Regular Expressions for Computer Science Students

Marco T. Morazán

Seton Hall University

Outline

Introduction

- Let's go beyond a pencil-and-paper formal languages and automata theory course (without losing rigor)
- Bugs in a pencil-and-paper regular expression are hard to detect
- Hard to prove anything in a buggy regular expression

Introduction

- Let's go beyond a pencil-and-paper formal languages and automata theory course (without losing rigor)
- Bugs in a pencil-and-paper regular expression are hard to detect
- Hard to prove anything in a buggy regular expression
- A programming-based approach to teaching regular expressions in the first automata theory course using FSM

Introduction

- Let's go beyond a pencil-and-paper formal languages and automata theory course (without losing rigor)
- Bugs in a pencil-and-paper regular expression are hard to detect
- Hard to prove anything in a buggy regular expression
- A programming-based approach to teaching regular expressions in the first automata theory course using FSM
- All the theory addressed by a traditional non-programming automata theory course
- Students are engaged by programming regular expressions and by designing and implementing programs based on regular expressions

Introduction

- Let's go beyond a pencil-and-paper formal languages and automata theory course (without losing rigor)
- Bugs in a pencil-and-paper regular expression are hard to detect
- Hard to prove anything in a buggy regular expression
- A programming-based approach to teaching regular expressions in the first automata theory course using FSM
- All the theory addressed by a traditional non-programming automata theory course
- Students are engaged by programming regular expressions and by designing and implementing programs based on regular expressions
- Brings students to the realization that *regular expressions are an elegant way to describe an algorithm* for generating members of a language

Related Work

- Start with finite-state automatons and discussion leads to regular expressions or vice versa

Related Work

- Start with finite-state automatons and discussion leads to regular expressions or vice versa
- Depth of their treatment varies a great deal
- Informal definition, briefly discuss an application (e.g., lexical analysis), and then the equivalence between regular expressions and finite-state automatons
- Most textbooks provide a formal definition and move the equivalence between regular expressions and finite-state automatons

Related Work

- Start with finite-state automatons and discussion leads to regular expressions or vice versa
- Depth of their treatment varies a great deal
- Informal definition, briefly discuss an application (e.g., lexical analysis), and then the equivalence between regular expressions and finite-state automatons
- Most textbooks provide a formal definition and move the equivalence between regular expressions and finite-state automatons
- Using `FSM`:
  1. Formal definition: type instance in a PL
  2. Examples: examples are executable programs

Related Work

- More in-depth treatment motivate regular expressions as a finite representation that may be used to describe infinite languages
  - Examples
  - Discuss properties: identity properties and simplification
  - Word generation

Related Work

- More in-depth treatment motivate regular expressions as a finite representation that may be used to describe infinite languages
    - Examples
    - Discuss properties: identity properties and simplification
    - Word generation
- Using `FSM`:
    - Simplification properties less emphasized
    - Examples purposely lead to an algorithm and its implementation for generating words in a the language
    - Embraces that randomness (i.e., nondeterminism) has its role in computation
    - Property-based unit testing to validate any generated word

Related Work

- Elaine Rich: More algorithmic, but only pseudo-code
- Word generation is discussed
- Think of any expression that is enclosed in a Kleene star as a loop

Related Work

- Elaine Rich: More algorithmic, but only pseudo-code
- Word generation is discussed
- Think of any expression that is enclosed in a Kleene star as a loop
- Using `FSM`:
  - Focuses in algorithms <u>and</u> implementation
  - Word-generating function is fully implemented based on the experience students gain from implementing regular expressions
  - Students walk away understanding how to design and implement a word-generating function for any given regular expression

Regular Expressions in FSM

- `A regular expression is either:`
  1. `(empty-regexp)`
  2. `(singleton-regexp "a")`, where a∈Σ
  3. `(union-regexp r1 r2)`, where r1 and r2 are regular expressions
  4. `(concat-regexp r1 r2)`, where r1 and r2 are regular expressions
  5. `(kleenestar-regexp r)`, where r is a regular expression

Regular Expressions in FSM

- A regular expression is either:
  1. (empty-regexp)
  2. (singleton-regexp "a"), where a∈Σ
  3. (union-regexp r1 r2), where r1 and r2 are regular expressions
  4. (concat-regexp r1 r2), where r1 and r2 are regular expressions
  5. (kleenestar-regexp r), where r is a regular expression
- Tailor-made error messaging:

```
> (union-regexp 2 (singleton-regexp 'w))
the input to the regexp #(struct:singleton-regexp w) must be a stri
> (union-regexp (empty-regexp) 3)
3 must be a regexp to be a valid second input to union-regexp
> (concat-regexp 3 (empty-regexp))
3 must be a regexp to be a valid first input to concat-regexp 3
> (kleenestar-regexp "A U B")
"A U B" must be a regexp to be a valid input to kleenestar-regexp
```

Regular Expressions in FSM

- A regular expression is either:
  1. `(empty-regexp)`
  2. `(singleton-regexp "a")`, where a∈Σ
  3. `(union-regexp r1 r2)`, where r1 and r2 are regular expressions
  4. `(concat-regexp r1 r2)`, where r1 and r2 are regular expressions
  5. `(kleenestar-regexp r)`, where r is a regular expression
- Tailor-made error messaging:

```
> (union-regexp 2 (singleton-regexp 'w))
the input to the regexp #(struct:singleton-regexp w) must be a stri
> (union-regexp (empty-regexp) 3)
3 must be a regexp to be a valid second input to union-regexp
> (concat-regexp 3 (empty-regexp))
3 must be a regexp to be a valid first input to concat-regexp 3
> (kleenestar-regexp "A U B")
"A U B" must be a regexp to be a valid input to kleenestar-regexp
```

- Printing:

```
> (printable-regexp (union-regexp (singleton-regexp "z")
                                  (union-regexp (singleton-regexp "
                                                (singleton-regexp "
"(z U (1 U q))"
> (printable-regexp (kleenestar-regexp
                     (concat-regexp (singleton-regexp "a")
                                    (singleton-regexp "b"))))
"(ab)*"
```

Regular Expressions in FSM

- The FSM selector functions for sub regular expressions are:

Regular Expressions in FSM

- The FSM selector functions for sub regular expressions are:
- Predicates:

```
empty-regexp?     singleton-regexp?     kleenestar-regexp?
union-regexp?     concat-regexp?
```

Regular Expressions in FSM

- The FSM selector functions for sub regular expressions are:
- Predicates:

  `empty-regexp?` `singleton-regexp?` `kleenestar-regexp?`
  `union-regexp?` `concat-regexp?`

- Function Template:

```
;; regexp ... → ...
;; Purpose: ...
(define (f-on-regexp rexp ...)
  (cond [(empty-regexp? rexp) ...]
        [(singleton-regexp? rexp) ...(singleton-regexp-a rexp)...]
        [(kleenestar-regexp? rexp)
         ...(f-on-regexp (kleenestar-regexp-r1 rexp))...]
        [(union-regexp? rexp)
         ...(f-on-regexp (union-regexp-r1 rexp))...
         ...(f-on-regexp (union-regexp-r2 rexp))...]
        [else ...(f-on-regexp (concat-regexp-r1 rexp))...
              ...(f-on-regexp (concat-regexp-r2 rexp))...]))
```

Programming with Regular Expressions

Binary Numbers

- `BIN-NUMS = {w | w is a binary number without leading zeroes}`

Programming with Regular Expressions

Binary Numbers

- `BIN-NUMS = {w | w is a binary number without leading zeroes}`
- 1. `Σ = {0 1}`
  2. The minimum length of a binary number is 1
  3. A binary number with a length greater than 1 cannot start with 0

Programming with Regular Expressions

Binary Numbers

- `BIN-NUMS = {w | w is a binary number without leading zeroes}`
- 1. `Σ = {0 1}`
  2. The minimum length of a binary number is 1
  3. A binary number with a length greater than 1 cannot start with 0
- ```
  (define ZERO (singleton-regexp "0"))
  (define ONE  (singleton-regexp "1"))
  ```
  ```

Programming with Regular Expressions

Binary Numbers

- `BIN-NUMS = {w | w is a binary number without leading zeroes}`
- 1. Σ = {0 1}
  2. The minimum length of a binary number is 1
  3. A binary number with a length greater than 1 cannot start with 0
- ```
  (define ZERO (singleton-regexp "0"))
  (define ONE  (singleton-regexp "1"))
  ```
- ```
  (define 0U1* (kleenestar-regexp (union-regexp ZERO ONE)))
  ```

Programming with Regular Expressions

Binary Numbers

- `BIN-NUMS = {w | w is a binary number without leading zeroes}`
- 1. Σ = {0 1}
  2. The minimum length of a binary number is 1
  3. A binary number with a length greater than 1 cannot start with 0
- ```
  (define ZERO (singleton-regexp "0"))
  (define ONE  (singleton-regexp "1"))
  ```
- ```
  (define 0U1* (kleenestar-regexp (union-regexp ZERO ONE)))
  ```
- ```
  (define STARTS1 (concat-regexp ONE 0U1*))
  ```
  ```

Programming with Regular Expressions

Binary Numbers

- `BIN-NUMS = {w | w is a binary number without leading zeroes}`
- 1. Σ = {0 1}
  2. The minimum length of a binary number is 1
  3. A binary number with a length greater than 1 cannot start with 0
- ```
  (define ZERO (singleton-regexp "0"))
  (define ONE  (singleton-regexp "1"))
  ```
- `(define 0U1* (kleenestar-regexp (union-regexp ZERO ONE)))`
- `(define STARTS1 (concat-regexp ONE 0U1*))`
- `(define BIN-NUMS (union-regexp ZERO STARTS1))`

  ```
  (check-equal? (printable-regexp BIN-NUMS) "(0 U 1(0 U 1)*)")
  ```
  ```

Programming with Regular Expressions

Generating BIN-NUMS

- Compare:

  BIN-NUMS = {w | w is a binary number without leading zeroes}

  $\rightarrow$ What is a bin num?

  BIN-NUMS = $(0 \cup 1(0 \cup 1)^*)$ $\rightarrow$ How to construct a bin num?

Programming with Regular Expressions

Generating BIN-NUMS

- Compare:

  BIN-NUMS = {w | w is a binary number without leading zeroes}
  → What is a bin num?

  BIN-NUMS = $(0 \cup 1(0 \cup 1)^*)$ → How to construct a bin num?
- DESIGN IDEA
- Simplify discussion: maximum length is 10
- Generate 0 with a 0.01 probability
- If 0 is not generated: first element is 1 and rest contains at most 9 binary digits
- Represent using a list

Programming with Regular Expressions

Generating BIN-NUMS

- Compare:

  BIN-NUMS = {w | w is a binary number without leading zeroes}
  → What is a bin num?

  BIN-NUMS = (0 ∪ 1(0 ∪ 1)*) → How to construct a bin num?
- DESIGN IDEA
- Simplify discussion: maximum length is 10
- Generate 0 with a 0.01 probability
- If 0 is not generated: first element is 1 and rest contains at most 9 binary digits
- Represent using a list
-
```
;;  → BIN-NUMS
;; Purpose: Generate a binary number without leading
;;          zeroes of length ≤ MAX-LENGTH
(define (generate-bn)

  (define MAX-LENGTH 10)
           ⋮
  ...)
```

Programming with Regular Expressions

Generating BIN-NUMS

- Tests
- Due to randomness, test that the generated words have the expected properties

Programming with Regular Expressions

Generating BIN-NUMS

- Tests
- Due to randomness, test that the generated words have the expected properties
- 1. `w` is a list
  2. $1 \leq$ `(length w)`
  3. `w` is `'(0)` or `(first w)` is 1
  4. `w` only contains 0s and 1s

Programming with Regular Expressions

Generating BIN-NUMS

- Tests
- Due to randomness, test that the generated words have the expected properties
- 1. w is a list
  2. $1 \leq$ (length w)
  3. w is '(0) or (first w) is 1
  4. w only contains 0s and 1s
-
```
;; word → Boolean
;; Purpose: Test if the given word is in L(BIN-NUMS)
(define (is-bin-nums? w)
 (and (list? w)
      (<= 1 (length w))
      (or (equal? w '(0)) (= (first w) 1))
      (andmap (λ (bit) (or (= bit 0) (= bit 1))) w)))

(check-equal? (is-bin-nums? '()) #f)
(check-equal? (is-bin-nums? '(0 0 0 1 1 0 1 0)) #f)
(check-equal? (is-bin-nums? '(0)) #t)
(check-equal? (is-bin-nums? '(1 0 0 1 0 1 1)) #t)
(check-equal? (is-bin-nums? '(1 1 1 0 1 0 0 0 1 1 0 1)) #t)
```

Programming with Regular Expressions

Generating BIN-NUMS

- ```
  (check-pred is-bin-nums? (generate-bn))
  (check-pred is-bin-nums? (generate-bn))
  (check-pred is-bin-nums? (generate-bn))
  (check-pred is-bin-nums? (generate-bn))
  (check-pred is-bin-nums? (generate-bn))
  ```
- Although the tests all look the same they are not the same test
- Recall that `(generate-bn)` is nondeterministic
  ```

Programming with Regular Expressions

Generating BIN-NUMS

- ```
  ;;  → BIN-NUMS
  ;; Purpose: Generate a binary number without leading zeroes of
  ;;          length <= MAX-LENGTH
  (define (generate-bn)
    (define MAX-LENGTH 10)
  ```
  ```

Programming with Regular Expressions

Generating BIN-NUMS

- ```
  ;;  → BIN-NUMS
  ;; Purpose: Generate a binary number without leading zeroes of
  ;;          length <= MAX-LENGTH
  (define (generate-bn)
     (define MAX-LENGTH 10)
  ```

- ```
   (if (< (random) 0.01)
       (list 0)
       (cons 1 (generate-0U1* (random MAX-LENGTH))))
  ```

Programming with Regular Expressions

Generating BIN-NUMS

- ```
  ;;  → BIN-NUMS
  ;; Purpose: Generate a binary number without leading zeroes of
  ;;          length <= MAX-LENGTH
  (define (generate-bn)
     (define MAX-LENGTH 10)
  ```

- ```
     ;; natnum → BIN-NUMS
     ;; Purpose: Generate a random word of bits of the given length
     (define (generate-0U1* n)
       (if (= n 0)
           '()
           (cons (generate-bit) (generate-0U1* (sub1 n)))))
  ```

- ```
   (if (< (random) 0.01)
       (list 0)
       (cons 1 (generate-0U1* (random MAX-LENGTH))))
  ```
  ```

Programming with Regular Expressions

Generating BIN-NUMS

- ```
  ;;  → BIN-NUMS
  ;; Purpose: Generate a binary number without leading zeroes of
  ;;          length <= MAX-LENGTH
  (define (generate-bn)
     (define MAX-LENGTH 10)
  ```
- ```
     ;;  → bit
     ;; Purpose: Generate a random bit
     (define (generate-bit) (if (< (random) 0.5) 0 1))
  ```
- ```
     ;; natnum → BIN-NUMS
     ;; Purpose: Generate a random word of bits of the given length
     (define (generate-0U1* n)
       (if (= n 0)
           '()
           (cons (generate-bit) (generate-0U1* (sub1 n)))))
  ```
- ```
   (if (< (random) 0.01)
       (list 0)
       (cons 1 (generate-0U1* (random MAX-LENGTH)))))
  ```

Generating Words

- Generalize to generate an arbitrary word in the language of an arbitrary regular expression
- To simplify: a constant is defined for the maximum number of repetitions when generating a word from a `kleenestar-regexp`

```
(define MAX-KLEENESTAR-REPS 20)
```

Generating Words

-
```
;; regexp → word  Purpose: Generate random using given regexp
(define (gen-regexp-word rexp)
  (cond [(empty-regexp? rexp) EMP]
```

Generating Words

- ```
  ;; regexp → word  Purpose: Generate random using given regexp
  (define (gen-regexp-word rexp)
    (cond [(empty-regexp? rexp) EMP]
  ```
- ```
          [(singleton-regexp? rexp)
           (let [(element (singleton-regexp-a rexp))]
             (if (not (string<=? "0" element "9"))
                 (list (string->symbol element))
                 (list (string->number element))))]
  ```

Generating Words

```
• ;; regexp → word  Purpose: Generate random using given regexp
  (define (gen-regexp-word rexp)
    (cond [(empty-regexp? rexp) EMP]
•         [(singleton-regexp? rexp)
           (let [(element (singleton-regexp-a rexp))]
             (if (not (string<=? "0" element "9"))
                 (list (string->symbol element))
                 (list (string->number element))))]
•         [(kleenestar-regexp? rexp)
           (let* [(reps (random (add1 MAX-KLEENESTAR-REPS)))
                  (element-list
                    (flatten
                     (build-list
                      reps
                      (λ (i)
                       (gen-regexp-word (kleenestar-regexp-r1 rexp))))))]
             (if (empty? element-list) EMP element-list))]
```

Generating Words

```
;; regexp → word  Purpose: Generate random using given regexp
(define (gen-regexp-word rexp)
  (cond [(empty-regexp? rexp) EMP]
        [(singleton-regexp? rexp)
         (let [(element (singleton-regexp-a rexp))]
           (if (not (string<=? "0" element "9"))
               (list (string->symbol element))
               (list (string->number element))))]
        [(kleenestar-regexp? rexp)
         (let* [(reps (random (add1 MAX-KLEENESTAR-REPS)))
                (element-list
                  (flatten
                   (build-list
                    reps
                    (λ (i)
                      (gen-regexp-word (kleenestar-regexp-r1 rexp))
           (if (empty? element-list) EMP element-list))]
        [(union-regexp? rexp)
         (let* [(uregexps (extract-union-regexps rexp))
                (chosen (list-ref uregexps (random (length uregexps
           (gen-regexp-word chosen))]
```

Generating Words

```
;; regexp → word  Purpose: Generate random using given regexp
(define (gen-regexp-word rexp)
  (cond [(empty-regexp? rexp) EMP]
        [(singleton-regexp? rexp)
         (let [(element (singleton-regexp-a rexp))]
           (if (not (string<=? "0" element "9"))
               (list (string->symbol element))
               (list (string->number element))))]
        [(kleenestar-regexp? rexp)
         (let* [(reps (random (add1 MAX-KLEENESTAR-REPS)))
                (element-list
                  (flatten
                   (build-list
                    reps
                    (λ (i)
                     (gen-regexp-word (kleenestar-regexp-r1 rexp)))
           (if (empty? element-list) EMP element-list))]
        [(union-regexp? rexp)
         (let* [(uregexps (extract-union-regexps rexp))
                (chosen (list-ref uregexps (random (length uregexps
           (gen-regexp-word chosen))]
        [else (let [(cregexps (extract-concat-regexps rexp))]
                (filter (λ (w) (not (eq? w EMP)))
                        (flatten (map gen-regexp-word cregexps))))]
```

Regular Expression Applications

- To illustrate the use of regular expressions we explore the problem of generating passwords

Regular Expression Applications

- To illustrate the use of regular expressions we explore the problem of generating passwords
- A password is a string that:
  - Has length ≥ 10
  - Includes at least one of each: lowercase letter, uppercase letter, and special character (i.e., \$, &, !, and *)

Regular Expression Applications

- To illustrate the use of regular expressions we explore the problem of generating passwords
- A password is a string that:
  - Has length ≥ 10
  - Includes at least one of each: lowercase letter, uppercase letter, and special character (i.e., $, &, !, and *)
- Based on this definition, the sets for lowercase letters, uppercase letters, and special characters are defined as follows:

```
(define lowers '(a b c d e f g h i j k l m n o p q r s t u v w
(define uppers '(A B C D E F G H I J K L M N O P Q R S T U V W
(define spcls  '($ & ! *))
```

- The corresponding sets of regular expressions are defined as:

```
(define lc (map (λ (lcl) (singleton-regexp (symbol->string lcl)))
(define uc (map (λ (ucl) (singleton-regexp (symbol->string ucl)))
(define spc (map (λ (sc) (singleton-regexp (symbol->string sc)))
```

Regular Expression Applications

- There are six different orderings these required elements may appear in (with arbitrary elements in between)

  `L U S    U L S    S U L    L S U    U S L    S L U`

- Each defines a language

Regular Expression Applications

- There are six different orderings these required elements may appear in (with arbitrary elements in between)

  `L U S    U L S    S U L    L S U    U S L    S L U`

- Each defines a language
- Union regular expression needed for each group of elements:

```
(define LOWER (create-union-regexp lc))
(define UPPER (create-union-regexp uc))
(define SPCHS (create-union-regexp spc))
(define ARBTRY (kleenestar-regexp
                 (union-regexp LOWER (union-regexp UPPER SPCHS))))
```

Regular Expression Applications

- Regular expressions for each of the six languages:

```
(define LUS (concat-regexp
              ARBTRY
              (concat-regexp
                LOWER
                (concat-regexp
                  ARBTRY
                  (concat-regexp
                    UPPER
                    (concat-regexp ARBTRY
                                   (concat-regexp SPCHS ARBTRY)))))))

(define LSU (concat-regexp
              ARBTRY
              (concat-regexp
                LOWER
                (concat-regexp
                  ARBTRY
                  (concat-regexp
                    SPCHS
                    (concat-regexp ARBTRY
                                   (concat-regexp UPPER ARBTRY)))))))
```

⋮

Regular Expression Applications

- The language of passwords is a word in any of the languages defined for the different orderings of required elements:

```
(define PASSWD (union-regexp
                 LUS
                 (union-regexp
                   LSU
                   (union-regexp
                     SLU
                     (union-regexp SUL
                                   (union-regexp USL ULS))))))
```

Regular Expression Applications

- The constructor for a password takes no input and returns a string

Regular Expression Applications

- The constructor for a password takes no input and returns a string
- A word is generated by applying `gen-regexp-word` to `PASSWD` and converted to a string

Regular Expression Applications

- The constructor for a password takes no input and returns a string
- A word is generated by applying `gen-regexp-word` to `PASSWD` and converted to a string
- If the length of the string is greater than or equal to 10 then it is returned as the generated password. Otherwise, a new password is generated.

Regular Expression Applications

- ```
  ;;  → string
  ;; Purpose: Generate a valid password
  (define (generate-password)
    (let [(new-passwd (passwd->string (gen-regexp-word PASSWD)))]
      (if (>= (string-length new-passwd) 10)
          new-passwd
          (generate-password))))
  ```
  ```

Regular Expression Applications

- ```
  ;;  → string
  ;; Purpose: Generate a valid password
  (define (generate-password)
    (let [(new-passwd (passwd->string (gen-regexp-word PASSWD)))]
      (if (>= (string-length new-passwd) 10)
          new-passwd
          (generate-password))))
  ```
- ```
  ;; string → Boolean
  ;; Purpose: Test if the given string is a valid password
  (define (is-passwd? p)
    (let [(los (str->los p))]
      (and (>= (length los) 10)
           (ormap (λ (c) (member c los)) lowers)
           (ormap (λ (c) (member c los)) uppers)
           (ormap (λ (c) (member c los)) spcls))))
  ```

Regular Expression Applications

- ```
  ;;  → string
  ;; Purpose: Generate a valid password
  (define (generate-password)
    (let [(new-passwd (passwd->string (gen-regexp-word PASSWD)))]
      (if (>= (string-length new-passwd) 10)
          new-passwd
          (generate-password))))
  ```
- ```
  ;; string → Boolean
  ;; Purpose: Test if the given string is a valid password
  (define (is-passwd? p)
    (let [(los (str->los p))]
      (and (>= (length los) 10)
           (ormap (λ (c) (member c los)) lowers)
           (ormap (λ (c) (member c los)) uppers)
           (ormap (λ (c) (member c los)) spcls))))
  ```
- ```
  (check-pred is-passwd? (generate-password))
  (check-pred is-passwd? (generate-password))
  (check-pred is-passwd? (generate-password))
  (check-pred is-passwd? (generate-password))
  (check-pred is-passwd? (generate-password))
  ```
  ```

Regular Expression Applications

- The students run the program and confirm that all the tests pass

Regular Expression Applications

- The students run the program and confirm that all the tests pass
- Students are encouraged to generate a few passwords:

```
> (generate-password)
"&&!$m*F!&$*"
> (generate-password)
"!e*e!*oS!lq$"
> (generate-password)
"!y*$r!C&*d$"
> (generate-password)
"&&!p$rUA$*"
> (generate-password)
"W&*!eKY**D"
> (generate-password)
"vxY*We!Wx*&&u"
```

- Students feel a sense of accomplishment seeing the results: robust passwords

Concluding Remarks

- Didactic approach for introducing students to regular expressions
- Work presented emphasizes algorithm design and implementation to keep Computer Science students motivated and engaged
- Most students comment that the password-generating approach is like nothing they had thought about before

Concluding Remarks

- Didactic approach for introducing students to regular expressions
- Work presented emphasizes algorithm design and implementation to keep Computer Science students motivated and engaged
- Most students comment that the password-generating approach is like nothing they had thought about before
- Future work will address creating a database of examples instructors and students may draw upon for practice or presentation
- The goal is to have a diverse set of examples
- In addition, extensions to `FSM` are being considered: a primitive to generate words in the language of a given regular expression?

Concluding Remarks

- Didactic approach for introducing students to regular expressions
- Work presented emphasizes algorithm design and implementation to keep Computer Science students motivated and engaged
- Most students comment that the password-generating approach is like nothing they had thought about before
- Future work will address creating a database of examples instructors and students may draw upon for practice or presentation
- The goal is to have a diverse set of examples
- In addition, extensions to `FSM` are being considered: a primitive to generate words in the language of a given regular expression?

- Thank you! Any questions?