Functional Automata
Software Support for Formal Languages Courses

Marco T. Morazán
Seton Hall University
morazanm@shu.edu

Rosario Antunez
City College of New York
mrantunez@gmail.com

An introductory formal languages course exposes students to automata theory, grammars, constructive proofs, computability, and decidability. This exposure usually comes late in the undergraduate curriculum or early in the graduate curriculum. In either setting, programming-oriented students find these topics to be challenging or, in many cases, overwhelming and on the fringe of Computer Science. The existence of this perception is not completely absurd, because rarely do formal languages courses, unlike programming courses, offer the sufficient software infrastructure for students to experiment with their ideas and designs. This article describes a library, FSM, designed to provide students with the opportunity to experiment and test their designs using finite-state machines, regular expressions, regular grammars, pushdown automata, context-free grammars, Turing machines, and context-sensitive grammars. Students are able to implement and test machines and grammars of their own design before proceeding with a formal proof of correctness. That is, students can test their designs much like they do in a programming course. In addition, the library easily allows students to implement the algorithms they develop as part of the constructive proofs they write. Providing students with this ability ought to be a new trend in the formal languages classroom.

1 Introduction

An introductory formal languages course exposes students to automata theory, grammars, constructive proofs, computability, and decidability. This exposure usually comes late in the undergraduate curriculum and early in the graduate curriculum. In either setting, programming-oriented students find these topics to be challenging or, in many cases, overwhelming and on the fringe of Computer Science. The existence of this perception is not completely absurd, because students are asked to design machines and grammars and to prove that they are correct without being able to experiment with their ideas like they do in any programming course. Machines and grammars, however, are representations of programs. Thus, in essence, designing machines and grammars without being able to test them goes against the grain of what students have learned in programming courses. The same holds true when arguing that a problem is decidable. In our experience, the inability to implement finite-state machines and grammars results in inexperienced students submitting solutions that are incorrect even when the exercise is relatively simple. Moreover, only a subset of the brightest students are able to tackle complex problems.

Rarely do formal languages textbooks (e.g., [3, 5, 7]), unlike programming textbooks, offer any software infrastructure for students to experiment with their ideas and designs. The difficulties raised regarding the design and implementation of automatons and grammars also has an impact on how well students understand constructive proofs. A constructive proof, in essence, spells out an algorithm to build a machine or grammar. For example, the proof that nondeterministic finite-state machines and deterministic finite-state machines are equivalent and the proof that context-free languages are closed under union outline algorithms. A student's Computer Science training tells her that algorithms need to be implemented and tested. This is especially true when the student has an intuition of how an algorithm

Submitted to:
TFPIE 2014

ought to proceed, but is uncertain of all the details–the typical case when a student starts thinking about a constructive proof. Without implementation and testing, it is common for students to try to prove the correctness of a machine or grammar that is incorrect. This leads to their work being marked down by an instructor and, in turn, to frustration and apathy for the material. The reader can contrast this scenario with a programming course in which the student can experiment with an implementation. Textbooks that reference software support (e.g., [4]) fail to integrate the use of the software as a part of the textbook. Instead, it is left as a recommendation. More worrisome, however, is that the support software is not easily extendible by students to integrate the algorithms they develop as part of their own constructive proofs. In other words, students are required to design algorithms that are not to be implemented.

Given that it is reasonable for Computer Science students to be able to experiment with the algorithms described in their textbook and with the algorithms they develop, a decision must be made as to how to engage programming students. One solution, of course, is to tell students to implement the algorithms from scratch without providing any software support. Although possible, this approach is likely to be too time-consuming within the confines of one semester. A more reasonable approach is to provide students with a library (or a programming language) that allows them to quickly implement, experiment, and test the machines and the grammars they design and the algorithms they develop. This article describes a Racket library, FSM, designed to provide students with the opportunity to implement, to experiment with, and to test their designs using finite-state machines, regular expressions, regular grammars, pushdown automata, context-free grammars, Turing machines, and context-sensitive grammars much like they do in a programming course. In addition, the library easily allows students to experiment with and implement the algorithms they develop as part of the constructive proofs they themselves write. This ability ought to be a new trend in the formal languages classroom and in the development of software to support such courses. The article is organized as follows. Section 2 highlights the interface and the implementation of the library. Section 3 provides examples of how the library has been used in practice. Section 4 describes related work. Finally, Section 5 presents concluding remarks and directions for future work.

2 Interface and Implementation

The FSM library presents the user with a generic interface to construct and manipulate finite-state machines and grammars. We divide constructors for each into two categories: primitive constructors and transformers. Primitive constructors build a finite-state machine or a grammar from a formal description provided by the programmer. Transformers build a finite-state machine or a grammar from existing machines or grammars exploiting algorithms obtained from constructive proofs. We divide observers into three categories: accessors, applicators, and testers. Accessors return a specified component used to build a grammar or a finite-state machine. Applicators apply a given machine or grammar to a word. Finally, testers allow for machines and grammars to be tested with words provided by the programmer or with randomly generated words by the software.

2.1 Finite-State Machines

A finite-state machine is either:

1. Deterministic finite-state machine (dfa)
2. Nondeterministic finite-state machine (ndfa)
3. Pushdown automata (pda)

4. Turing machine (tm)
5. Combined Turing machine (ctm)[1].

Every finite-state machine, except a ctm, has the user explicitly provide a finite set of states (S), a tape alphabet (Σ), a starting state (s), a set of final states (F), and a set of transition rules (δ). The transition rules must describe a function for a dfa while simply a relation for the other machines. Finally, a pda also requires a stack alphabet (Γ). $|M|$ denotes the representation of the finite-state machine of type M. The corresponding primitive constructors for machines with explicit rules have the following signatures:

1. make-dfsa: $S\ \Sigma\ s\ F\ \delta \rightarrow |dfa|$
2. make-ndfsa: $S\ \Sigma\ s\ F\ \delta \rightarrow |ndfa|$
3. make-pda: $S\ \Sigma\ \Gamma\ s\ F\ \delta \rightarrow |pda|$
4. make-tm: $S\ \Sigma\ \delta\ s\ F \rightarrow |tm|$

Finite-state machines are represented as functions that dispatch to the appropriate observer given some input.

A ctm is not defined by providing a formal description. Instead, the user provides a ctm description (*ctmd*) and Σ. A ctm description uses other Turing machines as building blocks. In essence, a ctm represents an iterative algorithm with conditional branches and gotos that changes the state of the machine. To aid in the development of these machines, the library allows for branches to abstract over the symbol being read. A variable captures the read symbol and is used to branch to a ctm where it becomes a constant whose value, if needed, can be written to the tape[2]. A ctm description is either:

1. An empty sequence
2. The sequence of a Turing machine and a *ctmd*
3. The sequence of a *ctmd* and a *ctmd*
4. The sequence of a label and a *ctmd*
5. The sequence of a branch and a *ctmd*
6. The sequence of a goto and a *ctmd*
7. The sequence of a variable branch and a *ctmd*

The constructor signature for a ctm is: combine-tms: $ctmd\ \Sigma \rightarrow |ctm|$. A ctm is represented as a function that takes as input a word and that returns a Turing machine configuration (i.e., a state, a tape, and the position of the head on the tape). In this manner, $\texttt{ctm}_1$ and $\texttt{ctm}_2$ can be composed as the resulting tape and head position of $\texttt{ctm}_1$ is used to build the initial configuration for $\texttt{ctm}_2$.

The finite-state-machine transformers build new machines from existing machines, from grammars, or from a regular expression. These constructors implement algorithms from constructive proofs typically covered in an introductory formal languages course. They include:

(regexp->fsa *regexp***)** Transforms a regular expression into a finite-state automaton.

(ndfa->dfsa $|fsa|$**)** Transforms a ndfa into a dfa.

[1]In terms of computational power there is no difference between a Turing Machine and a combined Turing machine. The latter is simply an abstraction that simplifies the design of Turing machines built using existing Turing Machines.

[2]In programming languages, this is akin to β-conversion.

(rename-states-fsm (listof state) |*fsm*|**)** Renames the states of the given fsm such that the intersection of the new names and the given list of symbols is empty[3]. The given fsm cannot be a ctm. This function is useful when combining two machines requires that the intersection of the set of states of both machines be empty like, for example, creating a machine using closure under union.

(union-fsm |*fsm*| |*fsm*|**)** Builds an fsm that accepts the union of the languages of the two given fsms. The given fsms cannot be ctms.

(concat-fsa |*fsm*| |*fsm*|**)** Builds an fsm that accepts the concatenation of the languages of the two given fsms. The given fsms cannot be ctms.

(kleenestar-fsm |*fsm*|**)** Builds an fsm that accepts the Kleene star of the given fsm's language. The given fsm cannot be a ctm.

(complement-fsm |*fsm*|**)** Builds an fsm that accepts the complement of the language of the given fsm. The given fsm cannot be a pda nor a ctm.

(intersection-fsm |*fsm*| |*fsm*|**)** Builds an fsm that accepts the intersection of the languages of the two given fsms. The given fsm cannot be a pda nor a ctm.

(grammar->fsm |*grammar*|**)** Builds an fsm for the language of the given grammar.

The observers to return a given component of an fsm take as input an fsm and return the desired component. The interesting observers are:

(apply-fsm |*fsm*| **word** {**natnum**}**)** Runs the given fsm assuming the given word is on the tape and the head of the fsm is on position 0 of the tape. If the optional natural number is provided, the head starts at with the head at that position. The returned value is either 'accept or 'reject. The given fsm is simulated by performing a breadth-first search of all the possible paths it can take by consuming the given word.

(show-transitions-fsm |*fsm*| **word** {**natnum**}**)** Similar to apply-fsm, but returns the path followed by the fsm. For a nondeterministic fsm, it returns an empty path if the machine halts in a non-accepting configuration.

The testers for finite-state machines offer the user the ability to experiment with the machines they create. There are three testers provided:

(same-result-fsm? |*fsm*| |*fsm*| **word)** Determines if both of the given fsms produce the same result for the given word.

(test-equiv-fsm |*fsm*| |*fsm*| {**natnum**}**)** Determines if both of the given fsms produce the same result on 100 randomly generated words. If the optional natural number is provided, then that number of random tests are performed. The function returns true or a list of words for which the given machines produce a different result.

(test-fsm |*fsm*| {**natnum**}**)** Generates 100 (or the given optional number) random words and returns a list of the words with the results obtained from the given fsm.

[3] In programming languages, this is akin to α-conversion.

2.2 Grammars

A grammar is either:

1. Regular grammar (rg)
2. Context-free grammar (cfg)
3. Context-sensitive grammar (csg)

Every grammar is composed of a set of terminal and nonterminal symbols (V), a set of terminal symbols (Σ), a set of of production rules (R), and a starting nonterminal symbol S. The left hand side of production rules for an rg and a cfg must only have a single nonterminal. The left hand side of production rules for a csg must contain at least one nonterminal. The right hand side of a production rule for a rg must have either a terminal symbol, a terminal symbol followed by a nonterminal symbol, or ε (the empty string) if the left hand side is S. The left hand side of a production rule for a cfg and a csg may have an arbitrary number of members of V. The primitive constructors for grammars are:

1. make-rg: $V\ \Sigma\ R_{rg}$ symbol $\rightarrow |rg|$
2. make-cfg: $V\ \Sigma\ R_{cfg}$ symbol $\rightarrow |cfg|$
3. make-csg: $V\ \Sigma\ R_{csg}$ symbol $\rightarrow\ |csg|$

The transformers for grammars build new grammars from either existing grammars, existing finite-state machines, or an existing regular expression. They include:

(fsm−>grammar $|fsm|$**)** Transforms a finite-state machine into a grammar.

(grammar-rename-nts (listof symbol) $|grammar|$**)** Renames the nonterminals of the given grammar to symbols not contained in the given list. This function is useful when the intersection of the nonterminals of two grammars must be empty.

The observers to return a given component of a grammar take as input a grammar and return the desired component. The interesting observers are:

(deriv $|grammar|$ **word)** If the given word is in the language of the given grammar, the derivation of the word is returned. Otherwise, a string stating that the word is not a member of the language of the grammar is returned.

The testers for grammars offer the user the ability to experiment with the grammars they create. There are three testers provided:

(both-deriv? $|grammar|$ $|grammar|$ **word)** Determines if both of the given grammars derive given word.

(test-equiv-grammar $|grammar|$ $|grammar|$ {**natnum**}**)** Determines if both of the given grammars produce the same result when attempting to derive 100 randomly generated words. If the the optional natural number is provided, then that number of random tests are performed. The function returns true or a list of words for which the given grammars produce a different result.

(test-grammar $|grammar|$ {**natnum**}**)** Generates 100 (or the given optional number) random words and returns a list of the words with the results obtained from trying to use the given grammar to derive them.

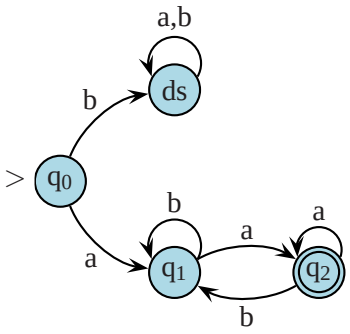

Figure 1: Buggy Finite-State Automaton for L.

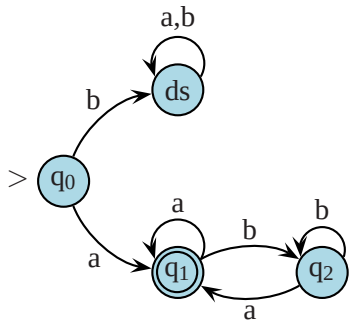

Figure 2: Correct Finite-State Automaton for L.

2.3 Regular Expressions

Let Σ be an alphabet of symbols. A regular expression (*re*) is a string that is either

1. ε (the empty string)
2. $a \in \Sigma$
3. $(re \cup re)$
4. $(re_1 re_2)$, where re_1 and re_2 are *re*
5. re^*

There is a constructor for each of the above. There is a single transformer, (fsa−>regexp fsa), that converts an fsa to a regular expression. There is a single observer, (printable-regexp r), that takes as input a regular expression and that returns a string representing the given regular expression.

3 The FSM Library in Practice

This section presents examples of the FSM library in practice. The examples chosen are problems that students have faced in an introductory formal languages course. These examples have been chosen to highlight how students and instructors can use the library to make its use relevant to both.

3.1 Write a Program to Recognize a Regular Language

Let $\Sigma = \{a, b\}$. Consider the problem of recognizing the regular language:

$L = \{w | w \in \Sigma^* \wedge$ *w starts and ends with an a*$\}$.

It is not uncommon for a student to submit the automaton in Figure 1. The automaton is clearly incorrect and the student is frustrated with the resulting poor grade. This frustration stems from realizing that the design has what they consider a small bug. A bug that experimentation can easily uncover assisting the student to refine their design.

If the student has access to the FSM library, an implementation of their design looks as follows:

```
(define sol1-dfa (make-dfsa  '(q0 q1 q2 ds)
                             '(a b)
                             'q0
                             '(q2)
                             '((q0 a q1)
```

```
                                (q0 b ds)
                                (q1 a q2)
                                (q1 b q1)
                                (q2 a q2)
                                (q2 b q1)
                                (ds a ds)
                                (ds b ds))))
```

The student can now test this machine as follows:

```
> (test-fsm sol1-dfa)
'(((b b a a b b b b a b a b b b b a b) reject)
  ((b a b a a a b a b a a a a a b b b a b) reject)
  ((a a b a b a a b a a b b b b) reject)
  ((a b a a b b a b b b a) accept)
  ((a b b a) accept)
  ...
  ((a) reject)
  ...
  ((b a a a b b b b a a b b b a b) reject)
  ((b a b a a b b b) reject))
```

The tests reveal that the word '(a) is rejected, but this word is an element of L. The student can now refine the solution to the one displayed in Figure 2. The corresponding implementation is:

```
(define sol1-dfa (make-dfsa  '(q0 q1 q2 ds)
                             '(a b)
                             'q0
                             '(q1)
                             '((q0 a q1)
                               (q0 b ds)
                               (q1 a q1)
                               (q1 b q2)
                               (q2 a q1)
                               (q2 b q2)
                               (ds a ds)
                               (ds b ds))))
```

3.2 The Reverse of a Regular Language

Proving that the reverse of a regular language, L_{rev}, is regular requires a constructive proof. This is a proof that students in an introductory formal languages course can be asked to write. Students are typically confused about how to tackle this kind of problem. They understand that they must show how to build a finite-state automaton, M_{rev}, for L_{rev}, but feel frustrated if their algorithm is not correct. In a programming course, students can implement and experiment with their proposed solution. The same ought to be true for a course in formal languages.

A common proposed solution is to build M_{rev} from a deterministic finite-state automaton, M, for L. Intuitively, M_{rev} has a new set of states containing the states of M and a new starting state, the same alphabet as M, the transition rules of M reversed, ε-transitions from the new starting state to each final

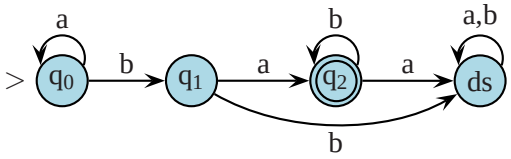

Figure 3: A finite-state automaton for $L = a^*bab^*$.

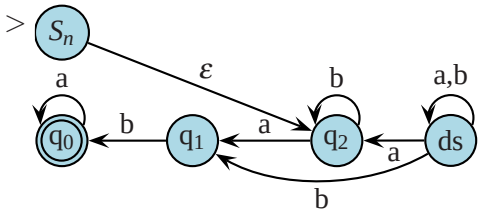

Figure 4: L_{rev} from Theorem Version I.

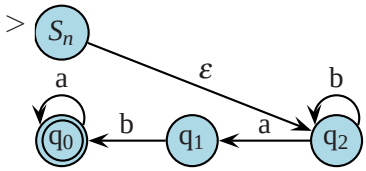

Figure 5: L_{rev} from Theorem Version II.

state of M, and the starting state of M as its only final state. More formally students define M_{rev} in their proposed version I of the theorem as follows:

$M_{rev} = (\{Q_M \cup \{S_{rev}\}\}, \Sigma_M, \delta_{rev}, S_{rev}, \{S_M\})$, where

- $Q_M =$ the states of M
- S_{rev} is the unique symbol for the starting state of M_{rev}
- $\Sigma_M =$ the alphabet of M
- δ_{rev} contains two types of rules:
  1. $(S_{rev}, \varepsilon, q_i)$ such that $q_i \in M_F =$ the final states of M
  2. (q_i, σ, q_j) such that $(q_j, \sigma, q_i) \in \delta_M$
- S_M is the starting state of M

The above algorithm can be implemented in FSM and tested by students. Consider applying the proposed theorem to a deterministic finite-state automaton for $L = a^*bab^*$ depicted in Figure 3. The resulting nfsa is depicted in Figure 4. Students can observe that the state *ds* is inaccessible and, therefore, ought not be part of the transformed machine. Furthermore, fewer states lead to a shorter formal proof. These observations suggest a refinement that eliminates the dead states of M and any transitions involving a dead state. Using this refinement, the resulting nfsa for M_{rev} is given in Figure 5. An FSM implementation of this second version of the proposed theorem is displayed in Figure 6. Once testing makes students confident that their solution is correct, they can proceed to write the formal proof.

Observe that new constructors are easily defined by students by simply writing a function. The code in Figure 6 only uses functions provided by FSM and list-processing functions. Thus, the coding of this constructor ought to be well within the reach of an advanced undergraduate Computer Science student. Students not familiar with higher-order functions need not use, for example, filter and map. Instead, they can write recursive functions to perform the necessary list-processing.

3.3 Determining if a Context-Free Language is Empty

Students also face exercises to prove that a problem is decidable. If a problem is decidable, then there is an algorithm to determine if an instance of a given problem fulfills stipulated conditions. Students can propose an algorithm for decidability, but without the ability to test their solution they remain unsure

```
;dfsa --> fsa
(define (reverse-fsa m1)

  ;symbol (listof rules)--> boolean
  (define (deadstate? s rules)
    (let  ((fromrules (filter (lambda (r) (eq? s (fsmrule-fromstate r)))
                              rules)))
           (andmap (lambda (r) (eq? s (fsmrule-tostate r))) fromrules)))

  (let* ((newfinal (fsm-getstart m1))
         (mstates (fsm-getstates m1))
         (newStart(gen-symbol 'S mstates))
         (deadsts (remove-duplicates
                    (filter (lambda (a) (deadstate? a (fsm-getdeltas m1)))
                            mstates)))
         (newQ (cons newStart (filter (lambda (q) (not (member q deadsts)))
                              mstates)))
         (addedrules (map (lambda (s) (list newStart EMP s))
                          (fsm-getfinals m1)))
         (changedrules
           (map reverse
                (filter (lambda (r) (and (not (member (fsmrule-tostate r)
                                deadsts))
                                         (not (member (fsmrule-fromstate r)
                                deadsts))))
                                (fsm-getdeltas m1)))
        (newrules (append addedrules changedrules)))
    (make-ndfsa newQ (fsm-getsigma m1) newStart (list newfinal) newrules)))
```

Figure 6: Proposed solution to build M_{rev}

about its validity. Testing their algorithm increases their confidence and brings their Computer Science education as programmers to bear in a formal languages course. We do not want students, however, to implement decidability algorithms using Turing machines. That would simply be too unwieldy. Instead, students ought to be able to use a library like FSM to implement and test their algorithms.

For example, consider the problem of deciding if the language, $L(G)$, of a context-free grammar is empty. Students realize that $L(G)$ is not empty if there exists a derivation for any word formed by elements of the alphabet of G. Usually, they must be guided to realize that what they need is an algorithm to detect the existence of a derivation by creating any backward derivation to G's starting symbol starting from the elements of the alphabet of G and ε. The algorithm accumulates the left hand sides of rules whose right hand side only contains symbols in the accumulator. If at any step, G's starting symbol is in the accumulator then $L(G)$ is not empty. If at any step, there are no new symbols to add to the accumulator then $L(G)$ is empty. Otherwise, the new symbols are added to the accumulator and the process recursively proceeds.

Figure 7 displays the implementation of this algorithm using the FSM library. Notice, that this code

```
; cfg --> boolean
(define (Lcfg-isempty? g)
  ; cfg-rule (listof symbol) --> (listof cfg-rule)
  (define (only-accum-elems? rule accum)
    (and (not (member (cfg-rule-lhs rule) accum))
         (andmap (lambda (s) (member s accum)) (cfg-rule-rhs rule))))
  ; (listof cfg-rules) nonterminal (listof symbol) --> boolean
  (define (isempty? rls S accum)
    (cond [(member S accum) #f]
          [else
           (let ((newmembers (map cfg-rule-lhs
                                  (filter (lambda (r)
                                            (only-accum-elems? r accum))
                                          rls))))
             (cond [(empty? newmembers) #t]
                   [else (isempty? rls S (append newmembers accum))]))]))
  (let ((rls (cfg-get-the-rules g))
        (S (cfg-get-start g))
        (sigma (cfg-get-alphabet g)))
    (isempty? rls S (cons EMP sigma))))
```

Figure 7: A Function to Determine if the Language of a CFG is Empty.

demonstrates that observers are easily added by simply writing a function. This observer only utilizes FSM provided functions and list-processing functions. Thus, once again, putting it well within the grasp of an advanced Computer Science undergraduate. Nonetheless, students tend to make mistakes at first and their algorithm requires refinement. The most common mistake is to not include ε in the initial value of the accumulator which is easily discovered once implemented.

3.4 Computing with Turing Machines

Although Turing machines are not the most attractive programming abstraction, it is important for students to understand their power and the reason the abstraction is less than attractive. The best way for students to begin to understand the power of Turing machines is to have then design Turing machines. Implementing formal descriptions of Turing machines in FSM is similar to developing the finite-state machines in Section 3.1. Such descriptions are best for language recognizers and operations that do not involve assignment (i.e., altering of the tape). For instance, the following Turing machine moves the head to the first blank to the right of the current position of the head.

```
(define RB (make-tm  '(s h)
                     '(I add1 sub1)
                      (list
                       (list (list 's 'I) (list 'h RIGHT))
                       (list (list 's 'add1) (list 'h RIGHT))
                       (list (list 's 'sub1) (list 'h RIGHT))
                       (list (list 's BLANK) (list 'h RIGHT)))
```

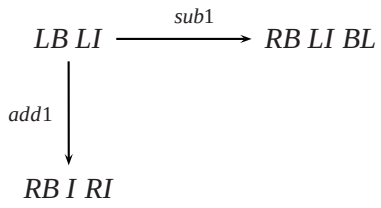

Figure 8: Draft Turing Machine to add or subtract 1 from a non-zero unary number.

```
's
'(h)))
```

Computations with Turing machines, however, may require assignment. Thus, requiring care during the development of Turing machines. In such a setting, offering abstractions is critical to keep students engaged. Many textbooks on formal languages develop a notation that is graphical and more transparent. In essence, the notation connects Turing machines, not states, using conditional branches and gotos. This allows for progressively more complex Turing machines to be designed from simpler Turing machines.

Consider the student having to add 1 or subtract 1 to a non-zero unary number. The first step is to state the precondition and the postcondition for the Turing Machine. For example, the algorithm can be designed assuming the machine starts in the following configuration: (op ⊔ *number* $\underline{⊔}$), where op is either `add1` or `sub1`, ⊔ denotes a blank space, and the head is on the first blank after the number. The machine stops in the following configuration: (op ⊔ *number* $\underline{⊔}$), where *number* is the result of the computation. Assume that in addition to RB above the following simpler machines are also defined:

LI Moves the head one space to the left.

RI Moves the head one space to the right.

I Writes I to the tape.

BL Writes ⊔ to the tape.

LB Moves the head to the first blank to the left of the head.

A first draft of the algorithm developed by a student is displayed in Figure 8. This algorithm moves the head to read the *op*. Then it branches depending on the *op*. After branching, it moves the head to the first blank to the right and proceeds to construct the resulting number. The FSM implementation is as follows:

```
(define addorsub (combine-tms (list LB LI (list BRANCH (list 'sub1 RB LI BL)
                                                        (list 'add1 RB I RI)))
                              '(I sub1 add1)))
```

This machine can be tested using apply-fsm to obtain the following results:

> (apply-fsm *addorsub* (list add1 _ I I I I _) 6)
(tmconfig 'h 2 '(add1 I I I I I _))
> (apply-fsm *addorsub* (list sub1 _ I I I I I I I _) 9)
(tmconfig 'h 0 '(_ _ I I I _))

A student quickly realizes that the postcondition of the machine is not met and, therefore, their design has a bug. Instead of handing in a buggy design and be marked down by the instructor, the student can now proceed to redesign their algorithm. The bug, of course, is that after the branch the machine must move to the second blank to the right. The resulting graphical notation is displayed in Figure 9. The FSM implementation is a s follows:

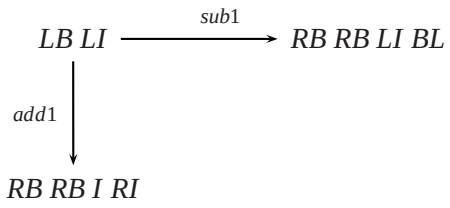

Figure 9: Turing Machine to add or subtract 1 from a non-zero unary number.

```
(define addorsub (combine-tms (list LB LI (list BRANCH (list 'sub1 RB RB LI BL)
                                                        (list 'add1 RB RB I RI)))
                              '(I sub1 add1)))
```

The tests are repeated and the student gets the following results:

> (apply-fsm *addorsub* (list add1 _ I I I I _) 6)
(tmconfig 'h 7 '(add1 _ I I I I I _))
> (apply-fsm *addorsub* (list sub1 _ I I I I I I I _) 9)
(tmconfig 'h 8 '(sub1 _ I I I I I I _ _))

With successful tests, the student can now proceed to develop arguments for correctness. This development establishes that having the ability to test Turing machines is an important part of the design process in a formal languages course.

4 Related Work

JFLAP was designed to experiment with finite-state machines and grammars as wells as to experiment with constructive proofs. JFLAP allows the user to create and simulate several types of finite-state machines, to create and parse strings in grammars, and to experiment with proof constructions such as converting a nondeterministic finite automaton to a deterministic finite automaton and then to a regular expression or regular grammar [6]. A study concluded that students felt more engaged and enjoyed a formal languages course more when using JFLAP [6]. JFLAP provides all the primitive and transformation constructors found in FSM. In contrast, however, the graphical nature of JFLAP does not offer the ability to easily add new observers and constructors to the software.

The jFAST library is designed to help beginners design finite-state machines [8]. It uses a graphical interface just as JFLAP, but unlike JFLAP it provides no functionality for regular expressions and grammars. Like JFLAP and, in contrast to FSM, there is not support for students to add the constructive algorithms they develop and prove. The FSA Simulator allows the user to work and experiment with finite-state automata offering the ability to compare the languages of two finite-state automatons [2]. That is, it provides testing facilities for the equivalence of two finite-state automatons as FSM. The FSA comparison feature lets the software give students feedback about the accuracy of their work much as intended by FSM for all types of finite-state machines. RegeXeX is an interactive system to write regular expressions [1]. In contrast to FSM, it provides testing facilities for regular expressions. The feedback provided by RegeXeX includes strings that ought to be accepted and ought to be rejected much like FSM provides testing facilities for finite-state machines.

5 Conclusions and Future Work

The FSM library provides users the necessary facilities to design and experiment with finite-state machines and grammars. It supports the view that the existence of a machine or grammar is proven by developing a constructive proof. This means that the proof presents an algorithm that can be implemented using FSM. Students no longer have to rely solely on paper-and-pencil traces to build confidence or discover bugs in their designs. This leads students to actively reason and learn about formal languages as well as to reinforce their Computer Science education by implementing and developing unit tests for their constructive algorithms. The library has received positive feedback by students and has provided the examples presented in the article.

Future work includes expanding the library to include more constructors particularly those for state minimization. We will also be expanding the library to include a graphical interface. Unlike the interfaces described in the related work, we do not wish to have students create machines and grammars using a graphical interface. Instead, our goal is to have students continue to write code to create machines and grammars that are then rendered using graphics to animate execution and visualize their structure. Finally, we will expand support for regular expressions and extensions to Turing machines. The latter machines, although not computationally more powerful than a standard Turing Machine, are likely to make certain designs easier to implement by students.

References

[1] Christopher W. Brown & Eric A. Hardisty (2007): *RegeXeX: an interactive system providing regular expression exercises*. In: *SIGCSE '07: Proceedinds of the 38th SIGCSE technical symposium on Computer science education*, ACM Press, New York, NY, USA, pp. 445–449, doi:http://doi.acm.org/10.1145/1227310.1227462.

[2] Michael T. Grinder (2003): *A Preliminary Empirical Evaluation of the Effectiveness of a Finite State Automaton Animator*. *SIGCSE Bull.* 35(1), pp. 157–161, doi:10.1145/792548.611958. Available at http://doi.acm.org/10.1145/792548.611958.

[3] Harry R. Lewis & Christos H. Papadimitriou (1997): *Elements of the Theory of Computation*, 2nd edition. Prentice Hall PTR, Upper Saddle River, NJ, USA.

[4] P. Linz (2012): *An Introduction to Formal Languages and Automata*, 5th edition. Jones & Bartlett Learning.

[5] J.C. Martin (2003): *Introduction to Languages and the Theory of Computation*. McGraw-Hill Series in Computer Science, McGraw-Hill.

[6] Susan H. Rodger, Eric Wiebe, Kyung Min Lee, Chris Morgan, Kareem Omar & Jonathan Su (2009): *Increasing Engagement in Automata Theory with JFLAP*. *SIGCSE Bull.* 41(1), pp. 403–407, doi:10.1145/1539024.1509011. Available at http://doi.acm.org/10.1145/1539024.1509011.

[7] Michael Sipser (2013): *Introduction to the Theory of Computation*, 3rd edition. Cengage Learning.

[8] Timothy M. White & Thomas P. Way (2006): *jfast: A java finite automata simulator*. In: *In Thirty-seventh SIGCSE Technical Symposium on Computer Science Education*, pp. 384–388.