introduced Scheme representation strategy to the different types of automaton. We conclude with a brief reflection of the pedagogic value of our approach.

2 What is an automaton?

2.1 The transition system

The most important terms in the theory of automata are *state* and *transition*, both of which have the meaning one intuitively expects. It is necessary, however, to define these terms formally.

A *state* of a system is a description of the current reality of that system. Imagine it as a snapshot of the system at some definite point in time. *Transition*s are changes of state that can happen spontaneously or in response to an external input on a tape. The network of transitions, also called a *transition system*, can be visualized by a *transition graph* as illustrated in figure 1.

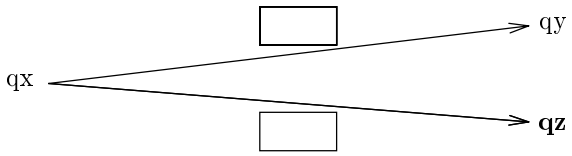


Figure 1: Transition graph

If the automaton shown in figure 1 is in state qx, then the next state will be either qy or qz. The state qz is emphasized in bold because it belongs to the set of *final states* for this system. To fill in the rectangular tags on the arrows, we must first specify the type of automaton. We discuss different types of automata below when we introduce the *communication unit*. This unit is a data object t, *i.e.*, some kind of tape-object, and it defines the interactions of the control with the contents of the tape in terms of procedures. These interactions correspond to the tags on the

arrows in the transition graph and they determine the particular type of communication unit to be used.

Here is a Scheme implementation of the transition system visualized in figure 1.

```
(define qx
  (lambda (t)
    (case (at t)
      ((...) (reconfig! t) (qy t))
      ((...) (reconfig! t) (qz t))
      (else #f))))
```

After reading the current symbol on the tape t, with $(at\ t)$, the tape must be reconfigured, with $(reconfig!\ t)$, and then the transition process continues with either qy or qz. Because qx is not a final state, the return value is #f. (This does not imply that the return value is always #t in the case of a final state.)

The main idea of this implementation is that each state of an automaton is a procedure that takes the object t and either returns #t or #f, or effects the transition to the next state. Thus, a transition is implemented as a procedure invocation.

An automaton is implemented with a procedure that encodes the entire transition system and that specifies the communication unit. This procedure expects the name of the starting state as an argument and returns a procedure of one argument to receive the input. The tape-object t will be initialized with this input.

We limit our discussion to acceptors. We implement a procedure that upon termination returns #t if the input is accepted by the automaton, and #f otherwise (see figure 2). The syntax of the input that is tested for acceptance is explained in subsection 2.2.

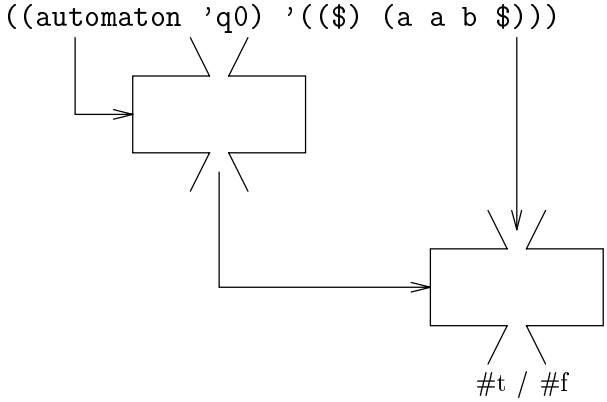


Figure 2: Run an acceptor on the input *aab*

Passing the name of the start state as an argument yields a very convenient representation for nondeterministic automata as well. A skeleton of the automaton's defined procedure might look like this:

```
(define automaton
  (lambda (start)
    (eval
      `(letrec                          ; Note the quasiquote mark!
         ((q0 (lambda (t) ...))
          (q1 (lambda (t) ...))
                :
                :
          (qn (lambda (t) ...)))
         (let ((t <tape-type>))
           (lambda (input)
             (init! t input)
             (eval (list ,start t))))))))    ; Note the comma!
```

A more convenient invocation that is equivalent to the application in figure 1 is (*run automaton* 'q0 '(($) (a a b $))), see the appendix.

2.2 The communication unit

During each transition, an interaction, *i.e.*, message passing occurs, with a private communication unit. You can think about it as an abstract data type. A particular tape-object t is an instance that understands the following four messages:

init! initializes t

at returns a copy of the appropriate item from t

reconfig! reconfigures t

contents returns the current value of t (not only an item)

The tape-object, usually referred to as tape, has *unbounded size*, meaning potentially infinite or, informally "as long as needed" to the left and to the right. There are three kinds of communication units:

1. read-only-tape
2. read-write-tape
3. read-only-tape-with-mp-stack

A tape is divided into consecutive *cells*. Each cell is preassigned with a blank, usually written as $. There is a *head* that has read-only or read-write capability, depending on the kind of communication unit. The capability of the head is indicated in the name of the object.

Following a suggestion of Springer and Friedman [4, p.375], a *configuration*, for example as given in figure 3, is internally represented by the list ((a y x $) (b $)), see section 3 for implementation details. Note that the sequence of cell contents is reversed for those to the left of the tape head.

Usually, we will initialize an automaton with an empty left sequence, for example, (($) (a a b $)) represents the input *aab* on the tape at the beginning as in figure 2.

The internal representation provides an efficient invocation, but it is not convenient for human beings interacting with the system. For that reason the external representation of the input used to

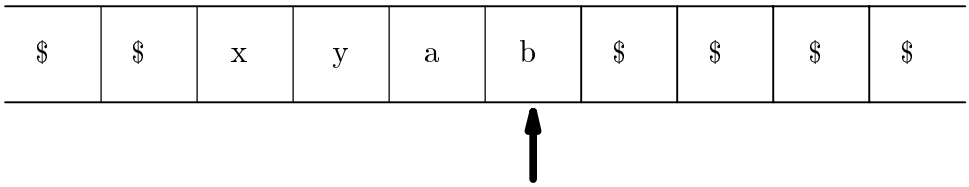


Figure 3: Tape configuation according ((a y x $) (b $))

initialize the tape corresponds to the real sequence. As explained below for the example in figure 3, the instruction is (*init! t* '(($ x y a) (b $))), which is somewhat easier to read.

The read-only-tape-with-mp-stack possesses a **m**ulti**p**ush stack. There is a procedure to push an entire list on the stack so that the first item of the list is on the top on the stack.

We provide an object-oriented implementation for the communication units in section 3. It is not necessary to understand the details of the implementation in order to use the package. Simply access the code from `http://www.inf-gr.htw-zittau.de/~wagenkn`, load the implementation file into Scheme, (*load* "c-unit.ss"), and the system is ready. To create an instance of a particular class, invoke the appropriate zero-argument procedure, for example (*read-only-tape*). Thus <*tape-type*> in the Scheme program *automaton* in subsection 2.1 can be replaced with (*read-only-tape*) to indicate the desired type of communication unit. This must be done for each new definition of an automaton since the type of the communication unit determines what the automaton is called.

Similarly, the associated procedures have different meanings. To avoid a lot of formal notation, we will describe the arguments and procedures with examples. Of course, the accompanying Scheme code provides precise definitions of their meaning.

Finite state automaton – fa

Communication unit: read-only-tape

(*init! t* '((\$) (h e l l o \$))) writes the input *hello* on the tape *t*, so that each character is associated with a separate cell. The head points at the first character, *h*.

(*at t*) returns the character in the cell pointed to by the head. There is no head movement.

(*reconfig! t*) Moves the head exactly one cell to the right.

(*contents t*) returns a list of two lists, the first represents the left part of the tape and the second represents the right part of the tape, with the head positioned on the first character of the second list.

Turing machines – tm

Communication unit: read-write-tape

(*init! t* '((\$) (h e l l o \$))) writes the input *hello* on the tape *t*, so that each character is associated with a separate cell. The head points at the first character, *h*.

(*at t*) returns the character in the cell pointed to by the head. There is no head movement.

(*reconfig! t* 'c 'left) writes a *c* on the current cell and move the head exactly one cell to the left. There is also a movement to the right by replacing 'left by 'right.

(*contents t*) returns a list of two lists, the first represents the left part of the tape and the second represents the right part of the tape, with the head positioned on the first character of the second list.

Pushdown automaton – pda

Communication unit: read-only-tape-with-mp-stack

(*init! t* '(((\$) (h e l l o \$)) (\$)) writes the input *hello* on the tape, so that each character is associated with a separate cell. The head points at the first character, *h*. Moreover, the stack is initialized with (\$), meaning the top of the stack is \$.

(*at t*) returns a list (h \$) of the value in the current cell and the top of stack. The top of stack is popped! There is no head movement.

(*reconfig! t* 'tape '(A B C)) moves the head of the tape exactly one cell to the right and multi-pushes the possibly-empty word, here ABC, onto the mp-stack. The new top of the stack is the first character of the word, here A. If 'tape is omitted, then no head movement occurs. This is a spontaneous transition.

(*contents t*) returns a list of two lists. The first list represents the tape contents (in the usual manner as a list of two lists) and the second represents the stack contents.

Now we have a well-defined communication unit and, therefore, the term automaton is precisely defined. We next describe what it means for an automaton to accept a particular input. Recall that every automaton begins with an initalized communication unit. An *fa* accepts an input if it enters a final state and the head reaches the first blank immediately after the right end of the input. A *tm* accepts an input if it enters a final state, regardless of the current contents of the tape. A *pda* accepts an input if it enters a final state and the head reaches the first blank immediatly after the right end of the input, regardless of the contents of the mp-stack.

2.3 Nondeterministic acceptors

Nondeterministic means that there may be more than one possible next state on a particular input symbol. Which transition should be used?

Thinking about nondeterminism of a transition machine means:

1. Create new machines, one for each possible next state, by cloning the current automaton. This includes copying the communication unit with its current content!
2. Each possible next state becomes the start state of one of these derived automata.
3. The input is said to be accepted if *one* of these automata returns #t. Otherwise, if they fail, the input is not accepted.

With this characterization we have eliminated the need to cover backtracking strategies, walks through trees and so forth. Henceforth, we use the names *nfa*, *ntm*, and *npda* to refer to nondeterministic machines. To distinguish the deterministic machines, we use *dfa*, *dtm*, and *dpda*.

Consider the following portion of a transition graph for an nfa.

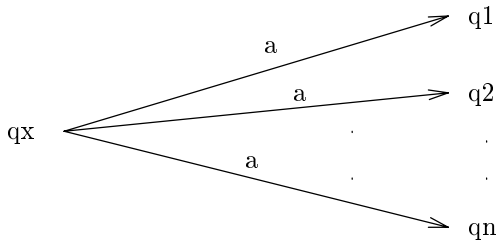


Figure 4: Portion of a transition graph – nfa

Here is a Scheme representation of the above graph.

```
(define nfa
  ...
  ((qx (lambda (tape)
         (case (at tape)
           ((a) (reconfig! tape)
                (or (clone-and-run nfa 'q1 tape)
                    (clone-and-run nfa 'q2 tape)
                        :
                    (clone-and-run nfa 'qn tape) ...)
```

Recall that the **or** operation in Scheme evaluates each operand in sequence until one evaluates to true (*i.e.*, non-#f), in which case the evaluation stops with that value. It is not even a "parallel-or" implemenation. If the operands evaluate to #f, then so does the **or** expression.

Obviously, the implementation of an nfa is easy. A bit more effort is required for ntm and npda, since the transitions involve changes on the tape and the stack, respectively. However, we simply follow the steps given above to generate the program.

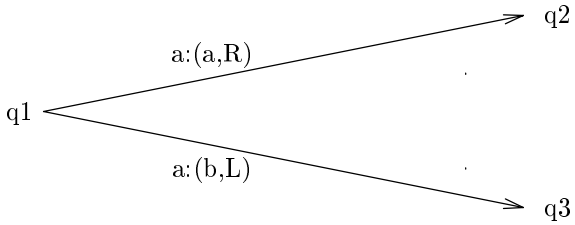


Figure 5: Portion of a transition graph – ntm

```
...
(q1
  (lambda (tape)
    (case (tape 'read)
      ((a) (or (let ((t (read-write-tape)))
                 (init! t (contents tape)) (reconfig! t 'a 'right) (clone-and-run ntm 'q2))
               (let ((t (read-write-tape)))
                 (init! t (contents tape)) (reconfig! t 'b 'left) (clone-and-run ntm 'q3)))))))
...
```

Next we present an extension of nondeterministic automata with ϵ-*transitions, i.e., spontaneous transitions* that ignore the tape. We will consider a part of an npda's transition graph and develop the corresponding Scheme implementation in a straightforward manner.

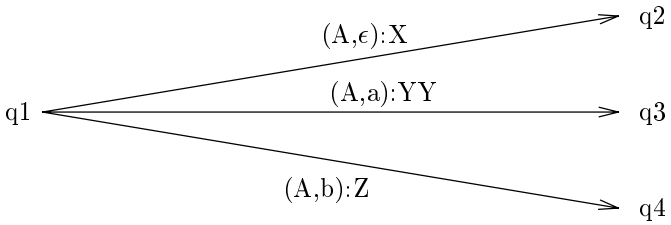


Figure 6: ϵ-transitions – npda

```
...
(q1
  (lambda (tape-stack)
    (let ((symbol-top (at tape-stack)))
      (case (cadr symbol-top)
        ((A) (or (let ((t-s (read-only-tape-with-mp-stack)))
                   (init! t-s (contents tape-stack))
                   (reconfig! t-s '(X))
                   (clone-and-run npda 'q2 t-s))
                 (case (car symbol-top)
                   ((a) (reconfig! tape-stack 'tape '(Y Y)) (q3 tape-stack))
                   ((b) (reconfig! tape-stack 'tape '(Z)) (q4 tape-stack)))))))))
...
```

There are some difficulties regarding ϵ-transitions. For simplicity imagine a bidirectional ϵ-transition between two states of an nfa. This could imply an infinite transition path. Observe that for all inputs that are not accepted, the automaton terminates and returns #f. This is significantly different from the situation where an automaton, on an unaccepted input, never stops and, thus, never returns any value.

This problem can be solved because for every nfa with ϵ-transitions there exists an equivalent nfa without ϵ-transitions. The main idea of this theorem could be expressed in a Scheme procedure that takes the original automaton as input and returns the equivalent ϵ-free one. We will not, however, use this approach here.

2.4 The Universal Turing Machine (UTM)

The left part of the tape is empty, meaning each cell is preassigned with blanks. The right part of the tape contains a word that represents the definition of a tm, followed by one blank, followed by an input to the tm.

The UTM scans the word defining the tm and stops after recognizing the intermediate blank at the cell that contains the first character of the input. Meanwhile, the word that defines the tm is overwritten with blanks. Thus, the configuration now, as usual, means the left part of the tape is empty, and the right part contains the input.

The *UTM simulates the control of the particular tm by interpreting the first scanned word and applying it to the input.*

We adopt some conventions to minimize the code:

1. The starting state of the simulated tm is generally called *start*.
2. The word describing the Turing machine is represented by a list of the form
   (**define** *tm* ...)

In figure 8, we reveal that the operation "read from a tape" is actually implemented using **car**. This leads to a convenient reading of sublists, assuming that the list is stored in one cell of the tape. Otherwise, if the list is stored character by character in a sequence of consecutive cells, the scanning process is somewhat more expensive. Two reconfigurations are needed since we assume that there is one blank cell between the list describing tm and the input of tm.

```
(define utm
  (lambda ()
    (let ((start (lambda (tape)
                   (eval (at tape))
                   (reconfig! tape '$ 'right)
                   (reconfig! tape '$ 'right)
                   (run tm 'start (contents tape)))))
      (let ((t (read-write-tape)))
        (lambda (input)
          (init! t input)
          (start t))))))
```

For example, consider the particular tm in figure 17. Following our conventions, we rename loopright to be start and busy-beaver to be tm. In this example the blank appears as the tm's input. If *abc* is the input, we invoke (*utm-run* '(($) ((define tm (lambda ...)) $ a b c $))).

```
> (run-utm '(($) ((define tm
                    (lambda (start)
                      (eval `(letrec
                               ((start (lambda (tape) ...)...)...)...)...)...)...)
                  $)))
(($ a a a a) (a a $))
#t
```

3 Implementation of communication units

The object-oriented implementation of the different communication units (described below), provides three important advantages:

1. The specification of automata is not required to make use of the implementation of the unit. We exploit the well-known technique of *data abstraction*.
2. The implementation illustrates the relations that exist between the different classes, communication units, and automata.
3. It is possible to change the implementation without needing to change the automaton's definition. It is a valuable exercise to investigate alternate implementations or different data structures, like streams, which would capture the unbounded length of the tape.

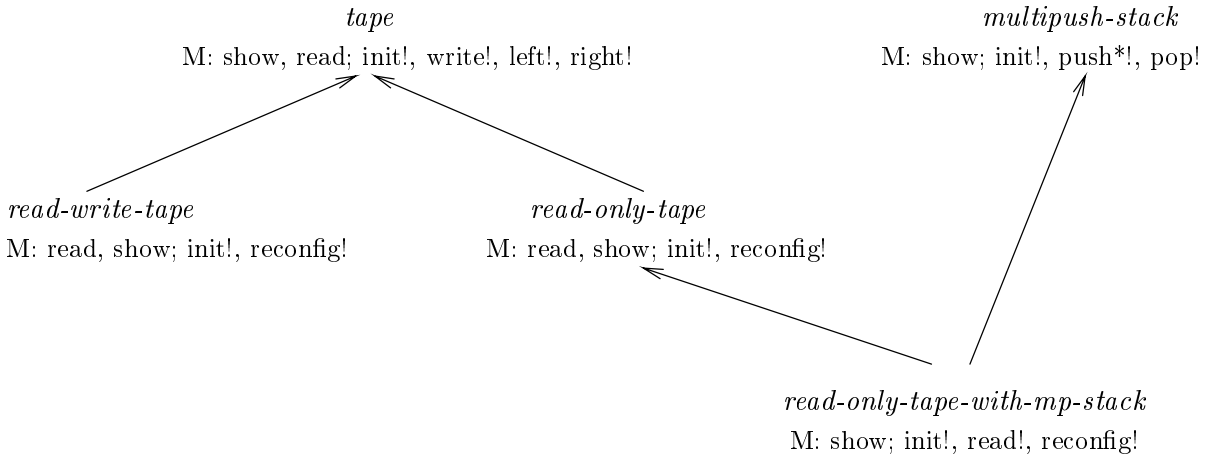


Figure 7: Class hierarchy

We use the *delegation* model in this object-oriented implementation. Delegation implies that a subclass has a private instance of a superclass's object responding to the forwarded messages. More details are presented in Friedman, Wand and Haynes [1], Springer and Friedman [4], and Wagenknecht [5]. For example, the class *read-write-tape* in figure 7 understands the message *read*. The method to be invoked when a *read* message is received is defined in *tape*. Thus a *read-write-tape* object contains a private instance of a *tape* class, that handles the *read* messages.

Moreover, delegation provides a convenient way to redefine methods inherited from a superclass, and a mechanism for deactivating inherited methods, as well.

Now we can reveal why we reverse (see init!, show) the character sequence in the left part of the tape, see figure 3. After a head movement one sublist loses an item, which the other sublist

```
(define tape
  (lambda ()
    (let ((blank '$))
      (let ((left-part (list blank))
            (right-part (list blank)))
        (lambda message
          (case (car message)
            ((init!)
              (set! left-part (reverse (caadr message)))
              (set! right-part (cadadr message)))
            ((write!)
              (if (equal? right-part (list blank))
                (set! right-part (cons (cadr message) right-part))
                (set! right-part (cons (cadr message) (cdr right-part)))))
            ((left!)
              (set! right-part (cons (car left-part) right-part))
              (if (not (equal? left-part (list blank)))
                (set! left-part (cdr left-part))))
            ((right!)
              (set! left-part (cons (car right-part) left-part))
              (if (not (equal? right-part (list blank)))
                (set! right-part (cdr right-part))))
            ((show)
              (list (reverse left-part) right-part))
            ((read)
              (car right-part))
            (else (error 'tape "Message ~a cannot be evaluated" (car message)))))))))
```

Figure 8: Definition of the class *tape*

gains. The natural Scheme procedure for inserting an item at the front of a list is **cons**, and, more importantly, the Scheme procedure to retrieve the first item from a list is **car**. In a list, the left-most item is the first one. So the left part of a tape is represented in the reverse order of how the items actually appear on the tape. Furthermore, it is easy to understand why we mark the right end of each list with the blank character $. If the head arrives at a cell where this blank character is stored, further moves to the left or to the right would generate as many blank characters as needed.

Here is a sample run.

```
> (define t (tape))
> (t 'init! '(($ h e l l o) (r o b $)))
> (t 'show)
(($ h e l l o) (r o b $))
> (t 'write! 'b)
> (t 'show)
(($ h e l l o) (b o b $))
> (t 'write 10)
```

```
Error in tape: Message write cannot be evaluated.
Type (debug) to enter the debugger.
> (t 'left!)
> (t 'show)
(($ h e l l) (o b o b $))
```

Two subclasses of *tape* are *read-write-tape* and *read-only-tape*. Certainly these definitions could be developed by the students themselves. We recommend that the reader unfamiliar with this approach to objects should create some instances and interact with them.

```
(define read-write-tape
  (lambda ()
    (let ((tape-obj (tape)))
      (lambda message
        (case (car message)
          ((reconfig!)
             (tape-obj 'write! (cadr message))
             (case (caddr message)
               ((right) (tape-obj 'right!))
               ((left) (tape-obj 'left!))))
          ((left! right! write!)
             (error 'message "~a is prohibited for read-write-tapes" (car message)))
          (else (apply tape-obj message)))))))
```

Figure 9: Definition of the subclass *read-write-tape*

```
> (define rwt1 (read-write-tape))
> (rwt1 'init! '(($) ($)))
> (rwt1 'show)
(($) ($))
> (rwt1 'right!)

Error in message: right! is prohibited for read-write-tapes.
Type (debug) to enter the debugger.
> (rwt1 'read)
$
> (rwt1 'reconfig! 'd 'right)
> (rwt1 'show)
(($ d) ($))
> (rwt1 'reconfig! 'c 'left)
> (rwt1 'show)
(($) (d c $))


> (define rot1 (read-only-tape))
> (rot1 'init! '(($) (a b $)))
> (rot1 'show)
```

```
(define read-only-tape
  (lambda ()
    (let ((tape-obj (tape)))
      (lambda message
        (case (car message)
          ((reconfig!) (tape-obj 'right!))
          ((left! right! write!)
             (error 'message "~a is prohibited for read-only-tapes" (car message)))
          (else (apply tape-obj message)))))))
```

Figure 10: Definition of the subclass *read-only-tape*

```
(($) (a b $))
> (rot1 'reconfig!)
> (rot1 'reconfig!)
> (rot1 'show)
(($ a b) ($))
```

For the implementation of a class *read-only-tape-with-mp-stack*, we need an *mp-stack* that looks like this:

```
(define mp-stack
  (lambda ()
    (let ((st '()))
      (lambda message
        (case (car message)
          ((init!)
             (set! st (cadr message)))
          ((show)
             st)
          ((push*!)
             (for-each
               (lambda (x) (set! st (cons x st)))
               (reverse (cadr message))))
          ((pop!)
             (if (null? st) (error 'pop! "Stack is empty"))
             (let ((answer (car st)))
               (set! st (cdr st))
               answer))
          (else (error 'tape "Message ~a cannot be evaluated" (car message)))))))))
```

Figure 11: Definition of the class *multipush-stack*

The dialog describing the interaction with a stack object is just slightly different from that with tapes given above.

```
> (define st1 (mp-stack))
```

```
> (st1 'init! '(a b c))
> (st1 'show)
(a b c)
> (st1 'pop!)
a
> (st1 'show)
(b c)
> (st1 'push*! '(x y z))
> (st1 'show)
(x y z b c)
```

Note that *push*!* takes a list as its argument and pushes the items of the list onto the stack starting with the **last** one. For readability, we write it as if were pushing the reversed list one item at a time.

```
(define read-only-tape-with-mp-stack
  (lambda ()
    (let ((tape-obj (read-only-tape))
          (stack-obj (mp-stack)))
      (lambda message
        (case (car message)
          ((init!) (tape-obj 'init! (caadr message))
                   (stack-obj 'init! (cadadr message)))
          ((show) (list (tape-obj 'show) (stack-obj 'show)))
          ((read) (list (tape-obj 'read) (stack-obj 'pop!)))
          ((reconfig!)
             (let ((second (cadr message)))
               (case second
                 ((tape) (tape-obj 'reconfig!)
                         (stack-obj 'push*! (caddr message)))
                 (else (stack-obj 'push*! second)))))
          (else
            (error 'message
              "~a is prohibited for read-only-tapes-with-mp-stack"
              (car message)))))))))
```

Figure 12: Definition of the class *read-only-tape-with-mp-stack*

```
> (define rotws (read-only-tape-with-mp-stack))
> (rotws 'init! '((($) (a b c $)) ($)))
> (rotws 'show)
((($) (a b c $)) ($))
> (rotws 'read)
(a $)
> (rotws 'read)

Error in pop!: Stack is empty.
```

```
Type (debug) to enter the debugger.

> (rotws 'show)
(((($) (a b c $)) ())
```

4 Teaching experience and pedagogical remarks

Following the intended approach of teaching formal languages and automata to undergraduates, three potential problems appear:

1. How can we motivate the students to deal with abstract ideas as discussed above?
2. How do we get students familiar with using Scheme as a tool to describe instances of these high-level terms.[1]
3. How can we maintain the dovetailed approach of teaching different types of formal languages in conjunction with the appropriate kinds of automata, while following the more general approach to understanding automata as suggested?

We now address the first two questions:

Earlier we started teaching theoretical concepts using Scheme without any kind of introduction to this programming language itself. Students were only familiar with C and C++, *i.e.*, they had no background in function-oriented programming. It turned out that almost all students mastered the one semester course on theory involving Scheme with positive feedback regarding the integration of the computer's use. Nevertheless it was hard work for students until they got good results towards the end of the semester. Consequently, the motivation to this pedagogical approach was not very sufficient.

Taking both these considerations together, we introduced Scheme at the beginning of the course in connection with treatments of the structure of a limited set of Scheme programs and their evaluation. Following the ideas published in Friedman, Wand and Haynes [5, pp. 83] we used Scheme to generate an abstract syntax tree relating to the concrete syntax of Scheme itself. This transformation is simple and does not require prerequisites in compiler design.

This approach makes certain that students get a general understanding of how compilers and interpreters work. We spend almost a quarter of the course on this part. While this is a considerable amount of time, it guarantees that the contents are taught effectively. We now address the third question.

Teaching formal languages in close connection with appropriate kinds of acceptors for each well-known type of formal grammar might come into pedagogical conflict with the more general issue of teaching automata as suggested in our approach. We decided to keep the traditional method of teaching but superimpose it on the new treatment of automata.

After the introduction of regular languages, dfa has to be taught. We did this with respect to the following aspects considered in the given sequence:

General information. An acceptor is an automaton that takes the input, writes it onto a tape-object (using *init!*), and calculates either *true* or *false* (meaning that the given input is

[1] We do not consider the Scheme programming language similar to Pascal, C or any other computer language used just to implement algorithms. Scheme is a function-oriented description tool that can be more precise, expressive and convenient than when using imperative computer programming languages, natural language or mathematical notation.

accepted or not). The calculation looks as follows: $q_0(t_a) = q_1(t_b) = ... = q_n(t_z) = true/false$, where q_i, the states of an automaton, are represented by procedures. Fundamentally, a procedure call on t_i generates a new procedure invokation on a modified tape t_j until `#t` or `#f` is finally returned. The contents of the tape are changed by side effects evoked by the procedure that was running.[2] A particular acceptor is completely defined by its transition system and the chosen type of communcation unit.

Experimenting with a dfa. The only thing we had to do additionally is to explain the syntax of *init!* and to point out that the dfa uses a read-only-tape. That's all! Students were experimenting with some different instances of these types of acceptors that are passed several inputs.

reconfig! is the key. Students figured out how *reconfig!* works in this particular case. Below figure 10 in section 3, an example shows how such an investigation might look. The understanding of *reconfig!* helps to get feedback on which kind of language a dfa is associated with.

Nondeterminism means cloning. The fundamental idea of thinking about nondeterminism as cloning is much more conceptional than as backtracking. Students understood very quickly that a deterministic automaton needs only a few modification of its transition system – but not of the communication unit – to be a nondeterministic machine.

Our students had no difficulties in representing acceptors of different types. They enjoyed running automata on several inputs and appreciated the semantical nearness between the conceptional ideas and their representations using the proposed skeletons to implement automata. Based on this experience most of the students were able to predict the whole relation between regular languages and finite automata.

Regarding pda that have to be taught later, we restricted the students to describe particular kinds of acceptors that cover context-free languages. As expected they could not find an answer immediately. However, after discussion of an additional stack memory they told us how they want the `reconfig!` method to work.

[2] It is important to take into consideration the difference between the traditional approach using a transition function $\delta(q, c)$ and a extended transition function $\hat{\delta}(q, w)$, where c is a character and w is a word.

5 Examples

Most of the examples come from Kelley [2]. The source code is available at `http://www.inf-gr.htw-zittau.de/~wagenkn`. One can follow the Scheme session scripts or experiment with the programs as much as is desired.

In order to get a printed transcript of the running automata, we inserted (*printf* `"~a~n~a~n"` (*contents tape*) 'qx), where *qx* stands for the name of the appropriate state.

5.1 Deterministic finite state automaton

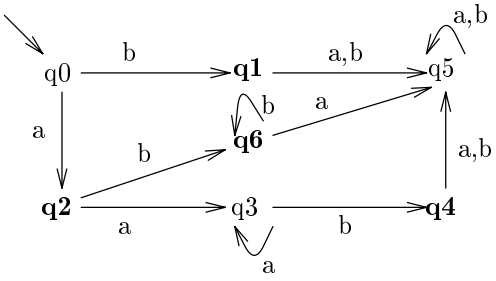


Figure 13: Deterministic finite state automata (accepts $a^*b + ab^*$)

```
> (run dfa01 'q0 '(($) (a a b $)))
(($) (a a b $))
q0
(($ a) (a b $))
q2
(($ a a) (b $))
q3
(($ a a b) ($))
q4
#t
```

```
> (run dfa01 'q0 '(($) (a a b b $)))
(($) (a a b b $))
q0
(($ a) (a b b $))
q2
(($ a a) (b b $))
q3
(($ a a b) (b $))
q4
(($ a a b b) ($))
q5
#f
```

```
(define dfa01
  (lambda (start)
    (eval
      `(letrec
         ((q0 (lambda (tape)      ; Start
                (printf "~a~n~a~n" (contents tape) 'q0)
                (case (at tape)
                  ((a) (reconfig! tape) (q2 tape))
                  ((b) (reconfig! tape) (q1 tape))
                  (else #f))))
          (q1 (lambda (tape)
                (printf "~a~n~a~n" (contents tape) 'q1)
                (case (at tape)
                  ((a) (reconfig! tape) (q5 tape))
                  ((b) (reconfig! tape) (q5 tape))
```

```
          (($) #t)
          (else #f))))
    (q2 (lambda (tape)
          (printf "~a~n~a~n" (contents tape) 'q2)
          (case (at tape)
            ((a) (reconfig! tape) (q3 tape))
            ((b) (reconfig! tape) (q6 tape))
            (($) #t)
            (else #f))))
    (q3 (lambda (tape)
          (printf "~a~n~a~n" (contents tape) 'q3)
          (case (at tape)
            ((a) (reconfig! tape) (q3 tape))
            ((b) (reconfig! tape) (q4 tape))
            (else #f))))
    (q4 (lambda (tape)
          (printf "~a~n~a~n" (contents tape) 'q4)
          (case (at tape)
            ((a) (reconfig! tape) (q5 tape))
            ((b) (reconfig! tape) (q5 tape))
            (($) #t)
            (else #f))))
    (q5 (lambda (tape)
          (printf "~a~n~a~n" (contents tape) 'q5)
          (case (at tape)
            ((a) (reconfig! tape) (q5 tape))
            ((b) (reconfig! tape) (q5 tape))
            (else #f))))
    (q6 (lambda (tape)
          (printf "~a~n~a~n" (contents tape) 'q6)
          (case (at tape)
            ((a) (reconfig! tape) (q5 tape))
            ((b) (reconfig! tape) (q6 tape))
            (($) #t)
            (else #f)))))
  (let ((t (read-only-tape)))
    (lambda (input)
      (init! t input)
      (eval (list ,start t)))))))
```

Recall that if the head is at a cell containing the blank symbol and the current state of the automaton is a final state, the return value is #t (true).

5.2 Nondeterministic finite state automaton

This automaton accepts the same language as the dfa in figure 13, but it is nondeterministic, without ϵ-transitions.

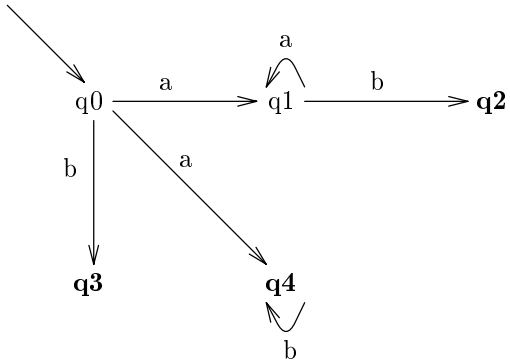


Figure 14: Nondeterministic finite state automata (accepts $a^*b + ab^*$)

```
(define nfa01
  (lambda (start)
    (eval
      `(letrec
         ((q0 (lambda (tape)      ; start
                (printf "~a~n~a~n" (contents tape) 'q0)
                (case (at tape)
                  ((a) (reconfig! tape) (or (clone-and-run nfa01 'q1 tape)
                                            (clone-and-run nfa01 'q4 tape)))
                  ((b) (reconfig! tape) (q3 tape))
                  (else #f))))
          (q1 (lambda (tape)
                (printf "~a~n~a~n" (contents tape) 'q1)
                (case (at tape)
                  ((a) (reconfig! tape) (q1 tape))
                  ((b) (reconfig! tape) (q2 tape))
                  (else #f))))
          (q2 (lambda (tape)
                (printf "~a~n~a~n" (contents tape) 'q2)
                (case (at tape)
                  (($) #t)
                  (else #f))))
          (q3 (lambda (tape)
                (printf "~a~n~a~n" (contents tape) 'q3)
                (case (at tape)
                  (($) #t)
                  (else #f))))
          (q4 (lambda (tape)
                (printf "~a~n~a~n" (contents tape) 'q4)
                (case (at tape)
```

```
                ((b) (reconfig! tape) (q4 tape))
                (($) #t)
                (else #f)))))
      (let ((t (read-only-tape)))
        (lambda (input)
          (init! t input)
          (eval (list ,start t))))))))
```

```
> (run nfa01 'q0 '(($) (a a a b b $)))
(($) (a a a b b $))
q0
(($ a) (a a b b $))
q1
(($ a a) (a b b $))
q1
(($ a a a) (b b $))
q1
(($ a a a b) (b $))
q2
(($ a) (a a b b $))
q4
#f
```

```
> (run nfa01 'q0 '(($) (a b b $)))
(($) (a b b $))
q0
(($ a) (b b $))
q1
(($ a b) (b $))
q2
(($ a) (b b $))
q4
(($ a b) (b $))
q4
(($ a b b) ($))
q4
#t
```

```
> (run nfa01 'q0 '(($) (a a a b $)))
(($) (a a a b $))
q0
(($ a) (a a b $))
q1
(($ a a) (a b $))
q1
(($ a a a) (b $))
q1
(($ a a a b) ($))
q2
#t
```

5.3 Nondeterministic finite state automaton with ϵ-transitions

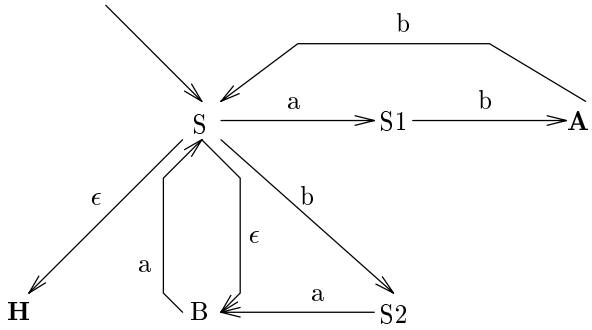


Figure 15: Nondeterministic finite state automaton with ϵ-transitions

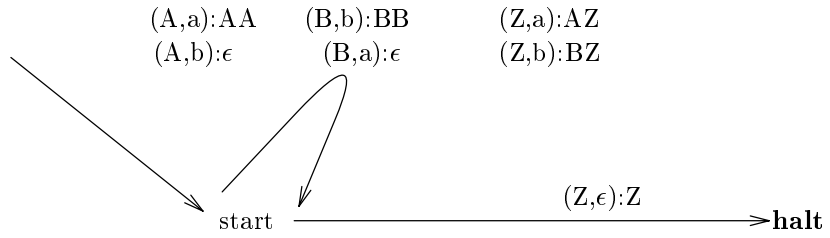
```
(define nfa02
  (lambda (start)
    (eval
      `(letrec
         ((S (lambda (tape)      ; start
               (printf "~a~n~a~n" (contents tape) 'S)
               (or (clone-and-run (nfa02 'H tape))
                   (clone-and-run (nfa02 'B tape))
                   (case (at tape)
                     ((a) (reconfig! tape) (S1 tape))
                     ((b) (reconfig! tape) (S2 tape))
                     (else #f)))))
          (S1 (lambda (tape)
                (printf "~a~n~a~n" (contents tape) 'S1)
                (case (at tape)
                  ((b) (reconfig! tape) (A tape))
                  (else #f))))
          (A (lambda (tape)
               (printf "~a~n~a~n" (contents tape) 'A)
               (case (at tape)
                 ((b) (reconfig! tape) (S tape))
                 (($) #t)
                 (else #f))))
          (S2 (lambda (tape)
                (printf "~a~n~a~n" (contents tape) 'S2)
                (case (at tape)
                  ((a) (reconfig! tape) (B tape))
                  (else #f))))
          (B (lambda (tape)
               (printf "~a~n~a~n" (contents tape) 'B)
               (case (at tape)
                 ((a) (reconfig! tape) (S tape))
                 (else #f))))
```

```
      (H (lambda (tape)
           (printf "~a~n~a~n" (contents tape) 'H)
           (case (at tape)
             (($) #t)
             (else #f)))))
    (let ((t (read-only-tape)))
      (lambda (input)
        (init! t input)
        (eval (list ,start t))))))))
```

```
> (run nfa02 'S '(($) (a b $)))
(($) (a b $))
s
(($) (a b $))
h
(($) (a b $))
b
(($ a) (b $))
s
(($ a) (b $))
h
(($ a) (b $))
b
(($ a b) ($))
s2
(($ a) (b $))
s1
(($ a b) ($))
a
#t
```

```
> (run nfa02 'S '(($) (a b b $)))
(($) (a b b $))
s
(($) (a b b $))
h
(($) (a b b $))
b
(($ a) (b b $))
s
(($ a) (b b $))
h
(($ a) (b b $))
b
(($ a b) (b $))
s2
(($ a) (b b $))
s1
(($ a b) (b $))
a
(($ a b b) ($))
s
(($ a b b) ($))
h
#t
```

5.4 Nondeterministic pushdown automaton with ϵ-transition



(t,s):i, where t – top of stack, s – current symbol (tape),
i – input to be multipushed onto the stack

Figure 16: npda with ϵ-transition

```
(define npda01
  (lambda (start)
    (eval
      `(letrec
         ((start              ; Start
            (lambda (tape-stack)
              (printf "~a~n~a~n" (contents tape-stack) 'start)
              (let ((symbol-top (at tape-stack)))
                (case (cadr symbol-top)    ; top of stack
                  ((A) (case (car symbol-top)
                         ((a) (reconfig! tape-stack 'tape '(A A)) (start tape-stack))
                         ((b) (reconfig! tape-stack 'tape '()) (start tape-stack))
                         (else #f)))
                  ((B) (case (car symbol-top)
                         ((a) (reconfig! tape-stack 'tape '()) (start tape-stack))
                         ((b) (reconfig! tape-stack 'tape '(B B)) (start tape-stack))
                         (else #f)))
                  ((Z) (or (let ((t-s (read-only-tape-with-mp-stack)))
                             (init! t-s (contents tape-stack))
                             (reconfig! t-s '(Z))
                             (clone-and-run npda01 'halt t-s))
                           (case (car symbol-top)
                             ((a) (reconfig! tape-stack 'tape '(A Z)) (start tape-stack))
                             ((b) (reconfig! tape-stack 'tape '(B Z)) (start tape-stack))
                             (else #f))))
                  (else #f)))))
          (halt
            (lambda (tape-stack)
              (printf "~a~n~a~n" (contents tape-stack) 'halt)
              (let ((symbol-top (at tape-stack)))
                (case (car symbol-top)
                  (($) #t)
                  (else #f))))))
         (let ((t (read-only-tape-with-mp-stack)))
```

```
            (lambda (input)
              (init! t input)
              (eval (list ,start t))))))))
> (run npda01 'start '((($) (a a a b b b $)) (Z)))
((($) (a a a b b b $)) (z))
start
((($) (a a a b b b $)) (z))
halt
((($ a) (a a b b b $)) (a z))
start
((($ a a) (a b b b $)) (a a z))
start
((($ a a a) (b b b $)) (a a a z))
start
((($ a a a b) (b b $)) (a a z))
start
((($ a a a b b) (b $)) (a z))
start
((($ a a a b b b) ($)) (z))
start
((($ a a a b b b) ($)) (z))
halt
#t

> (run npda01 'start '((($) (a b b $)) (Z)))
((($) (a b b $)) (z))
start
((($) (a b b $)) (z))
halt
((($ a) (b b $)) (a z))
start
((($ a b) (b $)) (z))
start
((($ a b) (b $)) (z))
halt
((($ a b b) ($)) (b z))
start
#f
```

5.5 Deterministic Turing machine

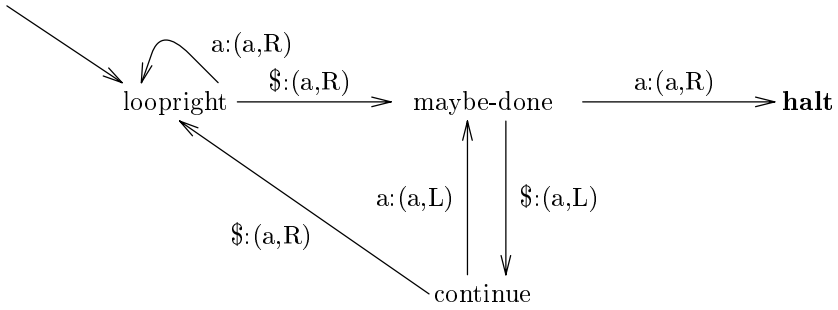


Figure 17: Deterministic Turing machine

```
(define busy-beaver
  (lambda (start)
    (eval
      `(letrec
         ((loopright (lambda (tape)
                       (printf "~a~n~a~n" (contents tape) 'loopright)
                       (case (at tape)
                         ((a) (reconfig! tape 'a 'right) (loopright tape))
                         (($) (reconfig! tape 'a 'right) (maybe-done tape))
                         (else #f))))
          (maybe-done (lambda (tape)
                        (printf "~a~n~a~n" (contents tape) 'maybe-done)
                        (case (at tape)
                          ((a) (reconfig! tape 'a 'right) (halt tape))
                          (($) (reconfig! tape 'a 'left) (continue tape))
                          (else #f))))
          (continue (lambda (tape)
                      (printf "~a~n~a~n" (contents tape) 'continue)
                      (case (at tape)
                        ((a) (reconfig! tape 'a 'left) (maybe-done tape))
                        (($) (reconfig! tape 'a 'right) (loopright tape))
                        (else #f))))
          (halt (lambda (tape)
                  (printf "~a~n~a~n" (contents tape) 'halt)
                  #t)))
         (let ((t (read-write-tape)))
           (lambda (input)
             (init! t input)
             (eval (list ,start t))))))))
```

```
> (run busy-beaver 'loopright '(($) ($)))
(($) ($))
loopright
(($ a) ($))
maybe-done
```

```
(($) (a a $))
continue
(($) ($ a a $))
maybe-done
(($) ($ a a a $))
continue
(($ a) (a a a $))
loopright
(($ a a) (a a $))
loopright
(($ a a a) (a $))
loopright
(($ a a a a) ($))
loopright
(($ a a a a a) ($))
maybe-done
(($ a a a a) (a a $))
continue
(($ a a a) (a a a $))
maybe-done
(($ a a a a) (a a $))
halt
#t
```

6 Conclusion

Many terms of the theory of formal languages and automata are constructive terms. We chose automata, in particular acceptors, as an example to illustrate the new approach of using a real programming language, like Scheme, to define these terms in a way that captures the control by functional representations. This approach yields significant advantages over the ususal mathematical definitions.

The proposed approach is general. We defined an automaton as a transition machine possessing a communication unit. To specify the particular type of automaton we have to select an appropriate communication unit.

Our approach is based on the idea of learning by programming. Representing an automaton with Scheme is one way to define an automaton that forces a precise description of its behavior when applied to an input. The idea differs from simply using software to simulate an automaton's behavior and then learning more about the automaton by modifying transition functions. Our approach is not a black box approach.

We evaluated our approach in real teaching situations. Generally, there was positive student feedback, observed in individual interviews with students and also evident in the discussions overheard between pairs of students working together at the same computer. Most of the time they spoke about the behavior of automata in detail. This is precisely what we had hoped for.

What we will do next is to evaluate Scheme representation strategies for regular expressions, automatic minimal dfa generation, and connections between automata and the appropriate kind of generative grammar in formal language theory.

Acknowledgements

The first author would like to express his gratitude to many people who have contributed to this paper. He would like to thank Suzanne Menzel and George Springer for reviewing the first version, aiding with its translation, and pointing out areas of potential misunderstandings. He has benefitted from conversations with Matthias Felleisen (Rice University) and Frank Pietschmann (HTWS Zittau/Görlitz) about fundamental relationships between mathematics and computer science. He would also like to express his appreciation to Chris Haynes for a discussion about object-oriented programming, and to Vikram Subramaniam for discussions, support, and suggestions. Moreover, he is grateful to the Computer Science Department at Indiana University for making his visit so enjoyable. He would like to thank Dorai Sitaram who provided the TEX style for SLATEX. Finally he would like to express his gratitude to the second author who inspired this paper with the words "Come into my office, I will show you a better way to do this."

References

[1] Friedman, Daniel P., Wand, Mitchell, and Haynes, Christopher T. *Essentials of Programming Languages*, Cambridge MA: MIT Press, and New York: McGraw-Hill, 1992.

[2] Kelley, Dean *Automata and Formal Languages: An Introduction*, Englewood Cliffs NJ: Prentice Hall, 1995.

[3] Martin, John C. *Introduction to Languages and the Theory of Computation*, New York: McGraw-Hill, 1991.

[4] Springer, George, Friedman, Daniel P. *Scheme and the Art of Programming*, Cambridge MA: MIT Press, and New York: McGraw-Hill, 1989.

[5] Wagenknecht, Christian *Rekursion: Ein didaktischer Zugang mit Funktionen*, Bonn: Dümmler, 1994.

Appendix

```
(define reconfig!
  (lambda ls
    (let ((first-part (list (car ls) (quote 'reconfig!))))
      (cond
        ((null? (cdr ls)) (eval first-part))
        ((null? (cddr ls)) (eval (append first-part (list `(quote ,(cadr ls))))))
      (else
        (eval (append first-part (list `(quote ,(cadr ls)) `(quote ,(caddr ls))))))))))

(define init!
  (lambda (tape input)
    (tape 'init! input)))

(define contents
  (lambda (tape)
    (tape 'show)))

(define at
  (lambda (tape)
    (tape 'read)))

(define run
  (lambda (automaton state-name input)
    ((automaton state-name) input)))

(define run-utm
  (lambda (input)
    ((utm) input)))

(define clone-and-run
  (lambda (automaton state-name tape)
    ((automaton state-name) (tape 'show))))
```