Exploring Female and Male Student Perceptions in a Functional-Programming-Based Automata Theory Course

Marco T. Morazán, Tijana Minić, and Andrés M. Garced

Seton Hall University, USA {morazanm|minictij|maldona2}@shu.edu

Abstract. Historically, it has been challenging for students to be interested in and for instructors to teach Formal Languages and Automata Theory courses. The challenges stem from a multitude of reasons including the theoretical nature of the material, the use of formal notation, student lack of experience with problem solving and proof development, and the student perception that the material is not relevant to majoring in Computer Science. In this article, we illustrate our novel design- and programming-based approach to building state machines and developing constructive proofs. Student perceptions, both overall and by gender, are explored to determine if this approach effectively addresses some of the negative historical trends. Our empirical results suggest that the approach has a positive impact overall and by gender. In addition, the results reveal nuanced differences between female and male students and areas for future improvements.

1 Introduction

Historically, among Computer Science students, it has been difficult to motivate interest in Formal Languages and Automata Theory (`FLAT`) courses. The lack of interest is due to a plethora of reasons that include: student perceptions of no association with other courses in the Computer Science curriculum [7,24,50,54], student difficulty to understand abstract mathematical topics [7,19,65], student difficulty to understand formal notation [5], lack of problem-solving skills [55,65], and lack of proof-development skills [7,54]. Most of these reasons are likely to resonate with `FLAT` instructors that need to deliver a Computer Science theory course to an audience trained to program and with an appetite for software development. These are all general problems in the `FLAT` student population that have led to high drop-out rates [30,50,63]. At a more human level, readers of this article are likely to recall multiple occasions when they have witnessed the enthusiasm in students when they get a program to run. The same readers are less likely to have witnessed the same enthusiasm when a student gets a proof right. We are trying to get students enthusiastic about `FLAT` without sacrificing content nor rigor.

In addition to the problems above, many female students have further compounding problems in Computer Science that include: the perception that "hacking" abilities are a required prerequisite skill in Computer Science [14,31], the

perception that male students know much more [14,31], the need to survive the "boys' club" [14,31], a stronger need to connect computing to other fields and to see the computer as a tool to use in a broader context [14,31], lower self-confidence [6,14,22,25,31,62], and the perception that Computer Science is narrowly focused on programming [14]. Many of these problems come from a male-dominated discipline that, indifferently or unconsciously, perpetuates stereotypes about persons in Computer Science. By listing the problems that tend to affect female students more, we are not suggesting that female students perform worse than male students in Computer Science. Studies have suggested, for example, that there are no differences between female and male students in programming performance [26,34].

In the literature, there is a clear emphasis on practices that discourage Computer Science students with many articles focusing on the practices that discourage women. In this article, we break away from the from this norm and try to shed light on practices that encourage students, including women, in a new programming-based `FLAT` curriculum tightly-coupled with `FSM`–a functional programming language for the `FLAT` classroom. We present data collected through anonymous surveys at two universities, Seton Hall University (`SHU`) and Worcester Polytechnic Institute (`WPI`), using the programming-based `FLAT` curriculum put forth in *Programming-Based Formal Languages and Automata Theory: Design, Implement, Validate, and Prove.* [40]. We analyze the collected data in aggregate to capture overall perceptions across our sample population as well as by gender[1] to determine any differences. Our focus is on the following research questions:

- RQ1 Do students feel that programming state machines is straightforward?
- RQ2 Do students feel that Automata Theory, programming state machines, and programming constructive algorithms is intellectually stimulating?
- RQ3 Do students feel that Automata Theory is relevant to their Computer Science education?
- RQ4 Is programming in `FSM` useful to understand Automata Theory and to develop constructive proofs?

The article is organized as follows. Section 2 presents a brief introduction to `FSM`, the design-based programming methodology used to instruct students, and examples of machine and constructive algorithm implementations. Section 3 presents the collected empirical data. Section 4 compares and contrasts with related work. Section 5 presents a discussion of the results. Finally, Section 6 presents concluding remarks and directions for future work.

2 Programming in `FSM`

To illustrate the design methodology and the style used in class instruction, two examples are presented. The first is the design and implementation of a `dfa`. The

[1] In this article, the term gender refers to female and male, given that our sample population only contains two individuals that do not self-identify as such.

second is the implementation of an algorithm developed as part of a constructive proof. Specifically, we outline the design and implementation of a construction algorithm developed to prove the closure of regular languages under Kleene star (e.g., as discussed in [27,29,40,56,60]).

2.1 The Language

`FSM` [41] is a domain-specific language embedded in Racket [16]. It provides constructors for state machines: deterministic finite-state automata (`dfa`), nondeterministic finite-state automata (`ndfa`), pushdown automata (`pda`), Turing machines (`tms`), composed Turing machines (`ctms`) [35], and multitape Turing machines (`mttms`)[2]. For instance, the constructor for deterministic finite-state automata has the following signature:

`make-dfa: S` Σ `s F` δ `['no-dead]` $\rightarrow$ `dfa`

`S` denotes the set of states. Σ denotes the input alphabet. The starting state is denoted by `s`. `F` denotes a set of final states. The transition function is denoted by δ. Finally, the optional argument, `'no-dead`, informs the constructor that the transition function is fully specified. When this argument is omitted, the constructor adds a dead/trap state and adds transitions to this state to fully specify the transition function. Constructors for other machines are similar in nature. Readers may consult the `FSM` documentation for all constructors [39].

The language also provides observers to extract the components of a machine, observers to apply a machine to a word, unit testing facilities using `RackUnit` [66], and random testing facilities. In addition, integrated into `FSM` is a powerful suite of static and dynamic visualization tools. On the static side, `FSM` offers the automatic generation of transition diagrams [48] and computation graphs [44]. On the dynamic side, `FSM` offers tools to trace machine application [48] and to trace the conversion between computation models (e.g., transforming to/from an `ndfa` to a regular expression [45] and transforming an `ndfa` to a `dfa` [46]). The required graphics are made visually appealing by using `Graphviz` [17,18] to generate them.

Finally, `FSM` offers tailor-made error-messaging system that associates misuse of machine constructors with an informative error messages [43,9]. The error messaging system is tightly-coupled with the design-based methodology put forth by Morazán [40]. Each error message includes the step of the design recipe that has not been successfully completed along with a short, nonprescriptive, and jargon-free problem description. The error messages use the same vocabulary used in the textbook of instruction [40] and that instructors are encouraged to use in class. We coin such errors as *recipe-based errors*[3] [9].

1. Name the machine, specify the alphabets, and formulate, if applicable, the pre and post conditions.
2. Write unit tests
3. Identify conditions that must be tracked as input is consumed, associate a state with each condition, and determine the start and final states.
4. Formulate the transition relation
5. Implement the machine
6. Run the tests and, if necessary, redesign
7. Design, implement, and test an invariant predicate for each state
8. Prove L = L(M)

Fig. 1: The design recipe for state machines.

2.2 Design-Based Programming Methodology

The design methodology put forth by Morazán uses design recipes [40]. A design recipe is a series of steps, each with a concrete outcome, that guides a student from a problem statement to a solution. Design recipes were first introduced by Felleisen et al. to instruct beginners in programming [12] and later expanded by Morazán to a 2-semester introduction for beginners [36,37,38]. The design recipe for state machines is displayed in Figure 1. Unlike most design recipes for beginners, this design recipe includes machine verification steps (i.e., steps 7 and 8). Such steps are appropriate and necessary for students starting to formally explore the realm of theoretical Computer Science. We note, however, that we are not suggesting that such steps are inappropriate for beginners. On the contrary, beginners have successfully been introduced to program correctness using Hoare logic [20,21] as part of designing `while`-loops [38,49].

Steps 1 and 2 ask for a descriptive machine name, the needed alphabets, and a thorough set of unit tests. Step 3 asks for the conditions that must be tracked as the machine processes the input and associating a state with each. In design recipes for beginners, this step is equivalent to developing a design idea. Step 4 asks for the development of the transition relation. Each transition is developed assuming that the condition describing the source state holds. The actions taken by a transition must guarantee that the conditions of the destination state hold. Steps 5 and 6 ask for the machine's implementation and the running of tests. If errors are thrown or tests fail, students must revisit the steps of the design recipe to debug. Step 7 asks students to formalize the conditions that are tracked by the states (developed for step 3) as a computation advances. This is done by designing and implementing an invariant predicate for each state to affirm its design role. Step 8 asks for a proof of partial correctness. This is done in two steps. First, students prove by induction that the state invariants hold during a

[2] Constructors for grammars and regular expressions are also provide, but these are not addressed in this article

[3] We thank Rose Bohrer for honing us into this vocabulary.

```
#lang fsm
;; State Documentation
;;  S aaba not in the consumed input and no proper prefix of aaba is at the end
;;    of the consumed input, starting state
;;  A aaba not in the consumed input and a is detected at the end of the
;;    consumed input
;;  B aaba not in the consumed input and aa is detected at the end of the
;;    consumed input
;;  C aaba not in the consumed input and aab is detected at the end of the
;;    consumed input
;;  F aaba is detected in the consumed input, final state
(define has-aaba (make-dfa '(S A B C F)
                           '(a b)
                           'S
                           '(F)
                           '((S a A) (S b S)
                             (A a B) (A b S)
                             (B a B) (B b C)
                             (C a F) (C b S)
                             (F a F) (F b F))
                           'no-dead))

(check-equal? (sm-apply has-aaba '())                    'reject)
(check-equal? (sm-apply has-aaba '(a))                   'reject)
(check-equal? (sm-apply has-aaba '(a a))                 'reject)
(check-equal? (sm-apply has-aaba '(a a b))               'reject)
(check-equal? (sm-apply has-aaba '(a a a b))             'reject)
(check-equal? (sm-apply has-aaba '(b b a a b))           'reject)
(check-equal? (sm-apply has-aaba '(a a b a))             'accept)
(check-equal? (sm-apply has-aaba '(b b a a a b a b b))   'accept)
(check-equal? (sm-apply has-aaba '(b b b a a a b a a a)) 'accept)
```

Fig. 2: `dfa` for `L`=$\{$`w`$|$`w`$\in$(a b)* $\wedge$ `w contains aaba`$\}$.

computation. Second, based on state invariants always holding, students prove that the machine's language is correct.

2.3 Programming Machines

To illustrate how students are instructed to design state machines, consider using the design recipe to implement a deterministic finite-state automaton for `L`=$\{$`w`$|$`w`$\in$(a b)* $\wedge$ `w contains aaba`$\}$. The reader is sure to recognize this as an instance of the problem of finding a pattern (i.e., a subword) in a word. Such an example is a good pedagogic choice as it serves as en effective introduction to illustrate the development of the `KMP` algorithm [23].

The machine implementation after following the first 6 steps of the design recipe is displayed in Figure 2. The first step of the design recipe is satisfied

by lines 12 and 13 using the descriptive name `has-aaba` and identifying the alphabet as `(a b)`. The second step of the design recipe is satisfied with the unit tests in lines 23–31. It is emphasized to students that tests must be thorough and, when a machine decides a language, include words that are in and that are not in the machine's language. In this example, the tests are written using `FSM`'s observer, sm-apply, to apply a machine to a word. Step 3 is satisfied by (informally) documenting the meaning of each state as done in lines 2–11 and providing the states to the constructor as done in line 12. Based on the result of step 3, the transition function is formulated to satisfy step 4. Students are taught to assume that the role of a state, say `M`, is satisfied and that the action taken by a transition out of `M` to state `N` must guarantee that the role of `N` is satisfied. For instance, consider developing the transitions out of `A`. In this state, the consumed input ends with an `a` and does not contain `aaba`. Upon reading an `a`, the consumed input does not contain `aaba` and ends with `aa`, satisfying the role of state `B`. Therefore, a transition needed is `(A a B)`. Upon reading a `b`, the consumed input does not contain `aaba` and does not end with a proper prefix of `aaba`, satisfying the role of state `S`. Therefore, a transition needed is `(A a S)`. Performing this analysis for each state yields the transitions needed to satisfy Step 4 and the result is displayed in lines 16–20 in Figure 2. Steps 5 and 6 are satisfied by the code in Figure 2 and running the program to establish that all the tests pass. If syntax or constructor misuse errors are thrown or if any tests fail, students debug by revisiting the steps of the design recipe. In our experience, the scaffolding provided by the design recipe and by the `FSM` error messaging system reduce frustration among students by providing a framework to reason about errors and reducing the sitting in front of a monitor not knowing what to do.

To satisfy step 7, the predicates are designed starting with the state roles identified in step 3. Figure 3 displays the state invariant predicates developed for `has-aaba`[4]. For instance, the invariant predicate, `C-INV` (lines 23–27 in Figure 3) for state `C` takes as input, `ci`, the consumed input. This predicate may be used by the `FSM` visualization tool when the machine transitions into `C`. It checks that `ci` does not contain `aaba` and that it ends with `aab`. If this predicate holds, then the role of `C` is affirmed and the visualization tool renders the state in green. Otherwise, the state is rendered in red. In this manner, students may validate their design before attempting to develop a proof of partial correctness.

To satisfy step 8, it must be established that `has-aaba`'s language is equal to `L`. This is done in two steps. The first establishes by induction, on the number of transitions performed, that the invariant predicates hold during machine execution. In the interest of brevity, we do not outline the entire proof. Instead, we provide proof snippets to communicate the spirit of what is done in the classroom. For the base case, we argue that `S-INV` holds as follows:

[4] In the interest of brevity, neither the unit tests for each invariant predicate nor the implementation of the auxiliary functions (i.e., `contains?`, `ends-with?`, and `does-not-end-with?`) are displayed. We trust that enough context is provided to understand the invariant predicates.

```
;; word → Boolean
;; Purpose: Determine that aaba not in the consumed input and no subword
;;          of aaba is at the end of the consumed input
(define (S-INV ci)
  (or (eq? ci '())
      (and (does-not-end-with? ci '(a))
           (does-not-end-with? ci '(a a))
           (does-not-end-with? ci '(a a b))
           (not (contains? ci '(a a b a))))))

;; word → Boolean
;; Purpose: Determine if aaba not in the consumed input and a is detected
;;          at the end of the consumed input
(define (A-INV ci)
  (and (not (contains? ci '(a a b a))) (ends-with? ci '(a))))

;; word → Boolean
;; Purpose: Determine that aaba not in the consumed input and aa is detected
;;          at the end of the consumed input
(define (B-INV ci)
  (and (not (contains? ci quot(a a b a))) (ends-with? ci '(a a))))

;; word → Boolean
;; Purpose: Determine that aaba not in the consumed input and aab is detected
;;          at the end of the consumed input
(define (C-INV ci)
  (and (not (contains? ci '(a a b a))) (ends-with? ci '(a a b))))

;; word → Boolean
;; Purpose: Determine that aaba is detected in the consumed input
(define (F-INV ci) (contains? ci '(a a b a)))
```

Fig. 3: State invariant predicates for `has-aaba`.

> When the machine starts, the consumed input is empty. Given that `(eq? ci '())` holds, we have that S-INV holds.

For the inductive step, each transition rule is proven correct by assuming that the predicate invariant for the source state holds and establishing that, after the actions performed by the transition, the predicate invariant for the destination state holds. For instance, consider using `(A a B)`:

> By inductive hypothesis, A-INV holds. This means that the consumed input does not contain aaba and that it ends with an a. This transition reads an a. Therefore, after performing this transition, the consumed input ends with aa and does not contain aaba. Thus, B-INV holds.

Finally, to argue for the correctness of the machine, students assume predicate invariants hold and prove the following equivalences:

$$\texttt{w}\in\texttt{L} \Leftrightarrow \texttt{w}\in\texttt{L(has-aaba)} \qquad \texttt{w}\notin\texttt{L} \Leftrightarrow \texttt{w}\notin\texttt{L(has-aaba)}$$

Briefly, for example, students argue that $\texttt{w}\in\texttt{L(has-abaa)} \Rightarrow \texttt{w}\in\texttt{L}$ holds as follows:

> Assume w∈L(has-aaba). Given that predicate invariants always hold, this means that w contains abaa. Thus, w∈L.

The remaining implications are established in a similar manner. In the interest of brevity, their proofs are omitted.

This completes our presentation of how students are taught to design, implement, validate, and verify machines. The reader can appreciate that none of the steps of the design recipe are beyond an advanced undergraduate that has taken a introduction to Discrete Mathematics course and has taken an introduction to programming course. We note that for students that are introduced to programming using HtDP [12] or APS [37] and APD [38], as is the case at WPI and at SHU, the use of FSM quickly becomes natural given that its syntax is familiar to students. For students that are not familiar with Lisp-like syntax, feeling comfortable with FSM may take a little longer and students are helped by the syntax explanations found in the textbook used for instruction [40].

2.4 Programming Constructive Proofs

To illustrate our approach to teaching students about constructive proofs, we outline an algorithm to establish the closure of regular languages under Kleene star. Let M = (make-ndfa S Σ s F R). We denote M's language as L(M).

After a brief class discussion, most students understand that a machine, say $\texttt{M}^*$, that accepts words obtained by concatenating zero or more words in L(M) needs to be designed. With a bit more effort, class discussion defines:

$$\texttt{L}(\texttt{M}^*) = \{\texttt{w} \mid \texttt{w}=\epsilon \vee \texttt{w}=\texttt{w}_0\ldots\texttt{w}_n, \text{ where } \texttt{w}_i\in\texttt{L(M)}\}$$

The definition above is not an algorithm for constructing $\texttt{M}^*$, but students are encouraged to use it to start designing such an algorithm. Based on it, students assert the following about $\texttt{M}^*$:

- needs M's states and δ
- starting state ought to be an accepting state, given that $\epsilon\in\texttt{L}(\texttt{M}^*)$
- $\texttt{M}^*$ needs to loop
- M's starting state cannot be made $\texttt{M}^*$'s starting state
- $\texttt{M}^*$ may nondeterministically move from a final state to its starting state

The first two assertions stem from class discussion observing that words in $\texttt{L}(\texttt{M}^*)$ are constructed from words in L(M) and that $\epsilon\in\texttt{L}(\texttt{M}^*)$. The third assertion stems from observing that multiple words in L(M) may need to be read by $\texttt{M}^*$. The

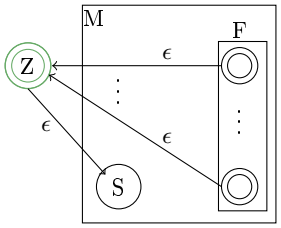


Fig. 4: Graphical description of the construction algorithm for L = L(M)*.

fourth assertion stems from class discussion and/or a homework exercise illustrating what goes wrong if `M`'s starting state is arbitrarily made a final state in `M*`. This observation naturally leads to concluding that a new starting state, say `Z`, is needed for `L(M*)`. The fifth assertion stems from realizing that `M*` needs to loop after reading a word in `L(M)`.

Based on this discussion, both the graphical description of the construction algorithm displayed in Figure 4 and the following formal description are developed:

`M*` = `(make-ndfa S∪{Z} Σ Z F∪{Z} R∪{(f ϵ Z)|f∈F})`, where `Z∉S`

Invariably, a significant number of students find both descriptions difficult to understand. A sample of questions/comments raised by students include:

1. *What is Z?*
2. *What is little f?*
3. *Where did all those arrows labeled with the empty word come from?*
4. *This is unreadable.*
5. *Why is Z a final state?*
6. *Why is there an or-statement [referring to "|"]?.*

We observe that such questions are consistent with the historical observations that students struggle with formal notation.

In our experience, an instructor may repeatedly answer such questions for students, but the answers are not fully assimilated until they are tied to design and programming. To this end, we use the ideas outlined above as a draft design idea, use elements from the design recipe for state machines, and follow the design recipe for function development taught to beginners [12,37,38] to complete the implementation. The developed constructor is displayed in Figure 5. For step 1, the constructor's signature, purpose, and header are displayed in lines 1–3[5]. To satisfy step 2, the unit tests displayed on lines 23–28 are developed using a sample constructor application displayed on line 21. To design and implement the body of the constructor, the input machine and the design of the new machine are considered. Locally, variables are defined the components of

[5] The type `fsa` (i.e., finite-state automaton) denotes either a `dfa` or an `ndfa`.

```
1   ;; fsa → ndfa
2   ;; Purpose: Build the ndfa for L(M)*
3   (define (build-Kleene*-machine M)
4     (let* [(Mstates (sm-states M))
5            (Msigma  (sm-sigma M))
6            (Mstart  (sm-start M))
7            (Mfinals (sm-finals M))
8            (Mrules  (sm-rules M))
9            (Z       (gen-state Mstates))]
10           ;; New state documentation
11           ;;   Z the consumed input is the concatenation of 0 or more words
12           ;;   in L(M), starting state
13      (make-ndfa (cons Z Mstates)
14                 Msigma
15                 Z
16                 (cons Z Mfinals)
17                 (append
18                  (cons (list Z 'ϵ Mstart) Mrules)
19                  (map (lambda (f) (list f 'ϵ Z)) Mfinals)))))
20
21  (define has-aaba-Kstar (build-Kleene*-machine has-aaba))
22
23  (check-equal? (sm-apply has-aaba-Kstar '(a b b))         'reject)
24  (check-equal? (sm-apply has-aaba-Kstar '(a a b b a a b)) 'reject)
25  (check-equal? (sm-apply has-aaba-Kstar '())              'accept)
26  (check-equal? (sm-apply has-aaba-Kstar '(a a b a))       'accept)
27  (check-equal? (sm-apply has-aaba-Kstar '( b a a b b a a b a b))
28                'accept)
```

Fig. 5: Constructor for regular language closure under Kleene star.

the input machine. In addition, a local variable, `Z`, is defined for the needed new state, using `FSM`'s `gen-state`[6]. The roles of `M`'s states remain unchanged in the constructed machine. Students must, however, define the `Z`'s role. An immediate answer provided by some students is that `Z` is the starting state of the constructed machine, which is not enough. It is during this design step that students begin to understand that if the constructed machine is in `Z` then the consumed input is the concatenation of zero or more words in the language of the input machine. This is reflected in lines 10–12 in Figure 5 and provides answers to the first and fifth student questions above. With this new found understanding, most students now understand that the constructed machine may nondeterministically transition from `Z` to `S` to start reading (another) word in the input machine's language and that ϵ-transitions are needed from the input machine's final states to `Z` to loop. Thus, providing concrete answers to the second, third, and sixth student questions above that resonate with most students. Finally, after designing and

[6] Generates a state guaranteeing that it does not match any state in its given input.

implementing the constructor most students state that Figure 4 and its formal description are understandable.

Finally, the proof of correctness follows in two steps as done in Section 2.3. First, students assume that state roles hold for the input machine and argue by induction that for the constructed machine the state roles hold. Second, assuming state roles hold, students argue that `L(M*)`=`L(M)*`, where `M*` is the constructed machine and `M` is the given machine.

3 Empirical Data

To answer RQ1–RQ4, students enrolled in their first `FLAT` course at `SHU` and at `WPI` were anonymously surveyed. The two courses enrolled a total of 106 students, 11 at `SHU` and 95 at `WPI`. A total of 53 students volunteered to participate in the survey, 10 from `SHU` and 43 from `WPI`, making the overall response rate among registered students 43% (the response rate among registered students is 91% at `SHU` and 45% at `WPI`). The day the survey was administered, a total of 43 students attended class at `WPI` (31 in-person and 12 remotely) and 10 students attended class at `SHU` (all in-person). This attendance rate is typical for both institutions. The responses, therefore, reflect the perceptions of students that regularly attend class and not the perceptions of all students registered for these courses. Among the respondents, 14 (26%) self identified as female, 36 (68%) self identified as male, and the remaining 3 (6%) self identified as: 1 nonbinary, 1 agender, and 1 declined to respond. Finally, students at both institutions received neither payment nor benefits for participating in the survey.

The survey includes 10 questions to measure their perceptions. Each question presents a statement and the respondent is asked to identify their level of agreement using using a Likert scale [28]: `[1] Strongly disagree...Strongly agree [5]` (including, [3], a neutral category). The overall response distributions as well as the distributions by gender are presented.

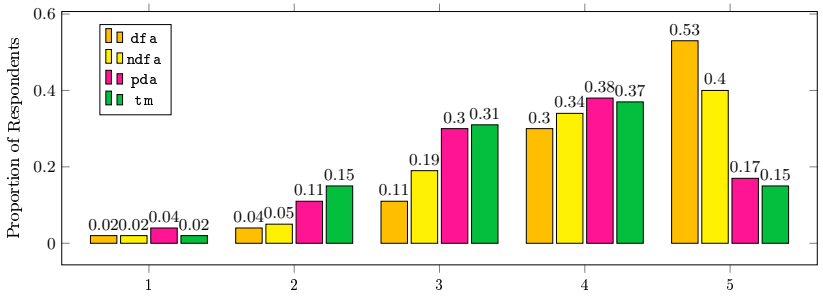
3.1 Programming Machines is Straightforward

To explore RQ1, *Do students feel that programming state machines is straightforward?*, the following statement are used:

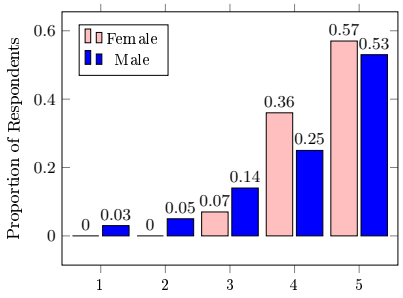
1. `Programming dfas is straightforward.`
2. `Programming ndfas is straightforward.`
3. `Programming pdas is straightforward.`
4. `Programming tms is straightforward`[7]`.`

The overall distribution of responses is displayed in Figure 6a. We observe that for all machine types a majority of respondents tend to agree with the statements: 83%, 74%, 55%, and 52%, respectively for `dfas`, `ndfas`, `pdas`, and `tms`. Thus, for our sample at large, the answer to RQ1 is that students feel that programming

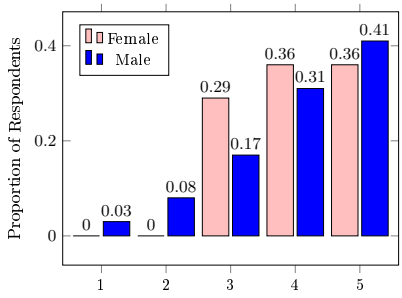
[7] One respondent failed to provide a valid response.



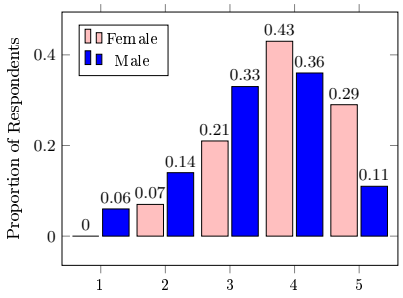
(a) Overall distribution.



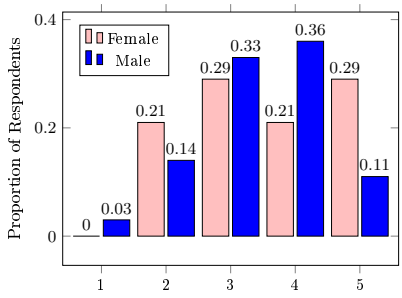
(b) `dfa`



(c) `ndfa`



(d) `pda`



(e) `tm`

Fig. 6: Programming `dfa`/`ndfa`/`pda`/`tm` machines is straightforward.

state machines is straightforward. As machine complexity increases, we observe that the proportion of respondents that tend to agree with the statements decreases. This is to be expected given that the programming tasks become more difficult. Nonetheless, the data suggests that students found the experience of programming state machines straightforward. It is also noteworthy that only a small proportion of respondents tend to disagree with the statements: 6%, 7%, 15%, and 17%, respectively, for `dfa`s, `ndfa`s, `pda`s, and `tm`s. We interpret this as

further evidence is support of our answer to RQ1. These results are very encouraging considering that Computer Science popular culture states: *Beware of the Turing tar-pit in which everything is possible but nothing of interest is easy* [53]. *Easy* may or may not be interchangeable with *straightforward* in this context, but it is certainly the case that anything easy is likely to also be straightforward.

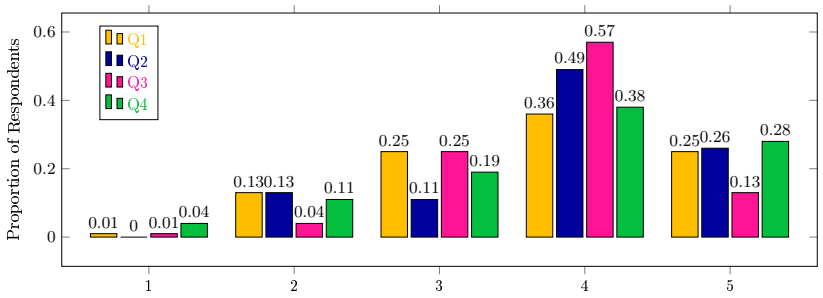
Figures 6b to 6e presents the distributions for females and males. These distributions paint a much more nuanced picture. For all machine types, females students tend to agree (i.e., responses 4 and 5) more or the same with the statements than male students: 93% versus 78%, 72% versus 72%, 72% versus 47%, and 50% versus 47%, respectively, for `dfa`s, `ndfa`s, `pda`s, and `tm`s. Among male students, the proportion agreeing with the statements tends to decrease more sharply as machine complexity increases and drops just under 50% for the two most complex machine types (i.e., `pda` and `tm`). Among females, no such tendency is observed. The female proportion agreeing with the statements drops from `dfa` to `ndfa`, remains steady from `ndfa` to `pda`, and drops from `pda` to `tm`. The first drop coincides with student's first exposure to nondeterminism and, therefore, is expected regardless of gender. The female proportion agreeing remains steady for `pda`s and presents the widest gap with male students. It is difficult to explain why this occurs, but a plausible explanation is that female students tend to do less unguided exploration and adhere more closely to the steps of the design recipe. Thus, they assimilate nondeterminism better than male students. The next drop in the female agreeing proportion, from `pda` to `tm`, coincides with the introduction of mutation (i.e., `tm`s mutating their tape), which is also expected regardless of gender given that reasoning about mutations is challenging. The challenge appears significant enough to virtually make the levels of agreement among female and male students the same. Our answer for RQ1, when examined by gender, is that both genders agree that programming state machines is straightforward. However, female students tend to feel stronger on this dimension.

3.2 Intellectual Stimulation and CS Education Relevance

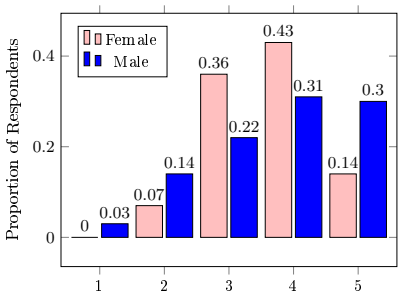
To explore RQ2, *Do students feel that Automata Theory, programming state machines, and programming constructive algorithms is intellectually stimulating?*, and RQ3, *Do students feel that Automata Theory is relevant to their Computer Science education?*, the following statements are used:

Q1. `Automata Theory is intellectually stimulating.`
Q2. `Programming state machines [grammars and regular expressions]`[8] `is intellectually stimulating.`
Q3. `Programming constructive algorithms is intellectually stimulating.`
Q4. `The course is relevant to my Computer Science education.`

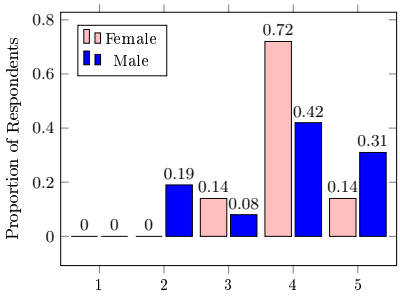
[8] The survey studied topics beyond the scope of this article. Square brackets are used to indicate full question text where not the entire text is relevant to this article.



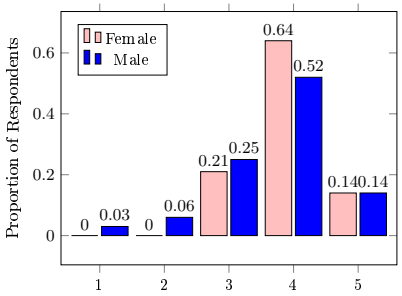
(a) Overall intellectual stimulation and relevance to CS education.



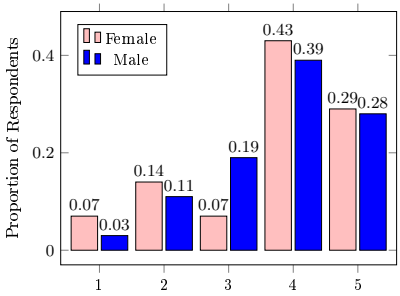
(b) Automata theory.



(c) Programming state machines.



(d) Programming constructive algorithms.



(e) Relevant to CS education.

Fig. 7: Intellectual stimulation and relevance to CS education .

The overall distribution is displayed in Figure 7a. We observe that for all statements a majority of respondents tend to agree (i.e., answers 4 and 5). An unexpected result is that 61% of the respondents tend to agree that Automata Theory is intellectually stimulating. A further unexpected result is that only a small minority, 14%, tend to disagree. This bucks the historical trend that suggests that students do not feel stimulated by Automata Theory. Given that our curriculum covers all the classical topics taught in a first Automata Theory

course, we interpret this as evidence that a programming-based approach makes Automata Theory more stimulating for students. Another encouraging result is that an overwhelming majority (i.e., $> 67\%$), respectively 75% and 70%, tend to agree that programming state machines and programming constructive algorithms is intellectually stimulating. Therefore, the respondents feel more strongly about programming in Automata Theory than about Automata Theory itself. We interpret this as evidence that programming is an effective means to engage Computer Science students in Automata Theory. Thus, our answer to RQ2, for our sample at large, is that students feel that Automata Theory, programming state machines, and programming constructive algorithms is intellectually stimulating.

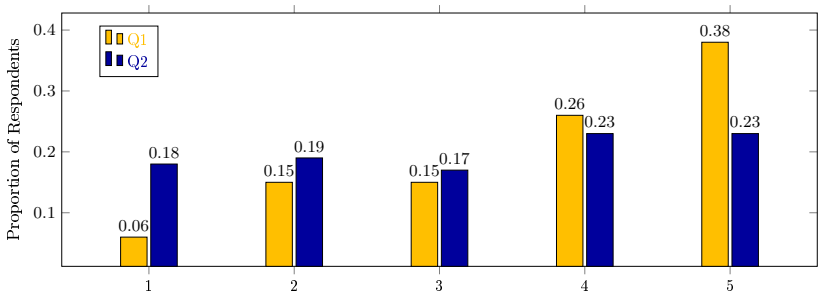
A majority of respondents, 66%, tend to agree that the course in relevant to their Computer Science education. Besides unexpected, this result, once again, goes against what is reported in the literature. To us, it suggests that a programming approach is a key that makes Automata Theory appealing and relevant to Computer Science majors. Therefore, for our sample at large, our answer to RQ3 is that students feel that course is relevant to their Computer Science education.

The distributions by gender are displayed in Figures 7b to 7e. We observe that a larger proportion of female students, 57%, than males students, 34%, tend to agree (responses 4 and 5) that Automata Theory is intellectually stimulating. This suggests that our design- and programming-based approach makes Automata Theory significantly more stimulating for female students. We also observe that a larger proportion of female students, 86%, than males students, 73%, tend to agree that programming state machines is intellectually stimulating. In addition, we see a larger proportion among females, 78% versus 66%, agreeing that programming constructive algorithms is intellectually stimulating. These results suggest that connecting Automata Theory with programming is more intellectually stimulating for females students. Our answer for RQ2, under the lens of gender, is that female students find Automata Theory, when connected with programming, more intellectually stimulating than male students and that both genders find programming in `FLAT` intellectually stimulating.

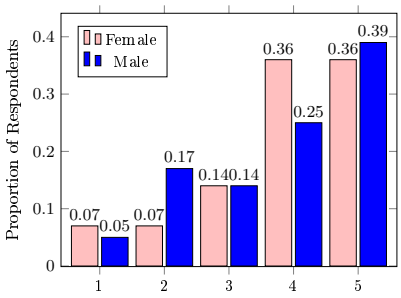
Finally, we observe that both genders exhibit overwhelming majorities, 72% for females and 67% for males, that tend to agree (i.e., responses 4 and 5) that the course is relevant to their Computer Science education. This is a rather astonishing result considering decades of research suggesting that students feel that Automata Theory has little relevance to their major. It suggests that a design- and programming-based methodology in `FLAT` is an effective means to make it relevant for male and female Computer Science students. Our answers for RQ3, under the lens of gender, is that both genders in our sample feel very strongly that `FLAT` is relevant to their Computer Science education.

3.3 `FSM` Programming Helpfulness

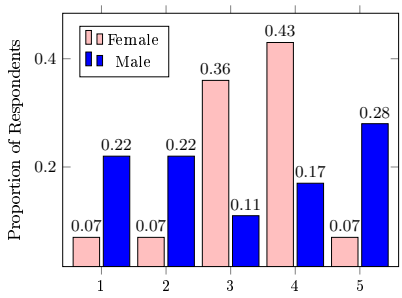
To explore RQ4, *Is programming in FSM useful to understand Automata Theory and to develop constructive proofs?*, the survey presents the following statements:



(a) Overall helpfulness for Automata Theory material.



(b) Understand Automata theory.



(c) Constructive proofs.

Fig. 8: `FSM` Programming helpfulness.

Q1 `Programming in FSM helped me understand Automata Theory.`
Q2 `Writing programs in FSM helped me develop constructive proofs.`

The overall distribution of responses is displayed in Figure 8. We observe that the majority of respondents, 64%, tend to agree (i.e., responses 4 and 5) that programming in `FSM` is useful to understand Automata Theory. We interpret this as evidence that programming helps elucidate topics in Automata Theory. This suggests that future studies ought to tease out what topics are elucidated. We hypothesize, for example, that programming helps students understand nondeterminism.

In contrast, the distribution for Q2 is polarized. Just less than half of the respondents, 46%, tend to agree that writing programs in `FSM` helps them develop constructive proofs while 37% tend to disagree. This polarization is unexpected. It can be argued that the act of only writing, not designing, a machine definition in `FSM` provides little help in developing a proof. Our curriculum, however, requires students to carefully elaborate the meaning of each state and develop invariant predicates that can be validated with `FSM` visualization tools. Both of these activities have a direct bearing on the development of correctness proofs. This suggest that there is room for improvement in tying together design and

proof development or that the survey statement is not detailed enough causing students to equate "writing programs" with only writing machine definitions. Therefore, our answer to RQ4, for the sample population at large, is that the results are inconclusive.

The distributions by gender are displayed in Figures 8b to 8c. It paints a much more nuanced picture. A larger proportion of female students, 72%, tend to agree (responses 4 and 5) with Q1 versus 64% for male students. A similar disparity is observed with the proportions, 14% and 22%, that disagree (i.e., responses 1 and 2). This suggests that female students find programming in `FSM` is more elucidating regarding Automata Theory topics. For Q2, we observe that half of the female respondents tend to agree (i.e., responses 4 and 5) and about a third, 36%, are neutral. The distribution for male students is polarized with 45% that tend to agree and 44% that tend to disagree. This suggests that female students are more likely than male students to see the connection between designing and implementing construction algorithms and the development of correctness proofs. A plausible explanation for this, once again, is that female students adhere more closely to following the steps of the design recipe. Thus, our answer for RQ4, under the lens of gender, is that female students find programming in `FSM` useful to understand Automata Theory and to develop constructive proofs. On the other hand, the results are mixed for male students. Male students find programming in `FSM` useful to understand Automata Theory. However, the results are inconclusive for male students regarding whether programming in `FSM` is useful to develop constructive proofs. These results, indeed, suggest that future research is required to explain this gender difference in more detail.

4 Related Work

4.1 `FLAT` Instruction

A significant hurdle that many `FLAT` students need to overcome is the intrinsic mathematical/theoretical nature of the material [5,7,11,30,51,54,64,65]. For Computer Science students, this problem can be compounded by a lack of experience with formal notation [5,30], a lack of a natural programming component that permeates most other courses in the curriculum [58,63], a lack of maturity in developing proofs [7], and a lack of problem-solving abilities [55,65]. Collectively, these problems lead students to feel that `FLAT` courses present a high risk of failure [65] and can lead to high drop-out rates [54,63]. Furthermore, students fail to develop motivation to master the material given that they perceive it as having little or no relevance to the Computer Science major [7,51,54]. If a `FLAT` course is taught using a traditional "chalkboard approach," it is common for students to also have difficulty piecing together the details of machine execution [33] and of construction algorithms [54,55] resulting in students only achieving a superficial understanding of the material [4,63].

To address these problems, many `FLAT` instructors have turned to a constructivism [4,54,63]. Constructivism is a theory of learning that postulates that knowledge is constructed by students engaged in building activities [2,52]. In

FLAT education, this has taken two primary approaches. The first (and most common) approach uses visualizations to engage students in building instances of the models of computations they study (e.g., [1,3,10,19,32,57,59,58,61,63]). These tools can help students develop conceptual models, but must be used with care. Visualization tools by themselves do not help students develop knowledge nor their problem-solving skills [2,55,63]. The second approach uses (textual) programming to engage students in building instances of the models of computations they study (e.g., [7,67]). Such approaches promote deep learning and encourage exploration by having students, carefully think about a programming implementation.

In contrast, FSM and its design-based methodology is the first effort uniting both approaches. In addition to providing students with the opportunity to program machines, FSM also integrates visualization tools to help understand algorithms taught in the FLAT classroom (e.g., [44,45,46,47,48]). The FSM visualization tools are designed to engage students in thinking by providing informative messages that explain the last step taken. In this manner, FSM helps students by presenting two views of FLAT algorithms and helps instructors by providing the means to move away from static images to explain dynamic processes.

4.2 Female Students in Computer Science

It is widely recognized that the female Computer Science student population faces unique obstacles. Perhaps, the most accentuated problem is self-confidence [8,13,15,22,31]. In this population, low self-confidence is attributed to many factors including stereotypes about hacking skills being a prerequisite to study Computer Science [15], about needing to be narrowly focused on programming [15,25,62], about women's abilities in science [31], and about Computer Science being mundane and tedious [25]. These stereotypes create a hostile environment, for example, by suggesting that others know so much more [15,31], by suggesting that the study of Computer Science is unrelated to its usefulness in the real world [62], by intimidating those that do not finish assignments quickly [68], and by suggesting that long hours of unguided exploration (i.e., hacking) or luck is rewarded when completing assignments [68].

To combat these stereotypes, the literature suggests taking actions outside and inside of the classroom. Outside classroom, universities can increase the number of female role models by increasing the number of female faculty members [68] and create a departmental environment that highlights the creative, the interesting, and the challenging opportunities found in Computer Science. Inside the classroom, instructors can use examples that highlight the benefits of Computer Science in real life [62], can create an environment that encourages questions [68], can highlight that successes are not due to luck and failures are not due to lack of ability [68], and can actively discourage the belief that "you are only here because you are a girl" [31].

5 Discussion

In this section, we address how we believe the FSM methodology addresses the challenges in teaching FLAT courses. We address some of the challenges faced by all students and some of the compounding challenges faced by female students.

5.1 Addressing Challenges in Teaching FLAT Courses

Providing a programming language is not enough to make FLAT courses attractive to students. The programming language must not require skills beyond those expected from students taking their first FLAT course and must provide the proper abstractions to build instances of computation models and to implement construction algorithms. For instance, we would not expect programming state machines using assembly to satisfy this criteria neither would we expect this result if students had to implement the searches necessary to determine if a nondeterministic machine accepts or rejects a given word. The results obtained for RQ1, suggest that FSM meets this criteria for programming state machines and associated construction algorithms. If the use of FSM abstractions were beyond the skills of our students, then it is unlikely that students would find programming them straightforward.

When the language provides an environment in which programming is straightforward, it provides a natural programming component that permeates most other courses in the Computer Science curriculum. Students have the opportunity to focus on problem solving and cast theory problems, expressed in formal notation, as programming problems. Thus, helping students understand the material's intrinsic theoretical nature and the formal notation used by creating an environment for exploration and knowledge-building that provides immediate feedback upon running programs. In turn, this sparks intellectual interest and provides the sense that FLAT is relevant to a student's Computer Science education as reflected in the responses obtained for RQ2 and RQ3. Furthermore, it helps understand the responses obtained for RQ4: implementing a machine or an algorithm that works confirms understanding and is a step towards developing a constructive proof.

The lack of problem solving and proof development skills commonly reported by FLAT instructors is, in part, addressed by providing students with a design recipe for state machines. The design recipe provides scaffolding to guide students in the problem solving process. This scaffolding does not exist in most FLAT textbooks and, therefore, probably in most FLAT courses. Thus, feeding student fears that they are at high risk for failure, which is likely to prompt higher drop-out rates. For students that are introduced to programming using a design-based methodology [12,37,38], this approach creates a stronger connection with other Computer Science courses, which reinforces the feeling of relevance to their major. In addition, it also provides scaffolding for proof development, which elucidates the results seen for RQ4.

The use of a design recipe also addresses some of the concerns we face with female students. It emphasizes design over hacking, expands problem solving

beyond coding, and moves `FLAT` beyond (the perception of) its mundane and tedious mathematical nature. Female students see themselves, as well as their male counterparts, work hard to fulfill the steps of the design recipe, thus, eliminating to some degree the perceptions that unguided exploration and luck is rewarded and that others know more because they finish assignments quickly. In addition, using examples, like finding a pattern from Section 2.3, reinforces (to students of all genders) that `FLAT` is relevant in the real world.

Finally, using a design recipe encourages questions. It provides a *lingua franca* in which students and instructors can discuss `FLAT` programs. Instead of not knowing what to ask when they are confused, students can ask questions about a design recipe step they are stuck on. Furthermore, instructors can ask questions about a student's design in the language used in class and found in the textbook.

6 Concluding Remarks and Future Work

This paper illustrates a design- and programming-based methodology to teach students about state machines in a Formal Languages and Automata Theory course using the domain-specific language `FSM`. In addition, it explores student perceptions, overall and by gender, about programming state machines, about how intellectually stimulating Automata Theory, programming state machines, and programming constructive proofs are, about the relevance of Automata Theory to majoring in Computer Science, and about the usefulness of programming to understand Automata Theory and to develop constructive proofs. The empirical results suggest the following conclusions:

- students of both genders feel that programming state machines in `FSM` is straightforward
- students feel that Automata Theory, programming state machines, and programming constructive algorithms is intellectually stimulating, but female students feel stronger than male students that programming state machines and construction algorithms is intellectually stimulating
- female students feel more strongly that Automata Theory is intellectually stimulating
- students of both genders feel that Automata Theory is relevant to majoring in Computer Science
- students feel that programming in `FSM` helps them understand Automata Theory, but female students feel more strongly about it
- students are polarized about how much `FSM` programming helps them develop constructive proofs, but male students more so than female students

Future work includes more nuanced studies to better understand what topics students feel are most elucidated by our design- and programming-based approach using state machines. Future work also includes studies to measure student perceptions about our design- and programming-based approach using grammars (i.e., regular, context-free, and context-sensitive). Finally, future work will explore the effectiveness of `FSM` visualizations to help students understand and to motivate interest in `FLAT`.

References

1. Barwise, J., Etchemendy, J.: Turing's world. Journal of Symbolic Logic **55**(1), 370–371 (1990). https://doi.org/10.2307/2275002
2. Ben-Ari, M.: Constructivism in computer science education. In: Proceedings of the Twenty-Ninth SIGCSE Technical Symposium on Computer Science Education. p. 257–261. SIGCSE '98, Association for Computing Machinery, New York, NY, USA (1998). https://doi.org/10.1145/273133.274308
3. Castro-Schez, J.J., del Castillo, E., Hortolano, J., Rodriguez, A.: Designing and Using Software Tools for Educational Purposes: FLAT, a Case Study. IEEE Transactions on Education **52**(1), 66–74 (2009). https://doi.org/10.1109/TE.2008.917197
4. Chesñevar, C.I., González, M.P., Maguitman, A.G.: Didactic Strategies for Promoting Significant Learning in Formal Languages and Automata Theory. In: Proceedings of the 9th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education. p. 7–11. ITiCSE '04, Association for Computing Machinery, New York, NY, USA (2004). https://doi.org/10.1145/1007996.1008002
5. Chesñevar, C.I., Cobo, M.L., Yurcik, W.: Using Theoretical Computer Simulators for Formal Languages and Automata Theory. ACM SIGCSE Bull. **35**(2), 33–37 (2003). https://doi.org/10.1145/782941.782975
6. Christensen, M.A.: Tracing the Gender Confidence Gap in Computing: A Cross-National Meta-Analysis of Gender Differences in Self-Assessed Technological Ability. Social Science Research **111**, 102853 (2023). https://doi.org/https://doi.org/10.1016/j.ssresearch.2023.102853
7. Chua, Y.S., Winton, C.N.: Teaching Theory of Computation at the Junior Level. In: Proceedings of the International Conference on APL: APL in Transition. p. 69–78. APL '87, Association for Computing Machinery, New York, NY, USA (1987). https://doi.org/10.1145/28315.28324
8. Du, J., Wimmer, H.: Hour of Code: A Study of Gender Differences in Computing. Information Systems Education Journal **17** (aug 2019)
9. Dzhatdoyev, S., Des Rosiers, J.A., Morazán, M.T.: A Contract-Based Error Messaging and Tutorial System to Support Design: Improved Error Messages in FSM. In: Hemann, J. (ed.) Proceedings of Trends in Functional Programming 2024. LNCS, Springer (2024), to appear
10. D'Antoni, L., Helfrich, M., Kretinsky, J., Ramneantu, E., Weininger, M.: Automata Tutor v3. In: Computer Aided Verification: 32nd International Conference, CAV 2020, Los Angeles, CA, USA, July 21–24, 2020, Proceedings, Part II. p. 3–14. Springer-Verlag, Berlin, Heidelberg (2020). https://doi.org/10.1007/978-3-030-53291-8_1
11. Estrebou, F.C., Lanza, M., Mauco, V., Barbuzza, R., Favre, L.: Minerva: Una Herramienta para un Curso de Lenguajes Formales y Autómatas. https://www.researchgate.net/publication/266384889_Minerva_Una_Herramienta_para_un_Curso_de_Lenguajes_Formales_y_Automatas (2002), last accessed: October 2024
12. Felleisen, M., Findler, R.B., Flatt, M., Krishnamurthi, S.: How to Design Programs: An Introduction to Programming and Computing. MIT Press, Cambridge, MA, USA, Second edn. (2018)
13. Fisher, A., Margolis, J.: Unlocking the Clubhouse: The Carnegie Mellon Experience. SIGCSE Bull. **34**(2), 79–83 (Jun 2002). https://doi.org/10.1145/543812.543836

14. Fisher, A., Margolis, J., Miller, F.: Undergraduate Women in Computer Science: Experience, Motivation and Culture. SIGCSE Bull. **29**(1), 106–110 (mar 1997). https://doi.org/10.1145/268085.268127
15. Fisher, A., Margolis, J., Miller, F.: Undergraduate Women in Computer Science: Experience, Motivation and Culture. In: Proceedings of the Twenty-Eighth SIGCSE Technical Symposium on Computer Science Education. p. 106–110. SIGCSE '97, Association for Computing Machinery, New York, NY, USA (1997). https://doi.org/10.1145/268084.268127
16. Flatt, M., Findler, R.B., PLT: The Racket Guide. PLT (2024), https://docs.racket-lang.org/guide/, last accessed: June 2024
17. Gansner, E.R., Koutsofios, E., North, S.C., V, K.P.: A Technique for Drawing Directed Graphs. IEEE Transactions on Software Engineering **19**(3), 214–230 (1993). https://doi.org/10.1109/32.221135
18. Gansner, E.R., North, S.C.: An Open Graph Visualization System and Its Applications to Software Engineering. Softw. Pract. Exper. **30**(11), 1203–1233 (sep 2000). https://doi.org/10.1002/1097-024X(200009)30:11<1203::AID-SPE338>3.0.CO;2-N
19. García-Osorio, C., Mediavilla-Sáiz, I.n., Jimeno-Visitación, J., García-Pedrajas, N.: Teaching Push-down Automata and Turing Machines. In: Proceedings of the 13th Annual Conference on Innovation and Technology in Computer Science Education. p. 316. ITiCSE '08, Association for Computing Machinery, New York, NY, USA (2008). https://doi.org/10.1145/1384271.1384359
20. Hoare, C.A.R.: An axiomatic basis for computer programming. Commun. ACM **12**(10), 576–580 (Oct 1969). https://doi.org/10.1145/363235.363259
21. Hoare, C., Jifeng, H.: Unifying Theories of Programming. Prentice Hall series in computer science, Prentice Hall (1998)
22. Kemp, P.E.J., Wong, B., Berry, M.G.: Female Performance and Participation in Computer Science: A National Picture. ACM Trans. Comput. Educ. **20**(1) (nov 2019). https://doi.org/10.1145/3366016
23. Knuth, D.E., Morris, Jr., J.H., Pratt, V.R.: Fast pattern matching in strings. SIAM Journal on Computing **6**(2), 323–350 (1977). https://doi.org/10.1137/0206024
24. Korte, L., Anderson, S., Pain, H., Good, J.: Learning by Game-Building: A Novel Approach to Theoretical Computer Science Education. SIGCSE Bull. **39**(3), 53–57 (jun 2007). https://doi.org/10.1145/1269900.1268802
25. Lasen, M.: Education and Career Pathways in Information Communication Technology: What are Schoolgirls Saying? Computers & Education **54**(4), 1117–1126 (2010). https://doi.org/https://doi.org/10.1016/j.compedu.2009.10.018
26. Lau, W.W.F., Yuen, A.H.K.: Exploring the Effects of Gender and Learning Styles on Computer Programming Performance: Implications for Programming Pedagogy. British Journal of Educational Technology **40**(4), 696–712 (2009). https://doi.org/https://doi.org/10.1111/j.1467-8535.2008.00847.x
27. Lewis, H.R., Papadimitriou, C.H.: Elements of the Theory of Computation. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2nd edn. (1997). https://doi.org/10.1145/300307.1040360
28. Likert, R.: A Technique for the Measurement of Attitudes. Archives of Psychology **140**, 1–55 (1932)
29. Linz, P.: An Introduction to Formal Languages and Automata. Jones and Bartlett Publishers, Inc., USA, 5th edn. (2011)
30. Mane, D.T., Howal, S.S., Lokare, V.T.: Problem-based Learning using Simulation Tools for Automata Theory. Journal of Engineering Education Transformations **30** (2016). https://doi.org/10.16920/jeet/2016/v0i0/85708

31. Margolis, J., Fisher, A.: Unlocking the Clubhouse: Women in Computing. MIT Press, Cambridge, MA (2001)
32. Martínez, M., Barbuzza, R., Mauco, M.V., Favre, L.: MTSolution: A Visual and Interactive Tool for a Formal Languages and Automata Course. Information Systems Education Journal **7**(10) (March 2009), http://isedj.org/7/10/
33. McDonald, J.: Interactive Pushdown Automata Animation. SIGCSE Bull. **34**(1), 376–380 (feb 2002). https://doi.org/10.1145/563517.563489
34. McDowell, C., Werner, L.L., Bullock, H.E., Fernald, J.: The Impact of Pair Programming on Student Performance, Perception and Persistence. In: Clarke, L.A., Dillon, L., Tichy, W.F. (eds.) Proceedings of the 25th International Conference on Software Engineering, May 3-10, 2003, Portland, Oregon, USA. pp. 602–607. IEEE Computer Society (2003). https://doi.org/10.1109/ICSE.2003.1201243
35. Morazán, M.: Composing Turing Machines in FSM. In: Proceedings of the 2023 ACM SIGPLAN International Symposium on SPLASH-E. p. 38–49. SPLASH-E 2023, Association for Computing Machinery, New York, NY, USA (2023). https://doi.org/10.1145/3622780.3623647
36. Morazán, M.T.: Infusing an HtDP-based CS1 with Distributed Programming Using Functional Video Games. Journal of Functional Programming **28**, e5 (2018). https://doi.org/10.1017/S0956796818000059
37. Morazán, M.T.: Animated Problem Solving - An Introduction to Program Design Using Video Game Development. Texts in Computer Science, Springer (2022). https://doi.org/10.1007/978-3-030-85091-3
38. Morazán, M.T.: Animated Program Design - Intermediate Program Design Using Video Game Development. Texts in Computer Science, Springer (2022). https://doi.org/10.1007/978-3-031-04317-8
39. Morazán, M.T.: FSM. Seton Hall University, South Orange, NJ, USA (2024), https://morazanm.github.io/fsm/fsm/index.html, last Accessed: October 2024
40. Morazán, M.T.: Programming-Based Formal Languages and Automata Theory - Design, Implement, Validate, and Prove. Texts in Computer Science, Springer (2024). https://doi.org/10.1007/978-3-031-43973-5
41. Morazán, M.T., Antunez, R.: Functional Automata–Formal Languages for Computer Science Students. In: Caldwell, J.L., Hölzenspies, P.K.F., Achten, P. (eds.) Proceedings 3rd International Workshop on Trends in Functional Programming in Education, TFPIE 2014, Soesterberg, The Netherlands, 25th May 2014. EPTCS, vol. 170, pp. 19–32 (2014). https://doi.org/10.4204/EPTCS.170.2
42. Morazán, M.T., Bohrer, R., Dzhatdoyev, S., Des Rosiers, J.A.: Cooking Up Errors for Languages and Automata: Recipe-Based Error Messages in the Classroom. In: TBA (ed.) Proceedings of CHI 2025. ACM (2025), under Review
43. Morazán, M.T., Des Rosiers, J.A.: FSM Error Messages. In: Achten, P., Miller, H. (eds.) Proceedings Seventh International Workshop on Trends in Functional Programming in Education, TFPIE@TFP 2018, Chalmers University, Gothenburg, Sweden, 14th June 2018. EPTCS, vol. 295, pp. 1–16 (2018). https://doi.org/10.4204/EPTCS.295.1
44. Morazán, M.T., Kempinski, O.: Using Computation Graphs to Explain Nondeterminism to Students. In: Proceedings of the 2024 ACM SIGPLAN International Symposium on SPLASH-E. p. 23–33. SPLASH-E '24, Association for Computing Machinery, New York, NY, USA (2024). https://doi.org/10.1145/3689493.3689978
45. Morazán, M.T., Minić, T.: Finite-state automaton to/from regular expression visualization. Electronic Proceedings in Theoretical Computer Science **405**, 36–55 (jul 2024). https://doi.org/10.4204/eptcs.405.3, in Proceedings TFPIE 2024

46. Morazán, M.T., Minić, T.: Nondeterministic to Deterministic Finite-State Machine Visualization: Implementation and Evaluation. In: ITiCSE 2024: Proceedings of the 2024 Conference on Innovation and Technology in Computer Science Education V. 1. p. 262–268. ITiCSE 2024, Association for Computing Machinery, New York, NY, USA (2024). https://doi.org/10.1145/3649217.3653641
47. Morazán, M.T., Minić, T., Kempinski, O.: Visualizing Composed Turing Machines. In: Proceedings of the 2024 ACM SIGPLAN International Symposium on SPLASH-E. p. 34–44. SPLASH-E '24, Association for Computing Machinery, New York, NY, USA (2024). https://doi.org/10.1145/3689493.3689979
48. Morazán, M.T., Schappel, J.M., Mahashabde, S.: Visual Designing and Debugging of Deterministic Finite-State Machines in FSM. Electronic Proceedings in Theoretical Computer Science **321**, 55–77 (aug 2020). https://doi.org/10.4204/eptcs.321.4
49. Morazán, M.T.: How to Design While Loops. Electronic Proceedings in Theoretical Computer Science **321**, 1–18 (Aug 2020). https://doi.org/10.4204/eptcs.321.1
50. Naveed, M.S., Sarim, M.: Didactic Strategy for Learning Theory of Automata and Formal Languages. Proceedings of the Pakistan Academy of Sciences: A. Physical and Computational Sciences **55**(2), 55–67 (2018), https://www.ppaspk.org/index.php/PPAS-A/article/view/171
51. Neeman, A.: Buy One Get One Free: Automata Theory Concepts Through Software Test. J. Comput. Sci. Coll. **31**(6), 90–96 (Jun 2016)
52. Papert, S., Harel, I.: Situating Constructionism. In: Papert, S., Harel, I. (eds.) Constructionism, chap. 1. Ablex Publishing Corporation, Norwood, NJ (1991), http://www.papert.org/articles/SituatingConstructionism.html
53. Perlis, A.J.: Special Feature: Epigrams on Programming. SIGPLAN Not. **17**(9), 7–13 (sep 1982). https://doi.org/10.1145/947955.1083808
54. Pillay, N.: Teaching the Theory of Formal Languages and Automata in the Computer Science Undergraduate Curriculum. South Afr. Comput. J. **42**, 87–94 (2008), http://reference.sabinet.co.za/document/EJC28069
55. Pillay, N.: Learning Difficulties Experienced by Students in a Course on Formal Languages and Automata Theory. SIGCSE Bull. **41**(4), 48–52 (jan 2010). https://doi.org/10.1145/1709424.1709444
56. Rich, E.: Automata, Computability and Complexity: Theory and Applications. Pearson Prentice Hall (2019)
57. Rodger, S.H.: JFLAP: An Interactive Formal Languages and Automata Package. Jones and Bartlett Publishers, Inc., USA (2006)
58. Rodger, S.H., Bilska, A.O., Leider, K.H., Procopiuc, C.M., Procopiuc, O., Salemme, J.R., Tsang, E.: A collection of tools for making automata theory and formal languages come alive. In: White, C.M., Erickson, C., Klein, B.J., Miller, J.E. (eds.) Proceedings of the 28th SIGCSE Technical Symposium on Computer Science Education, SIGCSE 1997, San Jose, California, USA, February 27 - March 1, 1997. pp. 15–19. ACM (1997). https://doi.org/10.1145/268084.268089
59. Rodger, S.H., Bressler, B., Finley, T., Reading, S.: Turning automata theory into a hands-on course. In: Baldwin, D., Tymann, P.T., Haller, S.M., Russell, I. (eds.) Proceedings of the 37th SIGCSE Technical Symposium on Computer Science Education, SIGCSE 2006, Houston, Texas, USA, March 3-5, 2006. pp. 379–383. ACM (2006). https://doi.org/10.1145/1121341.1121459
60. Sipser, M.: Introduction to the Theory of Computation. Cengage Learning, USA, 3rd edn. (2013)

61. Stallmann, M.F., Balik, S.P., Rodman, R.D., Bahram, S., Grace, M.C., High, S.D.: ProofChecker: An Accessible Environment for Automata Theory Correctness Proofs. In: Proceedings of the 12th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education. p. 48–52. ITiCSE '07, Association for Computing Machinery, New York, NY, USA (2007). https://doi.org/10.1145/1268784.1268801
62. Vekiri, I.: Information Science Instruction and Changes in Girls' and Boy's Expectancy and Value Beliefs: In Search of Gender-Equitable Pedagogical Practices. Computers & Education **64**, 104–115 (2013). https://doi.org/https://doi.org/10.1016/j.compedu.2013.01.011
63. Verma, R.M.: A Visual and Interactive Automata Theory Course Emphasizing Breadth of Automata. In: Proceedings of the 10th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education. p. 325–329. ITiCSE '05, Association for Computing Machinery, New York, NY, USA (2005). https://doi.org/10.1145/1067445.1067535
64. Vieira, L.F.M., Vieira, M.A.M., Vieira, N.J.: Language Emulator, A Helpful Toolkit in the Learning Process of Computer Theory. SIGCSE Bull. **36**(1), 135–139 (mar 2004). https://doi.org/10.1145/1028174.971348
65. Vijayalaskhmi, M., Karibasappa, K.: Activity Based Teaching Learning in Formal Languages and Automata Theory - An Experience. In: 2012 IEEE International Conference on Engineering Education: Innovative Practices and Future Trends (AICERA). pp. 1–5. IEEE (2012). https://doi.org/10.1109/AICERA.2012.6306722
66. Welsh, N., Culpepper, R.: RackUnit: Unit Testing. PLT Racket, v8.12 edn. (May 2024), last accessed 05-20-2024
67. Wermelinger, M., Dias, A.M.: A Prolog Toolkit for Formal Languages and Automata. In: Proceedings of the 10th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education. p. 330–334. ITiCSE '05, Association for Computing Machinery, New York, NY, USA (2005). https://doi.org/10.1145/1067445.1067536
68. Wilson, B.C.: A Study of Factors Promoting Success in Computer Science Including Gender Differences. Computer Science Education **12**(1-2), 141–164 (2002). https://doi.org/10.1076/csed.12.1.141.8211