Programming-Based Automata Theory

Marco T. Morazán

Seton Hall University

Quién soy?

Formal (fan of McCarthy to Felleisen)

```
Professor of Computer Science
Seton Hall University
```

Quién soy?

Formal (fan of McCarthy to Felleisen)

    Professor of Computer Science
    Seton Hall University

Service

    Chair   TFP 2026, Odense Denmark
    Steering Committee
    Trends in Functional Programming

    Founder
    Trends in Functional Programming in Education

    Member
    Steering Committee
    Implementation and Applications of Functional Languages

Quién soy?

Formal (fan of McCarthy to Felleisen)

    Professor of Computer Science
    Seton Hall University

Service

    Chair   TFP 2026, Odense Denmark
    Steering Committee
    Trends in Functional Programming

    Founder
    Trends in Functional Programming in Education

    Member
    Steering Committee
    Implementation and Applications of Functional Languages

Some Interests

    Program transformations
      Optimal Lambda Lifting (IFL 2007)
      Memoized Bytecode Closures (TFP 2013)

Quién soy?

Formal (fan of McCarthy to Felleisen)

Professor of Computer Science
Seton Hall University

Service

Chair TFP 2026, Odense Denmark
Steering Committee
Trends in Functional Programming

Founder
Trends in Functional Programming in Education

Member
Steering Committee
Implementation and Applications of Functional Languages

Some Interests

Program transformations
- Optimal Lambda Lifting (IFL 2007)
- Memoized Bytecode Closures (TFP 2013)

Functional Programming in Education
- CS1 (*Animated Problem Solving*, Springer)
- CS2 (*Animated Program Design*, Springer)
- Automata Theory (*Programming-Based Formal Languages and Automata Theory*, Springer)
- AI-Assisted Program Design (come to my IFL 2025 talk)

Quién soy?

Formal (fan of McCarthy to Felleisen)

- Professor of Computer Science
- Seton Hall University

Service

- Chair TFP 2026, Odense Denmark
- Steering Committee
- Trends in Functional Programming

- Founder
- Trends in Functional Programming in Education

- Member
- Steering Committee
- Implementation and Applications of Functional Languages

Some Interests

- Program transformations
  - Optimal Lambda Lifting (IFL 2007)
  - Memoized Bytecode Closures (TFP 2013)
- Functional Programming in Education
  - CS1 (*Animated Problem Solving*, Springer)
  - CS2 (*Animated Program Design*, Springer)
  - Automata Theory (*Programming-Based Formal Languages and Automata Theory*, Springer)
  - AI-Assisted Program Design (come to my IFL 2025 talk)
- DSLs
  - FSMt (come Andrés' IFL 2025 talk)

Quién soy?

Formal (fan of McCarthy to Felleisen)

Professor of Computer Science
Seton Hall University

Service

Chair   TFP 2026, Odense Denmark
Steering Committee
Trends in Functional Programming

Founder
Trends in Functional Programming in Education

Member
Steering Committee
Implementation and Applications of Functional Languages

Some Interests

- Program transformations
  - Optimal Lambda Lifting (IFL 2007)
  - Memoized Bytecode Closures (TFP 2013)
- Functional Programming in Education
  - CS1 (*Animated Problem Solving*, Springer)
  - CS2 (*Animated Program Design*, Springer)
  - Automata Theory (*Programming-Based Formal Languages and Automata Theory*, Springer)
  - AI-Assisted Program Design (come to my IFL 2025 talk)
- DSLs
  - FSMt (come Andrés' IFL 2025 talk)
- Visualizations
  - Too many to list
  - Unrestricted Grammar Derivation (come to Andrés' IFL 2025 talk)

Quién soy?

Formal (fan of McCarthy to Felleisen)

Professor of Computer Science
Seton Hall University

Service

Chair TFP 2026, Odense Denmark
Steering Committee
Trends in Functional Programming

Founder
Trends in Functional Programming in Education

Member
Steering Committee
Implementation and Applications of Functional Languages

Some Interests

- Program transformations
  - Optimal Lambda Lifting (IFL 2007)
  - Memoized Bytecode Closures (TFP 2013)
- Functional Programming in Education
  - CS1 (*Animated Problem Solving*, Springer)
  - CS2 (*Animated Program Design*, Springer)
  - Automata Theory (*Programming-Based Formal Languages and Automata Theory*, Springer)
  - AI-Assisted Program Design (come to my IFL 2025 talk)
- DSLs
  - FSMt (come Andrés' IFL 2025 talk)
- Visualizations
  - Too many to list
  - Unrestricted Grammar Derivation (come to Andrés' IFL 2025 talk)
- Validation
  - Automatic FSA Testing (come to Sophia's IFL 2025 talk)

Quién soy?

Formal (fan of McCarthy to Felleisen)

- Professor of Computer Science
- Seton Hall University

Service

- Chair — TFP 2026, Odense Denmark
- Steering Committee
- Trends in Functional Programming

- Founder
- Trends in Functional Programming in Education

- Member
- Steering Committee
- Implementation and Applications of Functional Languages

Some Interests

- Program transformations
  - Optimal Lambda Lifting (IFL 2007)
  - Memoized Bytecode Closures (TFP 2013)
- Functional Programming in Education
  - CS1 (*Animated Problem Solving*, Springer)
  - CS2 (*Animated Program Design*, Springer)
  - Automata Theory (*Programming-Based Formal Languages and Automata Theory*, Springer)
  - AI-Assisted Program Design (come to my IFL 2025 talk)
- DSLs
  - FSMt (come Andrés' IFL 2025 talk)
- Visualizations
  - Too many to list
  - Unrestricted Grammar Derivation (come to Andrés' IFL 2025 talk)
- Validation
  - Automatic FSA Testing (come to Sophia's IFL 2025 talk)
- Error Messages
  - Recipe-Based Errors (come to David's and Shamil's IFL 2025 talk)
- Etc.

Tutorial Outline

Regular Languages

- Regular expressions
- Deterministic Finite Automata
- Nondeterministic Finite Automata

Tutorial Outline

- Regular Languages
  - Regular expressions
  - Deterministic Finite Automata
  - Nondeterministic Finite Automata
- Context-Free Languages
  - Context-free Grammars
  - Pushdown Automata
  - Context-Free Expressions (look for papers soon)

Tutorial Outline

- Regular Languages
  - Regular expressions
  - Deterministic Finite Automata
  - Nondeterministic Finite Automata
- Context-Free Languages
  - Context-free Grammars
  - Pushdown Automata
  - Context-Free Expressions (look for papers soon)
- Recursively Enumerable Languages
  - Turing Machines
  - Multitape Turing Machines
  - Unrestricted Grammars

Course Outline

Domain-Specific Language: FSM

Design Implement Validate Verify

Course Outline

Domain-Specific Language: `FSM`

Design Implement Validate Verify

Systematic Design Recipes

Validation (Unit Testing and Invariant Testing)

Verification

Recipe-Based Errors

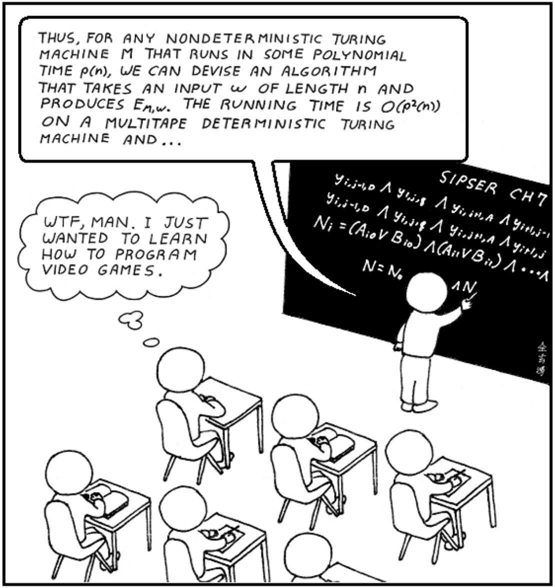
Textbook: *Programming-Based Formal Languages and Automata Theory*

Motivation

Why would you want to do programming-based CS theory?

Motivation

Why would you want to do programming-based CS theory?



Motivation

CS students dislike

- Mathematical nature
- Theory
- Formal Notation
- Lack of programming
- Proofs they get wrong!

Motivation

- CS students dislike
  - Mathematical nature
  - Theory
  - Formal Notation
  - Lack of programming
  - Proofs they get wrong!
- Constructivism in CS
  - knowledge is actively constructed by students engaged in building activities
  - Common denominator: interest in software development
  - Tools: visualization, tutoring, simulators
  - PLs: Few efforts, limited in scope: $\mathtt{P}^{\flat}$

Motivation

Need to integrate PLs and tools

Motivation

Need to integrate PLs and tools

`FSM`: Functional State Machines

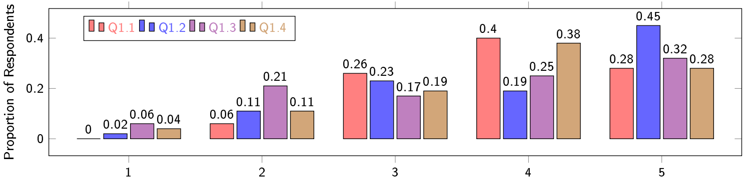
- A DSL in `Racket` that is a home for all
- Define
- Validate
- Visualize
- Verify

Motivation

- Need to integrate PLs and tools
- `FSM`: Functional State Machines
  - A DSL in `Racket` that is a home for all
  - Define
  - Validate
  - Visualize
  - Verify
- Tools are not enough
  - Systematic development: design recipes
  - Textbook: *Programming-Based Formal Languages and Automata Theory*
  - Support for validation and verification
  - Visualization: Norman Principles of Effective Design
  - Development of domain jargon

Motivation

Course Interest

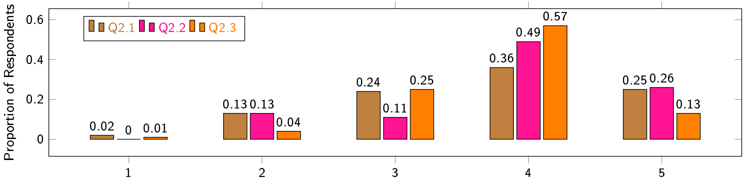


Survey Statements

- Q1.1 This course is interesting.
- Q1.2 Programming helped understand the material.
- Q1.3 Programming increased my interest in Formal Languages and Automata Theory.
- Q1.4 The course is relevant to my Computer Science education.

Motivation

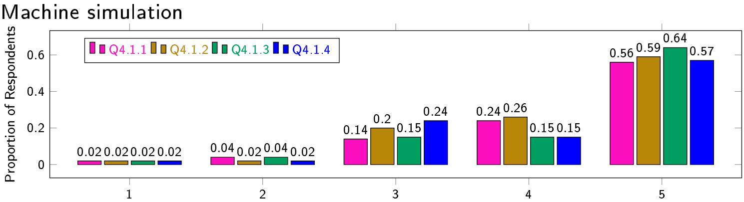
Intellectual Stimulation



Survey Statements

Q2.1 Automata Theory is intellectually stimulating.
Q2.2 Programming state machines grammars, and regular expressions is intellectually stimulating.
Q2.3 Programming constructive algorithms is intellectually stimulating.

Motivation



Q4.1.1	Visualizing the execution of deterministic finite state automata is useful.
Q4.1.2	Visualizing the execution of nondeterministic finite state automata is useful.
Q4.1.3	Visualizing the execution of pushdown automata is useful.
Q4.1.4	Visualizing the execution of Turing machines is useful.

Motivation

Data Definitions

```
A state/nonterminal is in [A--Z]

An alphabet, Σ, is in (setof [a--z])∪[0--9]

A word is either
  1. EMP
  2. (listof i), i∈Σ
```

Regular Expressions

A regular expression, over an alphabet Σ, is an FSM type instance:

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
6. (null-regexp) Not in our focus today

Regular Expressions

A regular expression, over an alphabet Σ, is an `FSM` type instance:

1. `(empty-regexp)`
2. `(singleton-regexp "a")`, where a$\in \Sigma$
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
6. `(null-regexp)` Not in our focus today

`L(r)` = language of r

A language that is described by a regular expression is called a *regular language*.

Regular Expressions

Selectors

`singleton-regexp-a`	`kleenestar-regexp-r1`
`union-regexp-r1`	`union-regexp-r2`
`concat-regexp-r1`	`concat-regexp-r2`

Regular Expressions

Selectors

```
singleton-regexp-a     kleenestar-regexp-r1

union-regexp-r1        union-regexp-r2

concat-regexp-r1       concat-regexp-r2
```

Predicates

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

Regular Expressions

Selectors

singleton-regexp-a kleenestar-regexp-r1

union-regexp-r1 union-regexp-r2

concat-regexp-r1 concat-regexp-r2

Predicates

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

Function Template

```
;; regexp ... → ...
;; Purpose: ...
(define (f—on—regexp rexp ...)
  (cond [(empty—regexp? rexp) ...]
        [(singleton—regexp? rexp)
         ...(f—on—string (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))...]))
```

Regular Expressions

More observers

gen-regexp-word: Nondeterministically generates a word in the language of the given `regexp`

Regular Expressions

More observers

gen-regexp-word: Nondeterministically generates a word in the language of the given `regexp`

printable-regexp: Transforms the given r to a string

Regular Expressions

Design Recipe for Regular Expressions

1. Identify the input alphabet, pick a name for the regular expression, and describe the language
2. Identify the sublanguages and outline how to compose them
3. Define a predicate to determine if a word is in the target language
4. Write unit tests
5. Define the regular expression
6. Run the tests and, if necessary, debug by revisiting the previous steps
7. Prove that the regular expression is correct

Regular Expressions

Problem

DNA: arbitrary number of four nucleotide bases:
adenine (a) guanine (g) cytosine(c) thymine (t)
Certain genetic disorders, such as Huntington's disease, are characterized by containing the subsequence cag repeated two or more times in a row.

To help test programs written to detect this disorder, it is useful to generate DNA sequences that contain such a subsequence. Design and implement a regular expression to generate DNA sequences with cag repeated two or more times in a row.

Regular Expressions

Step 3:

```
;; L={w | cagcag∈w}
(define DISORDER-DNA
 (let* [(SIGMA '(a c g t))
        ...]




   #:sigma SIGMA
```

Regular Expressions

Step 3:

```
;; L={w | cagcag∈w}
(define DISORDER-DNA
 (let* [(SIGMA '(a c g t))
        ...]



   #:sigma SIGMA


   #:pred in-DISORDER-DNA?
```

Regular Expressions

Step 3:

```
;; L={w | cagcag∈w}
(define DISORDER-DNA
 (let* [(SIGMA '(a c g t))
        ...]




   #:sigma SIGMA


   #:pred in-DISORDER-DNA?


   #:gen-cases 10
   #:in-lang '((c a g c a g) (g t c a g c a g t g))
   #:not-in-lang '((c g a t) (g t c a g a t)))
```

Regular Expressions

Step 3:

```
;; L={w | cagcag∈w}
(define DISORDER-DNA
 (let* [(SIGMA '(a c g t))
        ...]

  (concat-regexp DNA (concat-regexp CAG++ DNA)

   #:sigma SIGMA

   #:pred in-DISORDER-DNA?

   #:gen-cases 10
   #:in-lang '((c a g c a g) (g t c a g c a g t g))
   #:not-in-lang '((c g a t) (g t c a g a t)))
```

Regular Expressions

```
;; L=BASE*
(DNA (kleenestar-regexp BASE
        #:sigma SIGMA
        #:pred
          (λ (w)
            (andmap (λ (s) (list? (member s SIGMA)))
                    (word2lst w)))
        #:gen-cases 10
        #:in-lang '((g) () (t g g) (c g a) (a a t))
        #:not-in-lang '()))
```

Regular Expressions

```
;; L={a} U {c} U {g} U {t}
(BASE
  (union-regexp
    A
    (union-regexp C (union-regexp G T))
    #:sigma SIGMA
    #:pred (λ (w)
             (and (not-EMP? w)
                  (= (length w) 1)
                  (list? (member (first w) SIGMA))))
    #:gen-cases 10
    #:in-lang '((g) (t) (a) (c))
    #:not-in-lang '((a a t) (g a t c))))
```

Regular Expressions

```
;; L=CAG CAG+
(CAG++ (concat-regexp CAG CAG+
          #:sigma '(c a g)
          #:pred (λ (w)
                   (and (not-EMP? w)
                        (>= (length w) 6)
                        (lst-of-cag? w)))
          #:gen-cases 10
          #:in-lang '((c a g c a g c a g))
          #:not-in-lang '((a a c a g c g g))))
```

Regular Expressions

```
;; L=CAG+
(CAG+ (concat-regexp CAG CAG*
        #:sigma '(c a g)
        #:pred (λ (w)
                 (and (not-EMP? w)
                      (>= (length w) 3)
                      (lst-of-cag? w)))
        #:gen-cases 10
        #:in-lang '((c a g))
        #:not-in-lang '((c a g c g g))))
```

Regular Expressions

```
;; L=CAG*
(CAG* (kleenestar-regexp CAG
        #:sigma '(c a g)
        #:pred lst-of-cag?
        #:gen-cases 10
        #:in-lang '(() (c a g) (c a g c a g))
        #:not-in-lang '((g g g) (a c g c a a))))
```

Regular Expressions

```
;; L={(c a g)}
(CAG (concat-regexp C (concat-regexp A G)
       #:sigma '(c a g)
       #:pred (λ (w)
                (and (not-EMP? w)
                     (equal? w '(c a g))))
       #:gen-cases 1
       #:in-lang '((c a g))
       #:not-in-lang '((c a g c a g))))
```

Regular Expressions

```
;; L={c}
(C (singleton-regexp "c"))

;; L={a}
(A (singleton-regexp "a"))

;; L={g}
(G (singleton-regexp "g"))

;; L={t}
(T (singleton-regexp "t"))
```

Regular Expressions

Document for code readers

```
;;word → boolean
;;Purpose: Determines if the given word is in
;;        L(DISORDER−DNA)
(define (in-DISORDER-DNA? w)
  (let [(L (word2lst w))]
    (and (not (empty? L))
         (>= (length L) 6)
         (or (equal? (take L 6) '(c a g c a g))
             (in-DISORDER-DNA? (rest L))))))
```

Regular Expressions

Document for code readers

```
;;word → boolean
;;Purpose: Determines if the given word is in
;;          L(DISORDER−DNA)
(define (in-DISORDER-DNA? w)
  (let [(L (word2lst w))]
    (and (not (empty? L))
         (>= (length L) 6)
         (or (equal? (take L 6) '(c a g c a g))
             (in-DISORDER-DNA? (rest L))))))

;; word → Boolean
;; Purpose: Determine if given list is (c a g)*
(define (lst-of-cag? w)
  (let [(L (word2lst w))]
    (or (empty? L)
        (and (= (remainder (length L) 3) 0)
             (equal? (take L 3) '(c a g))
             (lst-of-cag? (drop L 3))))))
```

Regular Expressions

```
;; word → (listof base)
;; Purpose: Convert given word to a list
(define (word2lst w)
  (if (eq?  w EMP) '() w))

;; word → Boolean
;; Purpose: Determine that given word is not empty
(define (not-EMP? w) (not (eq? w EMP)))
```

Regular Expressions

Document for code readers

```
(check-gen? in-DISORDER-DNA?
            '(c a g c a g)
            '(a g g t c c a g c a g t a g))
(check-not-gen? in-DISORDER-DNA?
                '()
                '(a c g t)
                '(c a t c c a a)
                '(t g a c a g t a g))
(for-each (λ (w)
            (check-gen? in-DISORDER-DNA? w))
          (build-list
            20
            (λ (i)
              (gen-regexp-word DISORDER-DNA))))
```

Regular Expressions

Correctness

```
;; L={w | cagcag∈w}
(define DISORDER-DNA
  (concat-regexp DNA (concat-regexp CAG++ DNA)))
```

Regular Expressions

Correctness

```
;; L={w | cagcag∈w}
(define DISORDER-DNA
  (concat-regexp DNA (concat-regexp CAG++ DNA)))
```

Assume subexpressions are correct

Regular Expressions

Correctness

```
;; L={w | cagcag∈w}
(define DISORDER-DNA
  (concat-regexp DNA (concat-regexp CAG++ DNA)))
```

Assume subexpressions are correct

DNA generates an arbitrary dna strand
CAG++ generates 2 or more cag strands
Therefore, DISORDER-DNA generates an arbitrary strand with at least 2 cag strands

Regular Expressions

Correctness

```
;; L={w | cagcag∈w}
(define DISORDER-DNA
  (concat-regexp DNA (concat-regexp CAG++ DNA)))
```

Assume subexpressions are correct

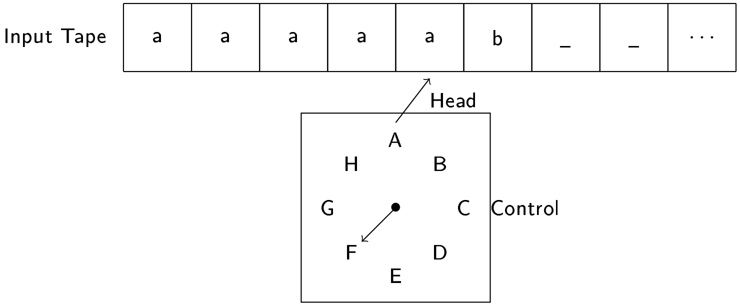
> DNA generates an arbitrary dna strand
> CAG++ generates 2 or more cag strands
> Therefore, DISORDER-DNA generates an arbitrary strand with at least 2 cag strands

Do the same for the subexpressions

Regular Expressions

Exercise: $L = \texttt{aa}(\texttt{ba} \cup \texttt{bb})^{*}\texttt{aa}$

Finite-State Machines



Finite-State Machines

Constructors

`make-dfa`:	K	Σ		s	F	Δ		$\rightarrow$	`dfa`
`make-ndfa`:	K	Σ		s	F	δ		$\rightarrow$	`ndfa`
`make-ndpda`:	K	Σ	Γ	s	F	δ		$\rightarrow$	`pda`
`make-tm`:	K	Σ	δ	s	F	[a]		$\rightarrow$	`tm`
`make-mttm`:	K	Σ	s	F	δ	n	[a]	$\rightarrow$	`mttm`

Finite-State Machines

Constructors

make-dfa:	K	Σ		s	F	Δ		→	dfa
make-ndfa:	K	Σ		s	F	δ		→	ndfa
make-ndpda:	K	Σ	Γ	s	F	δ		→	pda
make-tm:	K	Σ	δ	s	F	[a]		→	tm
make-mttm:	K	Σ	s	F	δ	n	[a]	→	mttm

Observers

`sm-states` `sm-sigma` `sm-start` `sm-finals` `sm-rules` `sm-gamma`

`sm-apply` `sm-showtransitions`

Finite-State Machines

Constructors

`make-dfa`:	K	Σ		s	F	Δ		→	`dfa`
`make-ndfa`:	K	Σ		s	F	δ		→	`ndfa`
`make-ndpda`:	K	Σ	Γ	s	F	δ		→	`pda`
`make-tm`:	K	Σ	δ	s	F	[a]		→	`tm`
`make-mttm`:	K	Σ	s	F	δ	n	[a]	→	`mttm`

Observers

`sm-states` `sm-sigma` `sm-start` `sm-finals` `sm-rules` `sm-gamma`

`sm-apply` `sm-showtransitions`

Visualizations

`sm-graph` `sm-cmpgraph`

`sm-viz` `ndfa2dfa-viz` `union-viz`
`concat-viz` `ndfa2regexp-viz` `regexp2ndfa-viz`

and more...

Finite-State Machines

Constructors

make-dfa:	K	Σ		s	F	Δ		→	dfa
make-ndfa:	K	Σ		s	F	δ		→	ndfa
make-ndpda:	K	Σ	Γ	s	F	δ		→	pda
make-tm:	K	Σ	δ	s	F	[a]		→	tm
make-mttm:	K	Σ	s	F	δ	n	[a]	→	mttm

Observers

sm-states sm-sigma sm-start sm-finals sm-rules sm-gamma

sm-apply sm-showtransitions

Visualizations

sm-graph sm-cmpgraph

sm-viz ndfa2dfa-viz union-viz
concat-viz ndfa2regexp-viz regexp2ndfa-viz

and more...

Testing

RBEs (David's and Shamil's talk @ IFL 2025)
FSMt (Andrés' talk @ IFL 2025)
Automatic (Sophia's talk @ IFL 2025)

Finite-State Machines

1. Name the machine and specify alphabets
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. Test the machine
7. Design, implement, and test an invariant predicate for each state
8. ProveL=L(M)

Finite-State Machines

;; $L = \{\epsilon\}$ U aa* U ab*

```
(define LNDFA
  (make-ndfa

    '(a b)
```

Finite-State Machines

```
;; L = {ε} U aa* U ab*

(define LNDFA
  (make-ndfa

    '(a b)







    #:rejects '((b a) (a b a) (a a b a))
    #:accepts '(() (a a) (a b b b b))))
(check-reject? LNDFA '(a b a) '(b b b b b)
                     '(a b b b b a a a))
(check-accept? LNDFA '() '(a) '(a a a a) '(a b b))
```

Finite-State Machines

```
;; L = {ϵ} U aa* U ab*
;; State Documentation
;;   S: ci = empty, starting state    A: ci = ab*, final state
;;   F: ci = empty, final state        B: ci = aa*, final state
(define LNDFA
  (make-ndfa
    '(S A B F)
    '(a b)
    'S
    '(A B F)




    #:rejects '((b a) (a b a) (a a b a))
    #:accepts '(() (a a) (a b b b b))))
(check-reject? LNDFA '(a b a) '(b b b b b)
                     '(a b b b b a a a))
(check-accept? LNDFA '() '(a) '(a a a a) '(a b b))
```

Finite-State Machines

```
;; L = {ϵ} U aa* U ab*
;; State Documentation
;;  S: ci = empty, starting state    A: ci = ab*, final state
;;  F: ci = empty, final state       B: ci = aa*, final state
(define LNDFA
  (make-ndfa
    '(S A B F)
    '(a b)
    'S
    '(A B F)
    `((S a A)  (S a B)  (S ,EMP F)
      (A b A)
      (B a B))
    #:rejects '((b a) (a b a) (a a b a))
    #:accepts '(() (a a) (a b b b b))))
(check-reject? LNDFA '(a b a) '(b b b b b)
                     '(a b b b b a a a))
(check-accept? LNDFA '() '(a) '(a a a a) '(a b b))
```

Finite-State Machines

```
;; word → Boolean
;; Purpose: Determine if the given word is empty
(define S-INV empty?)

;; word → Boolean
;; Purpose: Determine if the given word is empty
(define F-INV empty?)

;; word → Boolean
;; Purpose: Determine if the given word is in aa*
(define (B-INV ci)
  (and (not (empty? ci))
       (andmap (λ (s) (eq? s 'a)) ci)))

;; word → Boolean
;; Purpose: Determine if the given word is in ab*
(define (A-INV ci)
  (and (not (empty? ci))  (eq? (first ci) 'a)
       (andmap (λ (s) (eq? s 'b)) (rest ci))))
```

Finite-State Machines

* Correctness of M
    a. Prove the state invariants hold when M is applied to w∈L(M)
    b. Prove that L = L(M)
* This approach of for `dfas`, `ndfas`, and `pdas`

Finite-State Machines

Theorem

State invariants hold when LDNFA is applied to w∈L(LNDFA).

Proof.

Proof by induction on n = the number of steps M performs to consume w.
Base Case: n = 0
If n is 0 then the consumed input must be `'()`and machine is in S. Clearly, `S-INV`. □

Finite-State Machines

Proof.

Inductive Step:
Assume: State invariants hold for n = k.
Show: State invariants hold for n = k+1.
Consider each transition: □

Finite-State Machines

Proof.

Inductive Step:
Assume: State invariants hold for n = k.
Show: State invariants hold for n = k+1.
Consider each transition: □

(S ,EMP F) By IH, S-INV holds. After consuming no input, ci=EMP. Thus, F-INV holds.

Finite-State Machines

Proof.

Inductive Step:
Assume: State invariants hold for n = k.
Show: State invariants hold for n = k+1.
Consider each transition: □

(S ,EMP F) By IH, S-INV holds. After consuming no input, ci=EMP. Thus, F-INV holds.
(S a A) By IH, S-INV holds, which means ci=EMP. After consuming a, ci=a. Thus, ci∈ab* and A-INV holds.

Finite-State Machines

Proof.

Inductive Step:
Assume: State invariants hold for n = k.
Show: State invariants hold for n = k+1.
Consider each transition: □

(S ,EMP F) By IH, S-INV holds. After consuming no input, ci=EMP. Thus, F-INV holds.
(S a A) By IH, S-INV holds, which means ci=EMP. After consuming a, ci=a. Thus, ci∈ab* and A-INV holds.
(S a B) By IH, S-INV holds, which means ci=EMP. After consuming a, ci∈aa* and B-INV holds.

Finite-State Machines

Proof.

Inductive Step:
Assume: State invariants hold for n = k.
Show: State invariants hold for n = k+1.
Consider each transition: □

(S ,EMP F) By IH, S-INV holds. After consuming no input, ci=EMP. Thus, F-INV holds.
(S a A) By IH, S-INV holds, which means ci=EMP. After consuming a, ci=a. Thus, ci∈ab* and A-INV holds.
(S a B) By IH, S-INV holds, which means ci=EMP. After consuming a, ci∈aa* and B-INV holds.
(A b A) By IH, A-INV holds, which means ci∈ab*. After consuming b, ci∈ab* and A-INV holds.

Finite-State Machines

Proof.

Inductive Step:
Assume: State invariants hold for n = k.
Show: State invariants hold for n = k+1.
Consider each transition: □

(S ,EMP F) By IH, S-INV holds. After consuming no input, ci=EMP. Thus, F-INV holds.
(S a A) By IH, S-INV holds, which means ci=EMP. After consuming a, ci=a. Thus, ci∈ab* and A-INV holds.
(S a B) By IH, S-INV holds, which means ci=EMP. After consuming a, ci∈aa* and B-INV holds.
(A b A) By IH, A-INV holds, which means ci∈ab*. After consuming b, ci∈ab* and A-INV holds.
(B a B) By IH, B-INV holds, which means ci∈aa*. After consuming a, ci∈aa* and B-INV holds.

Finite-State Machines

Theorem

$L = L(LNDFA)$

Finite-State Machines

Theorem

$L = L(LNDFA)$

<u>w∈L ⇒ w∈L(LNDFA)</u>

Assume w∈L. This means w=EMP ∨ w∈ab* ∨ w∈aa*. There is a computation for each possible instance of w:

w=EMP: (S, EMP) ⊢ (F, EMP)

w∈ab*: (S, ab*) ⊢ (A, b*) ⊢* (A, EMP)

w∈aa*: (S, aa*) ⊢ (B, a*) ⊢* (B, EMP)

Thus, w∈L(LNDFA).

Finite-State Machines

Theorem

$L = L(LNDFA)$

<u>w∈L ⇒ w∈L(LNDFA)</u>

Assume w∈L. This means w=EMP ∨ w∈ab* ∨ w∈aa*. There is a computation for each possible instance of w:

w=EMP: (S, EMP) ⊢ (F, EMP)

w∈ab*: (S, ab*) ⊢ (A, b*) ⊢* (A, EMP)

w∈aa*: (S, aa*) ⊢ (B, a*) ⊢* (B, EMP)

Thus, w∈L(LNDFA).

<u>w∈L(LNDFA) ⇒ w∈L</u>

Assume w∈L(LNDFA). This means LNDFA halts in F, B, or A after consuming w. Given that invariants always hold, w∈L.

Finite-State Machines

Theorem

$L = L(LNDFA)$

w∈L ⇒ w∈L(LNDFA)

Assume w∈L. This means w=EMP ∨ w∈ab* ∨ w∈aa*. There is a computation for each possible instance of w:

w=EMP: (S, EMP) ⊢ (F, EMP)

w∈ab*: (S, ab*) ⊢ (A, b*) ⊢* (A, EMP)

w∈aa*: (S, aa*) ⊢ (B, a*) ⊢* (B, EMP)

Thus, w∈L(LNDFA).

w∈L(LNDFA) ⇒ w∈L

Assume w∈L(LNDFA). This means LNDFA halts in F, B, or A after consuming w. Given that invariants always hold, w∈L.

* w∉L ⇔ w∉L(LNDFA) by contraposition.

Finite-State Machines

An alternative design (using closure properties of regular languages)

Finite-State Machines

An alternative design (using closure properties of regular languages)

```
;; L = {ϵ}
;; State Documentation:   S: ci = empty
(define E (make-ndfa '(S) '(a b) 'S '(S) '()))
```

Finite-State Machines

An alternative design (using closure properties of regular languages)

```
;; L = {ϵ}
;; State Documentation:   S: ci = empty
(define E (make-ndfa '(S) '(a b) 'S '(S) '()))

;; L = aa*
;; State Documentation
;;   S: ci = empty   F: ci = a+
(define A+ (make-ndfa '(S F) '(a b) 'S '(F)
                      '((S a F) (F a F))))
```

Finite-State Machines

An alternative design (using closure properties of regular languages)

```
;; L = {ε}
;; State Documentation:   S: ci = empty
(define E (make-ndfa '(S) '(a b) 'S '(S) '()))

;; L = aa*
;; State Documentation
;;   S: ci = empty   F: ci = a+
(define A+ (make-ndfa '(S F) '(a b) 'S '(F)
                      '((S a F) (F a F))))

;; L = ab*
;; State Documentation
;;   S: ci = empty   F: ci = a+
(define AB* (make-ndfa '(S F) '(a b) 'S '(F)
                       '((S a F) (F b F))))
```

Finite-State Machines

An alternative design (using closure properties of regular languages)

```
;; L = {ϵ}
;; State Documentation:   S: ci = empty
(define E (make-ndfa '(S) '(a b) 'S '(S) '()))

;; L = aa*
;; State Documentation
;;   S: ci = empty   F: ci = a+
(define A+ (make-ndfa '(S F) '(a b) 'S '(F)
                      '((S a F) (F a F))))

;; L = ab*
;; State Documentation
;;   S: ci = empty   F: ci = a+
(define AB* (make-ndfa '(S F) '(a b) 'S '(F)
                       '((S a F) (F b F))))


;; L = {ϵ} U aa* U ab*
(define LNDFA (sm-union E (sm-union A+ AB*)))
```

Finite-State Machines

Exercise: Σ={c,d}. Design, implement, & verify a `dfa` for:
L = {w|w has even number of c's and even number of d's}

Context-Free Grammars

Constructor

make-cfg: N T R S $\rightarrow$ cfg

* R $\in$ {N $\rightarrow$ {N $\cup$ T $\cup$ {ϵ}}$^+$}

Context-Free Grammars

Constructor

make-cfg: N T R S → cfg

* $R \in \{N \rightarrow \{N \cup T \cup \{\epsilon\}\}^{+}\}$

Observers

grammar-nts grammar-sigma grammar-rules
grammar-start grammar-derive

Context-Free Grammars

Constructor

make-cfg: N T R S → cfg

* R∈{N → {N ∪ T ∪ {ϵ}}$^+$}

Observers

grammar-nts grammar-sigma grammar-rules
grammar-start grammar-derive

Testing

check-derive? check-not-derive?

#:accepts #:rejects

Context-Free Grammars

Constructor

make-cfg: N T R S $\rightarrow$ cfg

* $R \in \{N \rightarrow \{N \cup T \cup \{\epsilon\}\}^{+}\}$

Observers

grammar-nts grammar-sigma grammar-rules
grammar-start grammar-derive

Testing

check-derive? check-not-derive?

#:accepts #:rejects

Visualization

grammar-viz

Context-Free Grammars

1. Pick a name for the grammar and specify the alphabet
2. Define each syntactic category and associate each with a nonterminal clearly specifying the starting nonterminal
3. Develop the production rules
4. Write unit tests
5. Implement the grammar
6. Run the tests and redesign if necessary

7 For each syntactic category design and implement an invariant predicate to determine if a given word satisfies the role of the syntactic category

8 For words in L(G) prove that the invariant predicates hold for every derivation step.

9 Prove that L = L(G)

Context-Free Grammars

```
;; L = {ww^r | w in {a, b}*}

(define pali
  (make-cfg

    '(a b)
```

Context-Free Grammars

```
;; L = {ww^r | w in {a, b}*}
;; Syntactic Categories Documentation
;;  S: generates a palindrome, starting nt
(define pali
  (make-cfg
    '(S)
    '(a b)



    'S
```

Context-Free Grammars

```
;; L = {ww^r | w in {a, b}*}
;; Syntactic Categories Documentation
;;  S: generates a palindrome, starting nt
(define pali
  (make-cfg
    '(S)
    '(a b)
    `((S ,ARROW ,EMP) (S ,ARROW a) (S ,ARROW b)
      (S ,ARROW aSa)  (S ,ARROW bSb))
    'S
```

Context-Free Grammars

```
;; L = {ww^r | w in {a, b}*}
;; Syntactic Categories Documentation
;;  S: generates a palindrome, starting nt
(define pali
  (make-cfg
    '(S)
    '(a b)
    `((S ,ARROW ,EMP) (S ,ARROW a) (S ,ARROW b)
      (S ,ARROW aSa)  (S ,ARROW bSb))
    'S
    #:rejects '((a b)  (b a a) (a b b b))
    #:accepts '(() (a) (b) (a a b b a a)))))

(check-not-derive? pali '(a a a b)  '(b b a))
(check-derive? pali '(a a a) '(b b b b)
                    '(a b b a a a a b b a))
```

Context-Free Grammars

```
;; L = {ww^r | w in {a, b}*}

;; word → Boolean
;; Purpose: Determine if S should generate the given word
(define (S-INV wrd)
  (let* [(wlen (quotient (length wrd) 2))
         (w (take wrd wlen))
         (wrd-w (drop wrd wlen))
         (w^r (if (= (length w) (length wrd-w))
                  wrd-w
                  (rest wrd-w)))]
    (equal? w (reverse w^r))))

(check-inv-fails? S-INV '(b a b a)
                        '(a a a a a b))
(check-inv-holds? S-INV '(a a) '(a b a)
                        '(b b a b b b a b b))
```

Context-Free Grammars

Theorem

Invariants hold deriving w∈L(pali).

Proof.

Proof by induction on n = the height of derivation tree.
Base Case: n = 1
Yield is either EMP, a, or b. For all, w and w^r equal EMP.
Thus, S-INV holds. □

Context-Free Grammars

Inductive Step:

Assume: NT invariants hold for n = k.

Show: NT invariants hold for n = k+1.

Context-Free Grammars

Inductive Step:
Assume: NT invariants hold for n = k.
Show: NT invariants hold for n = k+1.

Consider each production for n = k+1:
(S $\rightarrow$ aSa) By IH, S-INV holds. This means (rhs) S generates wcw^r, where $c \in \{a, b, EMP\}$. By using this rule, the yield is $awcw^ra$. Observe that $(w^ra)^r = a(w^r)^r = aw$. Thus, S-INV holds.

Context-Free Grammars

Inductive Step:
Assume: NT invariants hold for n = k.
Show: NT invariants hold for n = k+1.

Consider each production for n = k+1:
(S → aSa) By IH, S-INV holds. This means (rhs) S generates wcw^r, where c∈{a, b, EMP}. By using this rule, the yield is $awcw^ra$. Observe that $(w^ra)^r$=$a(w^r)^r$=aw. Thus, S-INV holds.

(S → bSb) By IH, S-INV holds. This means (rhs) S generates wcw^r, where c∈{a, b, EMP}. By using this rule, the yield is $bwcw^rb$. Observe that $(w^rb)^r$=$b(w^r)^r$=bw. Thus, S-INV holds.

Context-Free Grammars

Theorem

$L = L(pali)$

Context-Free Grammars

Theorem

$L = L(pali)$

w∈L ⇒ w∈L(pali)

Assume w∈L. This means w=aPa or bPb, where P is a palindrome. S generates w by repeatedly applying S → aSa or S → bSb until S → ϵ, S → a, or S → b is applied. Thus, w∈L(pali).

Context-Free Grammars

Theorem

L = L(pali)

w∈L ⇒ w∈L(pali)

Assume w∈L. This means w=aPa or bPb, where P is a palindrome. S generates w by repeatedly applying S → aSa or S → bSb until S → ϵ, S → a, or S → b is applied. Thus, w∈L(pali).

w∈L(pali) ⇒ w∈L

Assume w∈L(pali). This means w is generated by S. Given that invariants always hold, w∈L.

Context-Free Grammars

EXERCISE

Design and implement a grammar for:
$L = \{w \mid w$ has balanced parenthesis$\}$, where o = (and c =).

Pushdown Automata

Would like to have a machine that decides if a given word is a member of a CFL

What should such a machine look like?

Pushdown Automata

Would like to have a machine that decides if a given word is a member of a CFL

What should such a machine look like?

Write a program to decide a^nb^n

Call an auxiliary function that takes as input w's sub-word without the leading as, if any, and an accumulator with w's leading as

3 conditions:

1. If the given word is empty then testing if the given accumulator is empty is returned.
2. If the first element of the given word is a then false is returned.
3. Otherwise, return the conjunction of testing if the accumulator is not empty and checking the rest of both the given word and the given accumulator.

Pushdown Automata

```
#lang fsm
;; word → Boolean
;; Purpose: Decide if given word is in a^nb^n
(define (is-in-a^nb^n? w)
```

```
;; Tests for is-in-anbn?
(check-pred (λ (w) (not (is-in-a^nb^n? w))) '(a))
(check-pred (λ (w) (not (is-in-a^nb^n? w))) '(b b))
(check-pred (λ (w) (not (is-in-a^nb^n? w))) '(a b b))
(check-pred (λ (w) (not (is-in-a^nb^n? w))) '(a b a a b b))
(check-pred is-in-a^nb^n? '())
(check-pred is-in-a^nb^n? '(a a b b))
```

Pushdown Automata

```
#lang fsm
;; word → Boolean
;; Purpose: Decide if given word is in a^nb^n
(define (is-in-a^nb^n? w)








(check (dropf w (λ (s) (eq? s 'a)))   ;; everything after initial a's
       (takef w (λ (s) (eq? s 'a))))) ;; the initial a's

  ;; Tests for is-in-anbn?
  (check-pred (λ (w) (not (is-in-a^nb^n? w))) '(a))
  (check-pred (λ (w) (not (is-in-a^nb^n? w))) '(b b))
  (check-pred (λ (w) (not (is-in-a^nb^n? w))) '(a b b))
  (check-pred (λ (w) (not (is-in-a^nb^n? w))) '(a b a a b b))
  (check-pred is-in-a^nb^n? '())
  (check-pred is-in-a^nb^n? '(a a b b))
```

Pushdown Automata

```
#lang fsm
;; word → Boolean
;; Purpose: Decide if given word is in a^nb^n
(define (is-in-a^nb^n? w)

  ;; word (listof symbol) → Boolean
  ;; Purpose: Determine if first given word has only bs that match as in the
  ;;          second given word
  ;; Accumulator Invariant
  ;;   acc = the unmatched a's at the beginning of w
  ;; Assume: w in (a b)*
  (define (check wrd acc)
    (cond [(empty? wrd) (empty? acc)]
          [(eq? (first wrd) 'a) #f]
          [else (and (not (empty? acc))
                     (check (rest wrd) (rest acc)))]))

(check (dropf w (λ (s) (eq? s 'a)))   ;; everything after initial a's
       (takef w (λ (s) (eq? s 'a))))) ;; the initial a's

   ;; Tests for is-in-anbn?
   (check-pred (λ (w) (not (is-in-a^nb^n? w))) '(a))
   (check-pred (λ (w) (not (is-in-a^nb^n? w))) '(b b))
   (check-pred (λ (w) (not (is-in-a^nb^n? w))) '(a b b))
   (check-pred (λ (w) (not (is-in-a^nb^n? w))) '(a b a a b b))
   (check-pred is-in-a^nb^n? '())
   (check-pred is-in-a^nb^n? '(a a b b))
```

Pushdown Automata

```
#lang fsm
;; word → Boolean
;; Purpose: Decide if given word is in a^nb^n
(define (is-in-a^nb^n? w)

  ;; word (listof symbol) → Boolean
  ;; Purpose: Determine if first given word has only bs that match as in the
  ;;            second given word
  ;; Accumulator Invariant
  ;;   acc = the unmatched a's at the beginning of w
  ;; Assume: w in (a b)*
  (define (check wrd acc)
    (cond [(empty? wrd) (empty? acc)]
          [(eq? (first wrd) 'a) #f]
          [else (and (not (empty? acc))
                     (check (rest wrd) (rest acc)))]))

(check (dropf w (λ (s) (eq? s 'a)))   ;; everything after initial a's
       (takef w (λ (s) (eq? s 'a))))) ;; the initial a's

  ;; Tests for is-in-anbn?
  (check-pred (λ (w) (not (is-in-a^nb^n? w))) '(a))
  (check-pred (λ (w) (not (is-in-a^nb^n? w))) '(b b))
  (check-pred (λ (w) (not (is-in-a^nb^n? w))) '(a b b))
  (check-pred (λ (w) (not (is-in-a^nb^n? w))) '(a b a a b b))
  (check-pred is-in-a^nb^n? '())
  (check-pred is-in-a^nb^n? '(a a b b))
```

The acc is a stack
The program first pushes all the as at the beginning of the given word onto the accumulator
The auxiliary function pops an a for each recursive call
Suggests extending an `ndfa` with a stack to remember part of the consumed input

Pushdown Automata

A (nondeterministic) pushdown automaton, pda, is an instance of:

```
(make-ndpda K Σ Γ S F δ)
```

Pushdown Automata

A (nondeterministic) pushdown automaton, `pda`, is an instance of:

```
(make-ndpda K Σ Γ S F δ)
```

The transition relation, δ, is a finite subset of:

$$((\mathtt{K} \times (\Sigma \cup \{\mathtt{EMP}\}) \times \Gamma^{+} \cup \{EMP\}) \times (\mathtt{K} \times \Gamma^{+} \cup \{EMP\})).$$

Pushdown Automata

A (nondeterministic) pushdown automaton, pda, is an instance of:

(make−ndpda K Σ Γ S F δ)

- The transition relation, δ, is a finite subset of:
  ((K × (Σ ∪ {EMP}) × $\Gamma^+ \cup \{EMP\}$) × (K × $\Gamma^+ \cup \{EMP\}$)).
- ((A a p) (B g)) means that the machine is in state A, reads a from the input tape, pops p off the stack, pushes g onto the stack, and moves to B
- May be nondeterministic

Pushdown Automata

A (nondeterministic) pushdown automaton, pda, is an instance of:

```
(make-ndpda K Σ Γ S F δ)
```

The transition relation, δ, is a finite subset of:

$((K \times (\Sigma \cup \{EMP\}) \times \Gamma^{+} \cup \{EMP\}) \times (K \times \Gamma^{+} \cup \{EMP\}))$.

((A a p) (B g)) means that the machine is in state A, reads a from the input tape, pops p off the stack, pushes g onto the stack, and moves to B

May be nondeterministic

((P EMP EMP) (Q j)) is a push operation that does not consult the input tape nor the stack

((P EMP j) (Q EMP)) is a pop operation

Pushdown Automata

A pda configuration is a member of $(K \times \Sigma^* \times \Gamma^*)$

Pushdown Automata

A pda configuration is a member of (K $\times$ Σ^* $\times$ Γ^*)

A computation step moves the machine from a starting configuration to a new configuration using a single rule denoted as:

`(P, xw, a)` $\vdash$ `(Q, w, g)`

Pushdown Automata

A pda configuration is a member of $(K \times \Sigma^* \times \Gamma^*)$

A computation step moves the machine from a starting configuration to a new configuration using a single rule denoted as:

$$(P,\ xw,\ a) \vdash (Q,\ w,\ g)$$

Zero or more steps are denoted using $\vdash^*$

Pushdown Automata

A pda configuration is a member of $(K \times \Sigma^* \times \Gamma^*)$

A computation step moves the machine from a starting configuration to a new configuration using a single rule denoted as:

$$(P,\ xw,\ a) \vdash (Q,\ w,\ g)$$

Zero or more steps are denoted using $\vdash^*$

A computation of length n on a word, w, is denoted by:

$$C_0 \vdash C_1 \vdash C_2 \vdash \ldots \vdash C_n, \text{ where } C_i \text{ is a pda configuration.}$$

Pushdown Automata

- A pda configuration is a member of (K $\times$ Σ^* $\times$ Γ^*)
- A computation step moves the machine from a starting configuration to a new configuration using a single rule denoted as:

  $(P, xw, a) \vdash (Q, w, g)$

- Zero or more steps are denoted using $\vdash^*$
- A computation of length n on a word, w, is denoted by:

  $C_0 \vdash C_1 \vdash C_2 \vdash \ldots \vdash C_n$, where C_i is a pda configuration.

- If a pda, M, starting in the starting state consumes all the input and reaches a final state with the stack empty then M accepts. Otherwise, M rejects
- A word, w, is in the language of M, L(M), if there is a computation from the starting state that consumes w and M accepts
- It does not matter that there may be computations that reject w.

Pushdown Automata

A state invariant predicate may be associated with each state

An invariant predicate has two inputs: the consumed input and the stack

It must test and relate the invariant conditions for and between the consumed input and the stack.

Pushdown Automata

Design and implement a pda for L = $a^n b^n$

How may the stack be used?

Pushdown Automata

Design and implement a pda for L = a^nb^n

How may the stack be used?

Accumulate the read as

Pushdown Automata

Design and implement a `pda` for `L` = $a^n b^n$

How may the stack be used?

Accumulate the read `a`s

Match `b`s by popping `a`s

Pushdown Automata

Design and implement a pda for $L = a^nb^n$

How may the stack be used?

Accumulate the read as

Match bs by popping as

After all the input is read, the machine ought to move to a final state

It accepts if the stack is empty. Otherwise, it rejects

Pushdown Automata

Name: a^nb^n Σ = {a b}

Pushdown Automata

Name: a^nb^n Σ = {a b}

```
;; Tests for a^nb^n
(check-reject? a^nb^n '(a) '(b b) '(a b b)
                        '(a b a a b b) '(a a b b a b))
(check-accept? a^nb^n '() '(a a b b) '(a a a a a b b b b b))
```

Pushdown Automata

Name: a^nb^n Σ = {a b}

```
;; Tests for a^nb^n
(check-reject? a^nb^n '(a) '(b b) '(a b b)
                        '(a b a a b b) '(a a b b a b))
(check-accept? a^nb^n '() '(a a b b) '(a a a a a b b b b b))
```

States

```
;; States
;;  S:   ci = a*  = stack, start state
```

Pushdown Automata

Name: a^nb^n Σ = {a b}

```
;; Tests for a^nb^n
(check-reject? a^nb^n '(a) '(b b) '(a b b)
                          '(a b a a b b) '(a a b b a b))
(check-accept? a^nb^n '() '(a a b b) '(a a a a a b b b b b))
```

States

```
;; States
;;  S:   ci = a*  = stack, start state



;;  M:   ci = (append (listof a) (listof b))
;;         ∧ stack = a*
;;         ∧ |ci a's| = |stack| + |ci \texttt{b}s|
```

Pushdown Automata

Name: a^nb^n Σ = {a b}

```
;; Tests for a^nb^n
(check-reject? a^nb^n '(a) '(b b) '(a b b)
                         '(a b a a b b) '(a a b b a b))
(check-accept? a^nb^n '() '(a a b b) '(a a a a a b b b b b))
```

States

```
;; States
;;  S:   ci = a*  = stack, start state


;;  M:   ci = (append (listof a) (listof b))
;;     ∧ stack = a*
;;     ∧ |ci a's| = |stack| + |ci \texttt{b}s|


;;  F:   ci = (append (listof a) (listof b))
;;     ∧ |stack| = 0
;;     ∧ |ci a's|=|ci b's|, final state
```

Pushdown Automata

Transition Relation

Push all a and nondeterministically move to M

```
((S ,EMP ,EMP) (M ,EMP))     ((S a ,EMP) (S (a)))
```

Pushdown Automata

Transition Relation

Push all a and nondeterministically move to M

```
((S ,EMP ,EMP) (M ,EMP))     ((S a ,EMP) (S (a)))
```

Match all b and nondeterministically move to F

```
((M b (a)) (M ,EMP))     ((M ,EMP ,EMP) (F ,EMP))
```

Pushdown Automata

```
;; L = {a^nb^n | n >= 0}
;; States
;;  S ci = (listof a) = stack, start state
;;  M ci = (append (listof a) (listof b)) AND
;;          (length ci as) = (length stack) + (length ci bs)
;;  F ci = (append (listof a) (listof b)) and all as and bs matched,
;;    final state
;; The stack is a (listof a)
(define a^nb^n (make-ndpda '(S M F) '(a b) '(a) 'S '(F)
                           `(((S ,ε ,ε) (M ,ε))
                             ((S a ,ε) (S (a)))
                             ((M b (a)) (M ,ε))
                             ((M ,ε ,ε) (F ,ε)))))

;; Tests for a^nb^n
(check-reject? a^nb^n '(a) '(b b) '(a b b) '(a b a a b b)
                      '(a a b b a b))
(check-accept? a^nb^n '() '(a a b b) '(a a a a a b b b b b))
```

Pushdown Automata

```
;; word stack → Boolean
;; Purpose: Determine if ci and stack are the same (listof a)
(define (S-INV ci stck)
 (and (= (length ci) (length stck))
      (andmap (λ (i g) (and (eq? i 'a) (eq? g 'a))) ci stck)))
;; Tests for S-INV
(check-inv-fails? S-INV '(() (a a)) '((a) ()) '((b b b) (b b b)))
(check-inv-holds? S-INV '(() ()) '((a a a) (a a a)))
```

Pushdown Automata

```
;; word stack → Boolean
;; Purpose: Determine if ci and stack are the same (listof a)
(define (S—INV ci stck)
 (and (= (length ci) (length stck))
      (andmap (λ (i g) (and (eq? i 'a) (eq? g 'a))) ci stck)))
;; Tests for S—INV
(check—inv—fails? S—INV '(() (a a)) '((a) ()) '((b b b) (b b b)))
(check—inv—holds? S—INV '(() ()) '((a a a) (a a a)))

;; word stack → Boolean
;; Purpose: Determine if ci = ϵ or a+b+ AND the stack
;;          only contains a AND |ci as| = |stack| + |ci bs|
(define (M—INV ci stck)
  (let* [(as (takef ci (\lamb (s) (eq? s 'a))))
         (bs (takef (drop ci (length as)) (λ (s) (eq? s 'b))))]
    (and (equal? (append as bs) ci)
         (andmap (λ (s) (eq? s 'a)) stck)
         (= (length as) (+ (length bs) (length stck))))))
;; Tests for M—INV
(check—inv—fails? M—INV '((a a b) (a a)) '((a)'())
                        '((a a a b) (a a a)) '((a a a b) (a)))
(check—inv—holds? M—INV
                  '(() ()) '((a) (a)) '((a b) ())
                  '((a a a b b) (a)))
```

Pushdown Automata

```
;; word stack → Boolean
;; Purpose: Determine if ci and stack are the same (listof a)
(define (S-INV ci stck)
 (and (= (length ci) (length stck))
      (andmap (λ (i g) (and (eq? i 'a) (eq? g 'a))) ci stck)))
;; Tests for S-INV
(check-inv-fails? S-INV '(() (a a)) '((a) ()) '((b b b) (b b b)))
(check-inv-holds? S-INV '(() ()) '((a a a) (a a a)))

;; word stack → Boolean
;; Purpose: Determine if ci = ϵ or a+b+ AND the stack
;;          only contains a AND |ci as| = |stack| + |ci bs|
(define (M-INV ci stck)
  (let* [(as (takef ci (\lamb (s) (eq? s 'a))))
         (bs (takef (drop ci (length as)) (λ (s) (eq? s 'b))))]
    (and (equal? (append as bs) ci)
         (andmap (λ (s) (eq? s 'a)) stck)
         (= (length as) (+ (length bs) (length stck))))))
;; Tests for M-INV
(check-inv-fails? M-INV '((a a b) (a a)) '((a)'())
                        '((a a a b) (a a a)) '((a a a b) (a)))
(check-inv-holds? M-INV
                  '(() ()) '((a) (a)) '((a b) ())
                  '((a a a b b) (a)))

;; word stack → Boolean Purpose: Determine if ci=a^nb^n & empty stack
(define (F-INV ci stck)
  (let* [(as (takef ci (λ (s) (eq? s 'a))))
         (bs (takef (drop ci (length as)) (λ (s) (eq? s 'b))))]
    (and (empty? stck) (equal? (append as bs) ci)  (= (length as) (length bs)))))
;; Tests for F-INV
(check-inv-fails? F-INV '((a a b) ()) '((a) ()) '((a a a b) (a a a)) '((b b b) (b b
       b)))
```

Pushdown Automata

`L = a`^{n}`b`^{n} `ci = the consumed input` `w`$\in$`(sm-sigma M)`*
`F=(sm-finals M)` `P = a^nb^n`

Theorem

The state invariants hold when P is applied to w.

The proof is by induction on, `n`, the number of transitions to consume `w`

Pushdown Automata

Base Case

- When P starts, S-INV holds because ci = `'()`and the stack = `'()`
- Observe that empty transitions into M and F may lead to accept and, therefore, we must establish that the invariants for these states also hold
- After using `((S EMP EMP) (M EMP))`, `M-INV` holds because ci=`'()`and the stack=`'()`
- After using `((M EMP EMP) (F EMP))`, `F-INV` also holds because ci=`'()`and the stack=`'()`

Pushdown Automata

Assume INVs hold for k steps. Show they hold for the k+1 step

Pushdown Automata

Assume INVs hold for k steps. Show they hold for the k+1 step

Proof invariants hold after each nonempty transition:

`((S a EMP) (S (a)))`

By inductive hypothesis, `S-INV` holds
After consuming an a and pushing an a, P may reach S and by empty transition M
Note that an empty transition into F with a nonempty stack cannot lead to an accept. Therefore, we do not concern ourselves about P making such a transition
`S-INV` and `M-INV` hold because both the length of the consumed input and of the stack increased by 1, thus, remaining of equal length and because both continue to only contain as

Pushdown Automata

Assume INVs hold for k steps. Show they hold for the k+1 step

Proof invariants hold after each nonempty transition:

`((S a EMP) (S (a)))`

- By inductive hypothesis, `S-INV` holds
- After consuming an `a` and pushing an `a`, P may reach S and by empty transition M
- Note that an empty transition into F with a nonempty stack cannot lead to an accept. Therefore, we do not concern ourselves about P making such a transition
- `S-INV` and `M-INV` hold because both the length of the consumed input and of the stack increased by 1, thus, remaining of equal length and because both continue to only contain `a`s

`((M b (a)) (M ,EMP))`:

- By inductive hypothesis, `M-INV` holds
- After consuming a `b` and popping an `a`, P may reach M or nondeterministically reach F because it may eventually accept
- `M-INV` holds because ci continues to be `a`s followed by `b`s, the stack can only contain `a`s, and the number of `a`s in ci remains equal to the sum of the number of `b`s in ci and the length of the stack.
- `F-INV` holds because for a computation that ends with an accept the read `b` is the last symbol in the given word and popping an `a` makes the stack empty, ci continues to be the read `a`s followed by the read `b`s, and, given that the stack is empty, the number of `a`s equals the number of `b`s in the consumed input.

Pushdown Automata

Proving `L = L(P)`

Pushdown Automata

Proving `L = L(P)`

Lemma

$w \in L \Leftrightarrow w \in L(P)$

Proof.

(⇒) Assume w∈L. This means that $w = a^n b^n$. Given that state invariants always hold, there is a computation that has P consume all the `a`s, then consume all the `b`s, and then reach F with an empty stack. Therefore, w∈L(P).
(⇐) Assume w∈L(P). This means that M halts in F, the only final state, with an empty stack having consumed w. Given that the state invariants always hold we may conclude that $w = a^n b^n$. Therefore, w∈L. □

Pushdown Automata

Time to have some fun!

Pushdown Automata

Time to have some fun!

According to Chomsky: RL $\subset$ CFL

Pushdown Automata

Time to have some fun!

According to Chomsky: RL $\subset$ CFL

Prove it!

Pushdown Automata

;; ndfa $\rightarrow$ pda
;; Purpose: Convert the given ndfa to a pda

Pushdown Automata

```
;; ndfa → pda
;; Purpose: Convert the given ndfa to a pda

(define (ndfa2pda M #:accepts [accs '()] #:rejects [rejs '()])
```

Pushdown Automata

```
;; ndfa → pda
;; Purpose: Convert the given ndfa to a pda

(define (ndfa2pda M #:accepts [accs '()] #:rejects [rejs '()])

  (let [(states  (sm-states M))
        (sigma   (sm-sigma M))
        (start   (sm-start M))
        (finals  (sm-finals M))
        (rules   (sm-rules M))]
```

Pushdown Automata

```
;; ndfa → pda
;; Purpose: Convert the given ndfa to a pda

(define (ndfa2pda M #:accepts [accs '()] #:rejects [rejs '()])

  (let [(states  (sm-states M))
        (sigma   (sm-sigma M))
        (start   (sm-start M))
        (finals  (sm-finals M))
        (rules   (sm-rules M))]

    (make-ndpda states sigma '() start finals
```

Pushdown Automata

```
;; ndfa → pda
;; Purpose: Convert the given ndfa to a pda

(define (ndfa2pda M #:accepts [accs '()] #:rejects [rejs '()])

  (let [(states  (sm-states M))
        (sigma   (sm-sigma M))
        (start   (sm-start M))
        (finals  (sm-finals M))
        (rules   (sm-rules M))]

   (make-ndpda states sigma '() start finals




                   #:accepts accs
                   #:rejects rejs)))
```

Pushdown Automata

```
;; ndfa → pda
;; Purpose: Convert the given ndfa to a pda

(define (ndfa2pda M #:accepts [accs '()] #:rejects [rejs '()])

  (let [(states  (sm-states M))
        (sigma   (sm-sigma M))
        (start   (sm-start M))
        (finals  (sm-finals M))
        (rules   (sm-rules M))]

   (make-ndpda states sigma '() start finals

               (map (λ (r)
                       (list (list (first r) (second r) ε)
                             (list (third r) ε)))
                    rules)

               #:accepts accs
               #:rejects rejs)))
```

Recursively Enumerable Languages

A Turing machine language recognizer is an instance of:

(make–tm K Σ R S F Y)

Recursively Enumerable Languages

A Turing machine language recognizer is an instance of:

(make−tm K Σ R S F Y)

R is a transition relation

Recursively Enumerable Languages

A Turing machine language recognizer is an instance of:

```
(make-tm K Σ R S F Y)
```

`R` is a transition relation

A Turing machine language recognizer requires two final states usually named `Y` and `N`

When a Turing machine reaches a final state it halts and performs no more transitions

Recursively Enumerable Languages

A Turing machine language recognizer is an instance of:

(make−tm K Σ R S F Y)

R is a transition relation

A Turing machine language recognizer requires two final states usually named Y and N

When a Turing machine reaches a final state it halts and performs no more transitions

Two special symbols that may appear of the input tape: LM and BLANK (both FSM constants)

A Turing machine rule, tm-rule, is an element of:

```
(list (list N a) (list M A))
```

N is non-halting state

a∈{Σ ∪ {LM BLANK}},

M∈K

A is an action∈{Σ ∪ {RIGHT LEFT}

If A∈Σ then the machine writes A in the tape position under the tape's head

Recursively Enumerable Languages

A Turing machine language recognizer is an instance of:

(make−tm K Σ R S F Y)

R is a transition relation

A Turing machine language recognizer requires two final states usually named Y and N

When a Turing machine reaches a final state it halts and performs no more transitions

Two special symbols that may appear of the input tape: LM and BLANK (both FSM constants)

A Turing machine rule, tm-rule, is an element of:

```
(list (list N a) (list M A))
```

N is non-halting state

a∈{Σ ∪ {LM BLANK}},

M∈K

A is an action∈{Σ ∪ {RIGHT LEFT}

If A∈Σ then the machine writes A in the tape position under the tape's head

When LM is read the tm must move the tape's head right (regardless of the state it is in)

May not overwrite LM

The tm cannot "fall off" the left end of the input tape

Recursively Enumerable Languages

- A Turing machine configuration is a triple: `(state natnum tape)`
- Only the "touched" part of the tape is displayed
- The touched part of the input tape includes the left-end marker and anything specified in the initial tape value including blanks

Recursively Enumerable Languages

A computation, $\texttt{C}_i \vdash^* \texttt{C}_j$, is valid for $\texttt{M}$ if and only if $\texttt{M}$ can move from $\texttt{C}_i$ to $\texttt{C}_j$ using zero or more transitions

Recursively Enumerable Languages

A computation, $\texttt{C}_i \vdash^* \texttt{C}_j$, is valid for M if and only if M can move from $\texttt{C}_i$ to $\texttt{C}_j$ using zero or more transitions

A word, w, is accepted by a tm language recognizer if it reaches the final accepting state

Otherwise, w is rejected

Recursively Enumerable Languages

- A computation, $\mathtt{C}_i \vdash^* \mathtt{C}_j$, is valid for `M` if and only if `M` can move from $\mathtt{C}_i$ to $\mathtt{C}_j$ using zero or more transitions
- A word, `w`, is accepted by a `tm` language recognizer if it reaches the final accepting state
- Otherwise, `w` is rejected
- A Turing machine language recognizer's execution may be observed using `sm-viz`

Recursively Enumerable Languages

- A computation, $C_i \vdash^* C_j$, is valid for M if and only if M can move from C_i to C_j using zero or more transitions
- A word, w, is accepted by a tm language recognizer if it reaches the final accepting state
- Otherwise, w is rejected
- A Turing machine language recognizer's execution may be observed using `sm-viz`
- State invariant predicates take as input the "touched" part of the input tape and the position, i, of the input tape's next element to read
- The predicate asserts a condition about the touched input that must hold which may or may not be in relation to the head's position

Recursively Enumerable Languages

Let's now design a nondeterministic Turing machine

The transition relation does not have to be a function

Recursively Enumerable Languages

Let's now design a nondeterministic Turing machine

The transition relation does not have to be a function

L = $a^* \cup a^*b$

Recursively Enumerable Languages

Let's now design a nondeterministic Turing machine

The transition relation does not have to be a function

L = $a^* \cup a^*b$

Name: `a*Ua*b` Σ = `{a b}`

;; PRE: tape = LMw AND i = 1

Recursively Enumerable Languages

Let's now design a nondeterministic Turing machine

The transition relation does not have to be a function

L = $a^* \cup a^*b$

Name: a*Ua*b Σ = {a b}

```
;; PRE: tape = LMw AND i = 1
```

Tests

```
;; Tests for a*Ua*b
(check-reject? a*Ua*b
               `((,LM b b) 1)
               `((,LM a a b a) 1))
(check-accept? a*Ua*b
               `((,LM ,BLANK) 1)
               `((,LM b) 1)
               `((,LM a b) 1)
               `((,LM a a a) 1)
               `((,LM a a a b) 1))
```

Recursively Enumerable Languages

When the machine starts in S nothing has been read

If the input word is empty the machine moves to accept

Recursively Enumerable Languages

When the machine starts in S nothing has been read

If the input word is empty the machine moves to accept

If the first element is an a then the machine nondeterministically moves to a state A to read a^* or to a state B to read a^*b

Recursively Enumerable Languages

When the machine starts in `S` nothing has been read

If the input word is empty the machine moves to accept

If the first element is an `a` then the machine nondeterministically moves to a state `A` to read `a`* or to a state `B` to read `a`*`b`

Upon reading `a`* in `A` the machine may accept

Upon reading `a`*`b` in `B` the machine moves to state `C` to determine if the end of the input word has been reached and then moves to either accept or reject

Recursively Enumerable Languages

When the machine starts in `S` nothing has been read

If the input word is empty the machine moves to accept

If the first element is an `a` then the machine nondeterministically moves to a state `A` to read `a`* or to a state `B` to read `a`*`b`

Upon reading `a`* in `A` the machine may accept

Upon reading `a`*`b` in `B` the machine moves to state `C` to determine if the end of the input word has been reached and then moves to either accept or reject

From `C` the machine may accept upon reading a blank and reject otherwise.

Recursively Enumerable Languages

When the machine starts in `S` nothing has been read

If the input word is empty the machine moves to accept

If the first element is an `a` then the machine nondeterministically moves to a state `A` to read `a`* or to a state `B` to read `a`*`b`

Upon reading `a`* in `A` the machine may accept

Upon reading `a`*`b` in `B` the machine moves to state `C` to determine if the end of the input word has been reached and then moves to either accept or reject

From `C` the machine may accept upon reading a blank and reject otherwise.

The states may documented as follows:

```
;; States (i is the position of the head)
;;   S: no tape elements read, starting sate
;;   A: tape[1..i-1] has only a
;;   B: tape[1..i-1] has only a
;;   C: tape[1..i-2] has only a and tape[i-1] = b
;;   Y: tape[i] = BLANK and tape[1..i-1] in a* or a*b,
;;      final accepting state
;;   N: tape[1..i-1] != a* nor a*b, final state
```

Recursively Enumerable Languages

Transition relation

```
((S ,BLANK) (Y ,BLANK))
((S a) (A ,RIGHT))
((S a) (B ,RIGHT))
((S b) (C ,RIGHT))
```

Recursively Enumerable Languages

Transition relation

```
((S ,BLANK) (Y ,BLANK))
((S a) (A ,RIGHT))
((S a) (B ,RIGHT))
((S b) (C ,RIGHT))

((A a) (A ,RIGHT))
((A ,BLANK) (Y ,BLANK))
```

Recursively Enumerable Languages

Transition relation

```
((S ,BLANK) (Y ,BLANK))
((S a) (A ,RIGHT))
((S a) (B ,RIGHT))
((S b) (C ,RIGHT))

((A a) (A ,RIGHT))
((A ,BLANK) (Y ,BLANK))

((B a) (B ,RIGHT))
((B b) (C ,RIGHT))
```

Recursively Enumerable Languages

Transition relation

```
((S ,BLANK) (Y ,BLANK))
((S a) (A ,RIGHT))
((S a) (B ,RIGHT))
((S b) (C ,RIGHT))

((A a) (A ,RIGHT))
((A ,BLANK) (Y ,BLANK))

((B a) (B ,RIGHT))
((B b) (C ,RIGHT))

((C a) (N ,RIGHT))
((C b) (N ,RIGHT))
((C ,BLANK) (Y ,BLANK)))
```

Turing Machines

Implementation

```
;; States (i is the position of the head)
;;   S: no tape elements read, starting sate
;;   A: tape[1..i-1] has only a
;;   B: tape[1..i-1] has only a
;;   C: tape[1..i-2] has only a and tape[i-1] = b
;;   Y: tape[i] = BLANK and tape[1..i-1] = a* or a*b,
;;      final accepting state
;;   N: tape[1..i-1] != a* or a*b, final state
;; L = a* U a*b     PRE: tape = LMw AND i = 1
(define a*Ua*b (make-tm '(S A B C Y N)
                        `(a b)
                        `(((S ,BLANK) (Y ,BLANK))
                          ((S a) (A ,RIGHT))
                          ((S a) (B ,RIGHT))
                          ((S b) (C ,RIGHT))
                          ((A a) (A ,RIGHT))
                          ((A ,BLANK) (Y ,BLANK))
                          ((B a) (B ,RIGHT))
                          ((B b) (C ,RIGHT))
                          ((C a) (N a))
                          ((C b) (N b))
                          ((C ,BLANK) (Y ,BLANK)))
                        'S
                        '(Y N)
                        'Y))
;; Tests for a*Ua*b
(check-reject? a*Ua*b `((,LM b b) 1)    `((,LM a a b a) 1))
(check-accept? a*Ua*b `((,LM ,BLANK) 1) `((,LM b) 1)
                       `((,LM a b) 1)    `((,LM a a a) 1)
                       `((,LM a a a b) 1))
```

Turing Machines

```
;; tape natnum → Boolean    Purpose: Determine that no tape elements read
(define (S-INV t i) (= i 1))
```

Turing Machines

```
;; tape natnum → Boolean    Purpose: Determine that no tape elements read
(define (S-INV t i) (= i 1))


;; tape natum → Boolean
;; Purpose: Determine that tape[1..i-1] only has a
(define (B-INV t i)
  (and (>= i 2) (andmap (λ (s) (eq? s 'a)) (take (rest t) (sub1 i)))))
```

Turing Machines

```
;; tape natnum → Boolean    Purpose: Determine that no tape elements read
(define (S−INV t i) (= i 1))


;; tape natum → Boolean
;; Purpose: Determine that tape[1..i−1] only has a
(define (B−INV t i)
  (and (>= i 2) (andmap (λ (s) (eq? s 'a)) (take (rest t) (sub1 i)))))


;; tape natnum → Boolean   Purpose: Determine that tape[1..i−1] only has a
(define (A−INV t i)
  (and (>= i 2)  (andmap (λ (s) (eq? s 'a)) (take (rest t) (sub1 i)))))
```

Turing Machines

```
;; tape natnum → Boolean    Purpose: Determine that no tape elements read
(define (S-INV t i) (= i 1))


;; tape natum → Boolean
;; Purpose: Determine that tape[1..i-1] only has a
(define (B-INV t i)
  (and (>= i 2) (andmap (λ (s) (eq? s 'a)) (take (rest t) (sub1 i)))))


;; tape natnum → Boolean   Purpose: Determine that tape[1..i-1] only has a
(define (A-INV t i)
  (and (>= i 2)  (andmap (λ (s) (eq? s 'a)) (take (rest t) (sub1 i)))))


;; tape natnum → Boolean
;; Purpose: Determine that tape[1..i-2] has only a and tape[i-1] = b
(define (C-INV t i)
  (and (>= i 2) (andmap (λ (s) (eq? s 'a)) (take (rest t) (- i 2)))
       (eq? (list-ref t (sub1 i)) 'b)))
```

Turing Machines

```
;; tape natnum → Boolean    Purpose: Determine that no tape elements read
(define (S−INV t i) (= i 1))


;; tape natum → Boolean
;; Purpose: Determine that tape[1..i−1] only has a
(define (B−INV t i)
  (and (>= i 2) (andmap (λ (s) (eq? s 'a)) (take (rest t) (sub1 i)))))


;; tape natnum → Boolean   Purpose: Determine that tape[1..i−1] only has a
(define (A−INV t i)
  (and (>= i 2)  (andmap (λ (s) (eq? s 'a)) (take (rest t) (sub1 i)))))


;; tape natnum → Boolean
;; Purpose: Determine that tape[1..i−2] has only a and tape[i−1] = b
(define (C−INV t i)
  (and (>= i 2) (andmap (λ (s) (eq? s 'a)) (take (rest t) (− i 2)))
       (eq? (list−ref t (sub1 i)) 'b)))


;; tape natnum → Boolean
;; Purpose: Determine that tape[i] = BLANK and tape[1..i−1] = a* or tape[1..i−1] =
        a*b
(define (Y−INV t i)
  (or (and (= i 2) (eq? (list−ref t (sub1 i)) BLANK))
      (andmap (λ (s) (eq? s 'a)) (take (rest t) (sub1 i)))
      (let* [(front (takef (rest t) (λ (s) (eq? s 'a))))
             (back (takef (drop t (add1 (length front))) (λ (s) (not (eq? s BLANK)))))]
        (equal? back '(b))))))
```

Turing Machines

Theorem

*State invariants hold when a*Ua*b is applied to w.*

Proof.

The proof, as before, is done by induction on, `n`, the number of steps taken by `a*Ua*b`. Let `a*Ua*b` = `(make-tm K Σ R S F Y)`.

□

Turing Machines

Theorem

*State invariants hold when a*Ua*b is applied to w.*

The proof, as before, is done by induction on, `n`, the number of steps taken by `a*Ua*b`. Let `a*Ua*b` = `(make-tm K Σ R S F Y)`.

Proof.

<u>Base case</u>: n = 0

If no steps are taken a*Ua*b may only be in S. By precondition, the head's position is 1. This means S-INV holds.

□

Turing Machines

Proof.

Inductive Step:
Assume: State invariants hold for a computation of length $n = k$
Show: State invariants hold for a computation of length $n = k + 1$

□

Turing Machines

Proof.

Inductive Step:
Assume: State invariants hold for a computation of length n = k
Show: State invariants hold for a computation of length n = k + 1

Let w = xcy, such that x,y∈Σ^*, |x|=k, and c∈{Σ ∪ {BLANK}}. The first k + 1 steps:

`(S 1 xcy)` $\vdash^*$ `(U r xcy)` $\vdash$ `(V s xcy), where V∈K` $\wedge$ `U∈K-{N Y}`

That is, the first k transitions take the machine to state U and move the head to position r without changing the contents of the tape

The k + 1 transition takes the machine to state V and leaves the head in position s without changing the contents of the tape

We must show that the state invariant holds for the k + 1 transition

Note that a rule of the form ((I @) (I ,RIGHT)) is never used because the machine never moves left and by precondition the head starts in position 1

□

Turing Machines

Proof.

We make an argument for each rule that may be used:

> ((S ,BLANK) (Y ,BLANK)): By inductive hypothesis, S-INV holds. This means that before using this rule nothing has been read from the input word because the head is in position 1. Reading the blank means the input word is empty. Thus, Y-INV holds.

□

Turing Machines

Proof.

We make an argument for each rule that may be used:

((S ,BLANK) (Y ,BLANK)): By inductive hypothesis, S-INV holds. This means that before using this rule nothing has been read from the input word because the head is in position 1. Reading the blank means the input word is empty. Thus, Y-INV holds.

((S a) (A ,RIGHT)): By inductive hypothesis, S-INV holds. This means that before using this rule nothing has been read from the input word because the head is in position 1. Using this rule means that the read part of the input word only contains a and that the head moves to position 2. Therefore, A-INV holds.

□

Turing Machines

Proof.

We make an argument for each rule that may be used:

((S ,BLANK) (Y ,BLANK)): By inductive hypothesis, S-INV holds. This means that before using this rule nothing has been read from the input word because the head is in position 1. Reading the blank means the input word is empty. Thus, Y-INV holds.

((S a) (A ,RIGHT)): By inductive hypothesis, S-INV holds. This means that before using this rule nothing has been read from the input word because the head is in position 1. Using this rule means that the read part of the input word only contains a and that the head moves to position 2. Therefore, A-INV holds.

((B b) (C ,RIGHT)): By inductive hypothesis, B-INV holds. This means that the read part of the input word is a member of a^* and, the head's position, $i \geq 2$. Reading a b means the read part of the input word is a member of a^*b and that $i \geq 2$ continues to hold. Thus, C-INV holds.

□

Turing Machines

Proof.

We make an argument for each rule that may be used:

((S ,BLANK) (Y ,BLANK)): By inductive hypothesis, S-INV holds. This means that before using this rule nothing has been read from the input word because the head is in position 1. Reading the blank means the input word is empty. Thus, Y-INV holds.

((S a) (A ,RIGHT)): By inductive hypothesis, S-INV holds. This means that before using this rule nothing has been read from the input word because the head is in position 1. Using this rule means that the read part of the input word only contains a and that the head moves to position 2. Therefore, A-INV holds.

((B b) (C ,RIGHT)): By inductive hypothesis, B-INV holds. This means that the read part of the input word is a member of a^* and, the head's position, $i \geq 2$. Reading a b means the read part of the input word is a member of a^*b and that $i \geq 2$ continues to hold. Thus, C-INV holds.

((C a) (N ,RIGHT)): By inductive hypothesis, C-INV holds. This means that the read part of the input word is a member of a^*b. Reading an a means the input word is not a member of a^* nor a^*b. Thus, N-INV holds.

□

Turing Machines

Theorem

$L = L(a*Ua*b)$

Turing Machines

Theorem

$L = L(a^*Ua^*b)$

Lemma

$w \in L \Leftrightarrow w \in L(a^*Ua^*b)$

Turing Machines

Theorem

$L = L(a{*}Ua{*}b)$

Lemma

$w \in L \Leftrightarrow w \in L(a{*}Ua{*}b)$

Proof.

(⇒) Assume w ∈ L. This means that w ∈ a^* or w ∈ a^*b. Given that state invariants always hold, a*Ua*b must halt in Y after reading w. Thus, w ∈ L(a*Ua*b).
(⇐) Assume w ∈ L(a*Ua*b). This means that a*Ua*b halts in Y after consuming w. Given that the invariants always hold, w ∈ a^* or w ∈ a^*b. Thus, w ∈ L. □

Lemma

$w \notin L \Leftrightarrow w \notin L(a{*}Ua{*}b)$

Proof.

By contraposition □

Recursively Enumerable Languages

`L = {w |w has equal number of a's, b's, and c's}`

Let's develop a design idea!

Recursively Enumerable Languages

`L = {w |w has equal number of a's, b's, and c's}`

Let's develop a design idea!

Maybe it is easier to design a multitape Turing machine?

Recursively Enumerable Languages

`L = {w |w has equal number of a's, b's, and c's}`

Let's develop a design idea!

Maybe it is easier to design a multitape Turing machine?

;; PRE (LM BLANK w) AND t0pos=1, t1pos=0, t2pos=0, t3pos=0

Recursively Enumerable Languages

`L = {w |w has equal number of a's, b's, and c's}`

Let's develop a design idea!

Maybe it is easier to design a multitape Turing machine?

;; PRE (LM BLANK w) AND t0pos=1, t1pos=0, t2pos=0, t3pos=0

```
1. Nondeterministically decide if the input is not empty.
   If so, go to 2. Otherwise, go to 3
2. Copy w to the auxiliary tapes
3. Traverse the auxiliary tape left as long as matching a's,
   b's, and c's are read
4. If a blank is read on all auxiliary tapes move to accept
```

Recursively Enumerable Languages

L = {w |w has equal number of a's, b's, and c's}

;; PRE (LM BLANK w) AND t0pos=1, t1pos=0, t2pos=0, t3pos=0

;; S: Nothing has been read

Recursively Enumerable Languages

```
L = {w |w has equal number of a's, b's, and c's}
```

;; PRE (LM BLANK w) AND t0pos=1, t1pos=0, t2pos=0, t3pos=0

;; S: Nothing has been read

;; C: Everything read is copied to an auxiliary tape such
;; For all j<i, {t1[i] t2[i] t3[i]} = {a b c}

Recursively Enumerable Languages

```
L = {w |w has equal number of a's, b's, and c's}
```

```
;; PRE (LM BLANK w) AND t0pos=1, t1pos=0, t2pos=0, t3pos=0

;;  S: Nothing has been read

;;  C: Everything read is copied to an auxiliary tape such
;;       For all j<i, {t1[i] t2[i] t3[i]} = {a b c}

;;  D: Everything read is copied to an auxiliary tape such
;;       For all j<=i, {t1[i] t2[i] t3[i]} = {a b c}
```

Recursively Enumerable Languages

L = {w |w has equal number of a's, b's, and c's}

```
;; PRE (LM BLANK w) AND t0pos=1, t1pos=0, t2pos=0, t3pos=0

;; S: Nothing has been read

;; C: Everything read is copied to an auxiliary tape such
;;       For all j<i, {t1[i] t2[i] t3[i]} = {a b c}

;; D: Everything read is copied to an auxiliary tape such
;;       For all j<=i, {t1[i] t2[i] t3[i]} = {a b c}

;; G: T0 copied to auxiliary tapes such that
;;       For all j>i, {t1[i] t2[i] t3[i]} = {a b c}
```

Recursively Enumerable Languages

```
L = {w |w has equal number of a's, b's, and c's}
```

;; PRE (LM BLANK w) AND t0pos=1, t1pos=0, t2pos=0, t3pos=0

;; S: Nothing has been read

;; C: Everything read is copied to an auxiliary tape such
;; For all j<i, {t1[i] t2[i] t3[i]} = {a b c}

;; D: Everything read is copied to an auxiliary tape such
;; For all j<=i, {t1[i] t2[i] t3[i]} = {a b c}

;; G: T0 copied to auxiliary tapes such that
;; For all j>i, {t1[i] t2[i] t3[i]} = {a b c}

;; Y: w has equal number of a's, b's, and c's

Recursively Enumerable Languages

L = {w |w has equal number of a's, b's, and c's}

```
(define EQABC-ND
  (make-mttm
    '(S Y C D G)
    `(a b c)
    'S
    '(Y)
    <transition relation>
    4
    'Y))

;; Tests for EQABC-ND
(check-reject? EQABC-ND
               `((,LM ,BLANK a a b b a c c) 1)
               `((@ ,BLANK a a b b a c c) 1)
               `((@ ,BLANK a a a) 1))
(check-accept? EQABC-ND
               `((,LM ,BLANK a c c b a b) 1)
               `((@ ,BLANK) 1)
               `((@ ,BLANK c c a b a b a b c) 1))
```

Recursively Enumerable Languages

```
(list
  (list '(S (_ _ _ _))  '(C (R R R R)))
  (list '(S (_ _ _ _))  '(G (R R R R)))
```

Recursively Enumerable Languages

```
(list
  (list '(S (_ _ _ _)) '(C (R R R R)))
  (list '(S (_ _ _ _)) '(G (R R R R)))


  ;; copy an a to any tape
  (list '(C (a _ _ _)) '(D (a a _ _)))
  (list '(D (a a _ _)) '(C (R R _ _)))
  (list '(C (a _ _ _)) '(D (a _ a _)))
  (list '(D (a _ a _)) '(C (R _ R _)))
  (list '(C (a _ _ _)) '(D (a _ _ a)))
  (list '(D (a _ _ a)) '(C (R _ _ R)))
```

Recursively Enumerable Languages

```
(list
 (list '(S (_ _ _ _)) '(C (R R R R)))
 (list '(S (_ _ _ _)) '(G (R R R R)))


 ;; copy an a to any tape
 (list '(C (a _ _ _)) '(D (a a _ _)))
 (list '(D (a a _ _)) '(C (R R _ _)))
 (list '(C (a _ _ _)) '(D (a _ a _)))
 (list '(D (a _ a _)) '(C (R _ R _)))
 (list '(C (a _ _ _)) '(D (a _ _ a)))
 (list '(D (a _ _ a)) '(C (R _ _ R)))


 ;; copy a b to any tape
 (list '(C (b _ _ _)) '(D (b b _ _)))
 (list '(D (b b _ _)) '(C (R R _ _)))
 (list '(C (b _ _ _)) '(D (b _ b _)))
 (list '(D (b _ b _)) '(C (R _ R _)))
 (list '(C (b _ _ _)) '(D (b _ _ b)))
 (list '(D (b _ _ b)) '(C (R _ _ R)))
 ;; copy a c to any tape
 (list '(C (c _ _ _)) '(D (c c _ _)))
 (list '(D (c c _ _)) '(C (R R _ _)))
 (list '(C (c _ _ _)) '(D (c _ c _)))
 (list '(D (c _ c _)) '(C (R _ R _)))
 (list '(C (c _ _ _)) '(D (c _ _ c)))
 (list '(D (c _ _ c)) '(C (R _ _ R)))
```

Recursively Enumerable Languages

```
;; match as, bs, and cs
(list '(C (_ _ _ _)) '(G (_ L L L)))
(list '(G (_ a b c)) '(G (_ L L L)))
(list '(G (_ a c b)) '(G (_ L L L)))
(list '(G (_ b a c)) '(G (_ L L L)))
(list '(G (_ b c a)) '(G (_ L L L)))
(list '(G (_ c a b)) '(G (_ L L L)))
(list '(G (_ c b a)) '(G (_ L L L)))
(list '(G (_ _ _ _)) '(Y (_ _ _ _))))
```

Recursively Enumerable Languages

DEMO!

Recursively Enumerable Languages

For Unrestricted Grammars attend Andrés' IFL 2025 talk!

Recursively Enumerable Languages

THANK YOU!!! :-)