

Part II:  
Regular  
Languages

Marco T.  
Morazán

Regular  
Expressions

Deterministic  
Finite  
Automata

Nondeterministic  
Finite  
Automata

Finite-State  
Automata and  
Regular  
Expressions

Regular  
Grammars

Pumping  
Theorem for  
Regular  
Languages

# Part II: Regular Languages

Marco T. Morazán

Seton Hall University

# Outline

## 1 Regular Expressions

## 2 Deterministic Finite Automata

## 3 Nondeterministic Finite Automata

## 4 Finite-State Automata and Regular Expressions

## 5 Regular Grammars

## 6 Pumping Theorem for Regular Languages

# Regular Expressions

- An English word is a string of juxtaposed letters found in the Roman alphabet: [a..z]
- As a Computer Science student and reader of this textbook, you are familiar with other languages

# Regular Expressions

- An English word is a string of juxtaposed letters found in the Roman alphabet: [a..z]
- As a Computer Science student and reader of this textbook, you are familiar with other languages
- Defining English words as strings is a representation choice
- Consider three different representations of the word *cat*:

English: "cat"

FSM: '(c a t)

Binary: 011000110110000101110100

- The representations of *cat* are different, but the same concept is being represented.

# Regular Expressions

- An English word is a string of juxtaposed letters found in the Roman alphabet: [a..z]
- As a Computer Science student and reader of this textbook, you are familiar with other languages
- Defining English words as strings is a representation choice
- Consider three different representations of the word *cat*:

English: "cat"

FSM: '(c a t)

Binary: 011000110110000101110100

- The representations of *cat* are different, but the same concept is being represented.
- A language may, for example, be represented as a set of strings, a set of lists, or a set of binary numbers
- Regardless of the representation, the same words in the English language are represented.

# Regular Expressions

- Languages are represented using a finite representation
- If a language is finite listing all the words in the language is a finite representation
- A finite representation for infinite languages is needed because all the words in an infinite language cannot be listed

# Regular Expressions

- Languages are represented using a finite representation
- If a language is finite listing all the words in the language is a finite representation
- A finite representation for infinite languages is needed because all the words in an infinite language cannot be listed
- A finite representation for a language must be written using a finite number of symbols
- If  $\Sigma$  is an alphabet used to write the finite representations of languages then all possible finite language representations are defined as  $\Sigma^*$
- This means that the language of finite language representations is countably infinite

# Regular Expressions

- Languages are represented using a finite representation
- If a language is finite listing all the words in the language is a finite representation
- A finite representation for infinite languages is needed because all the words in an infinite language cannot be listed
- A finite representation for a language must be written using a finite number of symbols
- If  $\Sigma$  is an alphabet used to write the finite representations of languages then all possible finite language representations are defined as  $\Sigma^*$
- This means that the language of finite language representations is countably infinite
- $2^{\Sigma^*}$ , however, is uncountable
- There is a countable number of finite language representations and an uncountable number of languages to represent

# Regular Expressions

- Languages are represented using a finite representation
- If a language is finite listing all the words in the language is a finite representation
- A finite representation for infinite languages is needed because all the words in an infinite language cannot be listed
- A finite representation for a language must be written using a finite number of symbols
- If  $\Sigma$  is an alphabet used to write the finite representations of languages then all possible finite language representations are defined as  $\Sigma^*$
- This means that the language of finite language representations is countably infinite
- $2^{\Sigma^*}$ , however, is uncountable
- There is a countable number of finite language representations and an uncountable number of languages to represent
- Therefore, a finite representation for each language does not exist
- The best that we can achieve is to develop a finite representation of some interesting languages
- As long as a representation is finite the majority of languages cannot be represented.

# Regular Expressions

- We start by considering languages formed by the union or the concatenation of words in two (not necessarily distinct) languages
- Such languages may be finitely represented using regular expressions

# Regular Expressions

- We start by considering languages formed by the union or the concatenation of words in two (not necessarily distinct) languages
- Such languages may be finitely represented using regular expressions
- A regular expression, over an alphabet  $\Sigma$ , is an FSM type instance:
  1. (empty-regexp)
  2. (singleton-regexp "a"), where  $a \in \Sigma$
  3. (union-regexp  $r_1\ r_2$ ), where  $r_1$  and  $r_2$  are regular expressions
  4. (concat-regexp  $r_1\ r_2$ ), where  $r_1$  and  $r_2$  are regular expressions
  5. (kleenestar-regexp  $r$ ), where  $r$  is a regular expression
- The language of a regular expression,  $r$ , is denoted by  $L(r)$
- It contains all the words that can be generated with  $r$
- A language that is described by a regular expression is called a *regular language*.

# Regular Expressions

- FSM provides informative error messages to help you overcome the misuse of constructors

# Regular Expressions

- FSM provides informative error messages to help you overcome the misuse of constructors
- ```
> (union-regexp 2 (singleton-regexp 'w))
the input to the regexp #(struct:singleton-regexp w)
must be a string
> (union-regexp (empty-regexp) 3)
3 must be a regexp to be a valid second input to
union-regexp #(struct:empty-regexp) 3
> (concat-regexp 3 (empty-regexp))
3 must be a regexp to be a valid first input to
concat-regexp 3 #(struct:empty-regexp)
> (kleenestar-regexp "A U B")
"A U B" must be a regexp to be a valid input to
kleenestar-regexp
> (singleton-regexp 1)
the input to the regexp #(struct:singleton-regexp 1)
must be a string
```

# Regular Expressions

- Selectors and Predicates

# Regular Expressions

- Selectors and Predicates
- singleton-regexp-a: Extracts the embedded string  
kleenestar-regexp-r1: Extracts the embedded regular expression  
union-regexp-r1: Extracts the first embedded regular expression  
union-regexp-r2: Extracts the second embedded regular expression  
concat-regexp-r1: Extracts the first embedded regular expression  
concat-regexp-r2: Extracts the second embedded regular expression

# Regular Expressions

- Selectors and Predicates
- singleton-regexp-a: Extracts the embedded string  
kleenestar-regexp-r1: Extracts the embedded regular expression  
union-regexp-r1: Extracts the first embedded regular expression  
union-regexp-r2: Extracts the second embedded regular expression  
concat-regexp-r1: Extracts the first embedded regular expression  
concat-regexp-r2: Extracts the second embedded regular expression
- empty-regexp?      singleton-regexp      kleenestar-regexp?  
union-regexp?      concat-regexp?

# Regular Expressions

- Selectors and Predicates
- singleton-regexp-a: Extracts the embedded string  
kleenestar-regexp-r1: Extracts the embedded regular expression  
union-regexp-r1: Extracts the first embedded regular expression  
union-regexp-r2: Extracts the second embedded regular expression  
concat-regexp-r1: Extracts the first embedded regular expression  
concat-regexp-r2: Extracts the second embedded regular expression
- 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)
         ...(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

- Observers

(gen-concat-word r f): Generate a word by concatenating words generated from the sub-regexps in the given concat-regexp using the given word-generating function.

(generate-ks-word n r f): Generate a word of arbitrary length in  $[0..n+1]$  using the given regular expression and the given word-generating function.

(extract-concat-regexpss r): Extracts all the nested regular expressions in a chain of concat-regexpss.

(extract-union-regexpss r): Extracts all the nested regular expressions in a chain of union-regexpss.

(pick-regexp r): Nondeterministically returns a nested sub-regexp from the given union-regexp. This includes any nested union-regexpss in a chain of union-regexpss.

(convert-singleton r): Converts the given singleton-regexp to a word of length 1 containing r's nested symbol or number.

(gen-regexp-word r): Nondeterministically generates a word in the language of the given regexp.

# Regular Expressions

- FSM provides `printable-regexp`:

|                       |                                                                       |
|-----------------------|-----------------------------------------------------------------------|
| (empty-regexp)        | the value of <code>EMP</code>                                         |
| (singleton-regexp a)  | "a"                                                                   |
| (union-regexp r1 r2)  | (string-append (printable-regexp r1)<br>"U"<br>(printable-regexp r2)) |
| (concat-regexp r1 r2) | (string-append (printable-regexp r1)<br>(printable-regexp r2))        |
| (kleenestar-regexp r) | (string-append (printable-regexp r1) "*")                             |

# Regular Expressions

- FSM provides `printable-regexp`:

|                                    |                                                                                      |
|------------------------------------|--------------------------------------------------------------------------------------|
| <code>(empty-regexp)</code>        | the value of <code>EMP</code>                                                        |
| <code>(singleton-regexp a)</code>  | "a"                                                                                  |
| <code>(union-regexp r1 r2)</code>  | <code>(string-append (printable-regexp r1)<br/>"U"<br/>(printable-regexp r2))</code> |
| <code>(concat-regexp r1 r2)</code> | <code>(string-append (printable-regexp r1)<br/>(printable-regexp r2))</code>         |
| <code>(kleenestar-regexp r)</code> | <code>(string-append (printable-regexp r1) "*")</code>                               |

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

# Regular Expressions

- We can now build a finite representation for some infinite languages.

# Regular Expressions

- We can now build a finite representation for some infinite languages.
- Consider the following language over  $\Sigma = \{a, b\}$ :  
$$L = \{w \mid w \text{ ends with an } a\}$$
- Every word in  $L$  must have at least one  $a$  at the end
- Before the last  $a$  there can be an arbitrary number of  $a$ s and  $b$ s

# Regular Expressions

- We can now build a finite representation for some infinite languages.
- Consider the following language over  $\Sigma = \{a, b\}$ :

$$L = \{w \mid w \text{ ends with an } a\}$$

- Every word in L must have at least one a at the end
- Before the last a there can be an arbitrary number of as and bs
- `#lang fsm`

- `(define ENDS-WITH-A (concat-regexp AUB* A))`

# Regular Expressions

- We can now build a finite representation for some infinite languages.
- Consider the following language over  $\Sigma = \{a, b\}$ :  
$$L = \{w \mid w \text{ ends with an } a\}$$
- Every word in L must have at least one a at the end
- Before the last a there can be an arbitrary number of as and bs
  - `#lang fsm`
  - `(define A (singleton-regexp "a"))`
- `(define ENDS-WITH-A (concat-regexp AUB* A))`

# Regular Expressions

- We can now build a finite representation for some infinite languages.
- Consider the following language over  $\Sigma = \{a, b\}$ :

$$L = \{w \mid w \text{ ends with an } a\}$$

- Every word in L must have at least one a at the end
- Before the last a there can be an arbitrary number of as and bs
  - `#lang fsm`
  - `(define A (singleton-regexp "a"))`

- `(define AUB* (kleenestar-regexp AUB))`
- `(define ENDS-WITH-A (concat-regexp AUB* A))`

# Regular Expressions

- We can now build a finite representation for some infinite languages.
- Consider the following language over  $\Sigma = \{a, b\}$ :

$$L = \{w \mid w \text{ ends with an } a\}$$

- Every word in L must have at least one a at the end
- Before the last a there can be an arbitrary number of as and bs
  - `#lang fsm`
  - `(define A (singleton-regexp "a"))`
  - `(define AUB (union-regexp A B))`
  - `(define AUB* (kleenestar-regexp AUB))`
  - `(define ENDS-WITH-A (concat-regexp AUB* A))`

# Regular Expressions

- We can now build a finite representation for some infinite languages.
- Consider the following language over  $\Sigma = \{a, b\}$ :

$$L = \{w \mid w \text{ ends with an } a\}$$

- Every word in L must have at least one a at the end
- Before the last a there can be an arbitrary number of as and bs
  - #lang fsm
  - (define A (singleton-regexp "a"))
  - (define B (singleton-regexp "b"))
  - (define AUB (union-regexp A B))
  - (define AUB\* (kleenestar-regexp AUB))
  - (define ENDS-WITH-A (concat-regexp AUB\* A))

# Regular Expressions

- We can now build a finite representation for some infinite languages.
- Consider the following language over  $\Sigma = \{a, b\}$ :

$$L = \{w \mid w \text{ ends with an } a\}$$

- Every word in L must have at least one a at the end
- Before the last a there can be an arbitrary number of as and bs
  - #lang fsm
  - (define A (singleton-regexp "a"))
  - (define B (singleton-regexp "b"))
  - (define AUB (union-regexp A B))
  - (define AUB\* (kleenestar-regexp AUB))
  - (define ENDS-WITH-A (concat-regexp AUB\* A))
  - (check-equal? (printable-regexp ENDS-WITH-A) "(a U b)\*a")

# Regular Expressions

- Consider the following language:

$\text{BIN-NUMS} = \{w \mid w \text{ is a binary number without leading zeroes}\}$

# Regular Expressions

- Consider the following language:  
 $\text{BIN-NUMS} = \{w \mid w \text{ is a binary number without leading zeroes}\}$
- Although the above definition may sound clear it is lacking
- It does not provide any details about the structure of the binary numbers in the language nor any indication on how to build such numbers

# Regular Expressions

- Consider the following language:  
 $\text{BIN-NUMS} = \{w \mid w \text{ is a binary number without leading zeroes}\}$
- Although the above definition may sound clear it is lacking
- It does not provide any details about the structure of the binary numbers in the language nor any indication on how to build such numbers
- We shall attempt to formally define BIN-NUMS using a regular expressions

# Regular Expressions

- - 1  $\Sigma = \{0, 1\}$
  - 2 The minimum length of a binary number is 1
  - 3 A binary number with a length greater than 1 cannot start with 0

# Regular Expressions

- ①  $\Sigma = \{0, 1\}$
- ② The minimum length of a binary number is 1
- ③ A binary number with a length greater than 1 cannot start with 0
- #lang fsm

```
(define ZERO (singleton-regexp "0"))
```

```
(define ONE (singleton-regexp "1"))
```

# Regular Expressions

- - ①  $\Sigma = \{0, 1\}$
  - ② The minimum length of a binary number is 1
  - ③ A binary number with a length greater than 1 cannot start with 0
- #lang fsm

```
(define ZERO (singleton-regexp "0"))

(define ONE (singleton-regexp "1"))

(define OU1* (kleenestar-regexp (union-regexp ZERO ONE)))
```
- ```
(define BIN-NUMS (union-regexp ZERO STARTS1))

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

# Regular Expressions

- ①  $\Sigma = \{0, 1\}$
- ② The minimum length of a binary number is 1
- ③ A binary number with a length greater than 1 cannot start with 0
- #lang fsm
  - (define ZERO (singleton-regexp "0"))
  - (define ONE (singleton-regexp "1"))
  - (define OU1\* (kleenestar-regexp (union-regexp ZERO ONE)))
  - (define STARTS1 (concat-regexp ONE OU1\*))
  - (define BIN-NUMS (union-regexp ZERO STARTS1))
  - (check-equal? (printable-regexp BIN-NUMS) "(0 U 1(0 U 1)\*)")

# Regular Expressions

- HOMEWORK: 1–4

# Regular Expressions

- Compare the two formulations for BIN-NUMS:

$\text{BIN-NUMS} = \{w \mid w \text{ is a binary number without leading zeroes}\}$

$\text{BIN-NUMS} = (0 \cup 1(0 \cup 1)^*)$

- The first provides a quick understanding of what the language BIN-NUMS represents. It lacks, however, any description for constructing words in BIN-NUMS

# Regular Expressions

- Compare the two formulations for BIN-NUMS:

$\text{BIN-NUMS} = \{w \mid w \text{ is a binary number without leading zeroes}\}$

$\text{BIN-NUMS} = (0 \cup 1(0 \cup 1)^*)$

- The first provides a quick understanding of what the language BIN-NUMS represents. It lacks, however, any description for constructing words in BIN-NUMS
- The second describes an algorithm for constructing binary numbers without leading zeroes.
- How are such binary numbers generated?

# Regular Expressions

- Compare the two formulations for BIN-NUMS:

$\text{BIN-NUMS} = \{w \mid w \text{ is a binary number without leading zeroes}\}$

$\text{BIN-NUMS} = (0 \cup 1(0 \cup 1)^*)$

- The first provides a quick understanding of what the language BIN-NUMS represents. It lacks, however, any description for constructing words in BIN-NUMS
- The second describes an algorithm for constructing binary numbers without leading zeroes.
- How are such binary numbers generated?
- Either generate 0 or generate 1 followed by an arbitrary number of 0s and 1s.

# Regular Expressions

- To simplify the discussion the maximum word length shall have a default value of 10
- Allow, however, for the user to define the default maximum length through an optional parameter

# Regular Expressions

- To simplify the discussion the maximum word length shall have a default value of 10
- Allow, however, for the user to define the default maximum length through an optional parameter
- Optional arguments are accumulated in a list whose name appears after a . in the function header

# Regular Expressions

- To simplify the discussion the maximum word length shall have a default value of 10
- Allow, however, for the user to define the default maximum length through an optional parameter
- Optional arguments are accumulated in a list whose name appears after a . in the function header
- ```
;;  [natnum>0] → word
;; Purpose: Generate a binary number without leading
;;           zeroes of length ≤ MAX-LENGTH
(define (generate-bn . n)

(define MAX-LENGTH (if (null? n) 10 (first n)))
:
...)
```

# Regular Expressions

- We are unable to write tests using `check-equals?` because words are generated from regular expressions that require a nondeterministic choice to be made

# Regular Expressions

- We are unable to write tests using `check-equals?` because words are generated from regular expressions that require a nondeterministic choice to be made
- `union-regexp` requires a sub-regexp to be chosen
- `kleene-star-regexp` requires the number of repetitions to be chosen

# Regular Expressions

- We are unable to write tests using `check-equals?` because words are generated from regular expressions that require a nondeterministic choice to be made
- `union-regexp` requires a sub-`regexp` to be chosen
- `kleene-star-regexp` requires the number of repetitions to be chosen
- To test such a function we use property-based testing
- We shall test that the generated words have the expected properties

# Regular Expressions

- We are unable to write tests using `check-equals?` because words are generated from regular expressions that require a nondeterministic choice to be made
- `union-regexp` requires a sub-`regexp` to be chosen
- `kleene-star-regexp` requires the number of repetitions to be chosen
- To test such a function we use property-based testing
- We shall test that the generated words have the expected properties
- Use rackunit's `check-pred`
- `check-pred` requires a predicate to test and input for the predicate

# Regular Expressions

- Any word generated must have the following properties:
  - ①  $w$  is a list (i.e., it cannot be  $\text{EMP}$ )
  - ②  $1 \leq (\text{length } w)$
  - ③  $w$  is ' $(0)$ ' or  $(\text{first } w)$  is 1
  - ④  $w$  only contains 0s and 1s

# Regular Expressions

- Any word generated must have the following properties:
  - ①  $w$  is a list (i.e., it cannot be  $\text{EMP}$ )
  - ②  $1 \leq (\text{length } w)$
  - ③  $w$  is ' $(0)$ ' or  $(\text{first } w)$  is 1
  - ④  $w$  only contains 0s and 1s
- To test we ignore the length limit

# Regular Expressions

- Any word generated must have the following properties:

- ① w is a list (i.e., it cannot be EMP)
- ②  $1 \leq (\text{length } w)$
- ③ w is '(0) or (first w) is 1
- ④ w only contains 0s and 1s

- To test we ignore the length limit

- `; ; 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))
```

# Regular Expressions

- Any word generated must have the following properties:
  - ① w is a list (i.e., it cannot be EMP)
  - ②  $1 \leq (\text{length } w)$
  - ③ w is '(0) or (first w) is 1
  - ④ w only contains 0s and 1s
- To test we ignore the length limit
- `; ; 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)`
- Anything returned by generate-bn must satisfy is-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))`

# Regular Expressions

- The body of generate-bn must generate a word starting with BIN-NUMS
- This task is delegated to an auxiliary function that knows how to generate words using the regexps it may receive as input:  
`(gen-word BIN-NUMS)`

# Regular Expressions

- `gen-word` generates a word representing a valid binary number
- It must be able to process any `regexp` subtype that is part of the `BIN-NUMS` definition: `singleton-regexp`, `concat-regexp`, `union-regexp`, and `kleenestar-regexp`
- The function template to process a `regexp` is specialized

# Regular Expressions

- `;; [natnum>0] → word Purpose: ...  
(define (generate-bn . n)  
 (define MAX-LENGTH (if (null? n) 10 (first n))))`

- `(gen-word BIN-NUMS MAX-LENGTH))`
- `;; word → Boolean Purpose: Test if the given word is a 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))`

# Regular Expressions

- $\cdot ; [natnum > 0] \rightarrow word$  Purpose: ...  
`(define (generate-bn . n)  
 (define MAX-LENGTH (if (null? n) 10 (first n))))`
- $\cdot ; ; regexp \rightarrow word$  Purpose: ...  
`(define (gen-word r . m)`

- `(gen-word BIN-NUMS MAX-LENGTH))`
- $\cdot ; ; word \rightarrow Boolean$  Purpose: Test if the given word is a 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))`

# Regular Expressions

- $\cdot ; [natnum > 0] \rightarrow word$  Purpose: ...  
`(define (generate-bn . n)  
 (define MAX-LENGTH (if (null? n) 10 (first n))))`
- $\cdot ; regexp \rightarrow word$  Purpose: ...  
`(define (gen-word r . m)`
- `(cond  
 [(singleton-regexp? r) (convert-singleton r)])`

- `(gen-word BIN-NUMS MAX-LENGTH))`
- $\cdot ; word \rightarrow Boolean$  Purpose: Test if the given word is a 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))`

# Regular Expressions

- $\cdot ; [natnum > 0] \rightarrow word$  Purpose: ...  
`(define (generate-bn . n)  
 (define MAX-LENGTH (if (null? n) 10 (first n))))`
- $\cdot ; ; regexp \rightarrow word$  Purpose: ...  
`(define (gen-word r . m)`
- $\cdot (cond$ 
  - $\cdot [(singleton-regexp? r) (convert-singleton r)]$
  - $\cdot [(concat-regexp? r)  
 (let [(w1 (gen-word (concat-regexp-r1 r)))  
 (w2 (gen-word (concat-regexp-r2 r)))]  
 (append w1 w2))]$
- $\cdot (gen-word BIN-NUMS MAX-LENGTH))$
- $\cdot ; ; word \rightarrow Boolean$  Purpose: Test if the given word is a 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))`

# Regular Expressions

- $\cdot ; [natnum > 0] \rightarrow word$  Purpose: ...  
`(define (generate-bn . n)  
 (define MAX-LENGTH (if (null? n) 10 (first n))))`
- $\cdot ; ; regexp \rightarrow word$  Purpose: ...  
`(define (gen-word r . m)`
- $\cdot (\text{cond}$ 
  - $\cdot [(\text{singleton-regexp? } r) (\text{convert-singleton } r)]$
  - $\cdot [(\text{concat-regexp? } r)  
 (\text{let } [(w1 (\text{gen-word} (\text{concat-regexp-r1 } r)))  
 (w2 (\text{gen-word} (\text{concat-regexp-r2 } r)))]  
 (\text{append } w1 w2)))]$
  - $\cdot [(\text{union-regexp? } r) (\text{gen-word} (\text{pick-regexp } r))]$
- $\cdot (\text{gen-word BIN-NUMS MAX-LENGTH})$
- $\cdot ; ; word \rightarrow Boolean$  Purpose: Test if the given word is a 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))`

# Regular Expressions

- $\cdot ; [natnum > 0] \rightarrow word$  Purpose: ...  
`(define (generate-bn . n)  
 (define MAX-LENGTH (if (null? n) 10 (first n))))`
- $\cdot ; ; regexp \rightarrow word$  Purpose: ...  
`(define (gen-word r . m)`
- $\cdot (\text{cond}$ 
  - $\cdot [(\text{singleton-regexp? } r) (\text{convert-singleton } r)]$
  - $\cdot [(\text{concat-regexp? } r)  
 (\text{let } [(w1 (\text{gen-word} (\text{concat-regexp-r1 } r)))  
 (w2 (\text{gen-word} (\text{concat-regexp-r2 } r)))]  
 (\text{append } w1 w2))]$
  - $\cdot [(\text{union-regexp? } r) (\text{gen-word} (\text{pick-regexp } r))]$
  - $\cdot [(\text{kleenestar-regexp? } r)  
 (\text{flatten} (\text{build-list}  
 (\text{pick-reps } MAX-KS-REPS)  
 (\lambda (i)  
 (\text{gen-word} (\text{kleenestar-regexp-r1 } r)))))))])])])$
- $\cdot (\text{gen-word } BIN-NUMS \text{ MAX-LENGTH})$
- $\cdot ; ; word \rightarrow Boolean$  Purpose: Test if the given word is a BIN-NUMS  
`(define (is-bin-nums? w)  
 (and (list? w) (<= 1 (length w))  
 (or (equal? w '(0)) (= (first w) 1))  
 (andmap (\lambda (bit) (or (= bit 0) (= bit 1))) w)))  
 (check-equal? (is-bin-nums? '()) #f))`

# Regular Expressions

- The development of `generate-bn` confirms that a regular expression,  $r$ , describes a construction algorithm for words in  $L(r)$
- This suggests that there is an algorithm to generate an arbitrary word in the language of an arbitrary regular expression

# Regular Expressions

- Design Idea
- Input a regular expression and an optional natural number for  $k$ s repetitions
- A constant, MAX-KLEENESTAR-REPS, is locally defined: the default value 20

# Regular Expressions

- Design Idea
- Input a regular expression and an optional natural number for `ks` repetitions
- A constant, `MAX-KLEENESTAR-REPS`, is locally defined: the default value 20
- Specialize the function template to process a `regexp`

# Regular Expressions

- Design Idea
- Input a regular expression and an optional natural number for  $k$ s repetitions
- A constant, MAX-KLEENESTAR-REPS, is locally defined: the default value 20
- Specialize the function template to process a `regexp`
- empty regular expression generates `EMP`

# Regular Expressions

- Design Idea
- Input a regular expression and an optional natural number for `ks` repetitions
- A constant, `MAX-KLEENESTAR-REPS`, is locally defined: the default value 20
- Specialize the function template to process a `regexp`
- empty regular expression generates `EMP`
- singleton regular expression use `convert-singleton`

# Regular Expressions

- Design Idea
- Input a regular expression and an optional natural number for `ks` repetitions
- A constant, `MAX-KLEENESTAR-REPS`, is locally defined: the default value 20
- Specialize the function template to process a `regexp`
- empty regular expression generates `EMP`
- singleton regular expression use `convert-singleton`
- Kleene star regular expression, nondeterministically pick the number of reps, generate that many words, filter out `EMPs` and flatten.

# Regular Expressions

- Design Idea
- Input a regular expression and an optional natural number for `ks` repetitions
- A constant, `MAX-KLEENESTAR-REPS`, is locally defined: the default value 20
- Specialize the function template to process a `regexp`
- empty regular expression generates `EMP`
- singleton regular expression use `convert-singleton`
- Kleene star regular expression, nondeterministically pick the number of reps, generate that many words, filter out `EMPs` and flatten.
- union regular expression, `pick-regexp` is used to nondeterministically select one of the expressions in the union to generate a word

# Regular Expressions

- Design Idea
- Input a regular expression and an optional natural number for `ks` repetitions
- A constant, `MAX-KLEENESTAR-REPS`, is locally defined: the default value 20
- Specialize the function template to process a `regexp`
- empty regular expression generates `EMP`
- singleton regular expression use `convert-singleton`
- Kleene star regular expression, nondeterministically pick the number of reps, generate that many words, filter out `EMPs` and flatten.
- union regular expression, `pick-regexp` is used to nondeterministically select one of the expressions in the union to generate a word
- concatenation regular expression generate two words with the embedded sub-`regexpss`. If both `EMP` return `EMP`. If one is `EMP` return the other. Otherwise, append both.

# Regular Expressions

- ```
;; regexp [natnum] → word
;; Purpose: Nondeterministically generate a word in the
;;           language of the given regexp. The maximum
;;           repetitions for a Kleene star is the given
;;           natnum if provided. Otherwise, it is 20
(define (gen-word rexp . reps)
```

# Regular Expressions

- ```
;; regexp [natnum] → word
;; Purpose: Nondeterministically generate a word in the
;;           language of the given regexp. The maximum
;;           repetitions for a Kleene star is the given
;;           natnum if provided. Otherwise, it is 20
(define (gen-word rexp . reps)
```

- ```
(check-pred is-bin-nums? (gen-word BIN-NUMS))
(check-pred is-bin-nums? (gen-word BIN-NUMS))
(check-pred is-bin-nums? (gen-word BIN-NUMS))
(check-pred is-bin-nums? (gen-word BIN-NUMS 30))
(check-pred is-bin-nums? (gen-word BIN-NUMS 50))
```

```
(check-pred is-ends-with-a? (gen-word ENDS-WITH-A))
```

# Regular Expressions

- `;; regexp [natnum] → word`  
`;; Purpose: Nondeterministically generate a word in the`  
`;; language of the given regexp. The maximum`  
`;; repetitions for a Kleene star is the given`  
`;; natnum if provided. Otherwise, it is 20`  
`(define (gen-word rexp . reps)`  
- `(cond [(empty-regexp? rexp) EMP]`

- `(check-pred is-bin-nums? (gen-word BIN-NUMS))`  
`(check-pred is-bin-nums? (gen-word BIN-NUMS))`  
`(check-pred is-bin-nums? (gen-word BIN-NUMS))`  
`(check-pred is-bin-nums? (gen-word BIN-NUMS 30))`  
`(check-pred is-bin-nums? (gen-word BIN-NUMS 50))`

# Regular Expressions

- ```
;; regexp [natnum] → word
;; Purpose: Nondeterministically generate a word in the
;;           language of the given regexp. The maximum
;;           repetitions for a Kleene star is the given
;;           natnum if provided. Otherwise, it is 20
(define (gen-word rexp . reps)
  (cond [(empty-regexp? rexp) '()]
        [(singleton-regexp? rexp) (convert-singleton rexp)]
```

- ```
(check-pred is-bin-nums? (gen-word BIN-NUMS))
(check-pred is-bin-nums? (gen-word BIN-NUMS))
(check-pred is-bin-nums? (gen-word BIN-NUMS))
(check-pred is-bin-nums? (gen-word BIN-NUMS 30))
(check-pred is-bin-nums? (gen-word BIN-NUMS 50))

(check-pred is-ends-with-a? (gen-word ENDS-WITH-A))
```

# Regular Expressions

- ```

•;; regexp [natnum] → word
•;; Purpose: Nondeterministically generate a word in the
•;;           language of the given regexp. The maximum
•;;           repetitions for a Kleene star is the given
•;;           natnum if provided. Otherwise, it is 20
(define (gen-word regexp . reps)
•  (cond [(empty-regexp? regexp) '()]
•        [((singleton-regexp? regexp) (convert-singleton regexp))]
•        [(kleenestar-regexp? regexp)
•         (let*
•          [(reps (pick-reps MAX-KLEENESTAR-REPS))
•           (low (flatten
•                  (filter
•                    (λ (w) (not (eq? w '())))
•                    (build-list
•                      reps
•                      (λ (i)
•                        (generate (kleenestar-regexp-r1 regexp)))))))
•           (if (empty? low) '() low))]

•        (check-pred is-bin-nums? (gen-word BIN-NUMS))
•        (check-pred is-bin-nums? (gen-word BIN-NUMS))
•        (check-pred is-bin-nums? (gen-word BIN-NUMS))
•        (check-pred is-bin-nums? (gen-word BIN-NUMS 30))
•        (check-pred is-bin-nums? (gen-word BIN-NUMS 50))

```

# Regular Expressions

- ```

• ;; regexp [natnum] → word
• ;; Purpose: Nondeterministically generate a word in the
• ;; language of the given regexp. The maximum
• ;; repetitions for a Kleene star is the given
• ;; natnum if provided. Otherwise, it is 20
• (define (gen-word regexp . reps)
•   (cond [(empty-regexp? regexp) '()]
•         [((singleton-regexp? regexp) (convert-singleton regexp))
•          '()]
•         [(kleenestar-regexp? regexp)
•          (let*
•            [(reps (pick-reps MAX-KLEENESTAR-REPS))
•             (low (flatten
•                   (filter
•                     (λ (w) (not (eq? w '())))
•                     (build-list
•                       reps
•                       (λ (i)
•                         (generate (kleenestar-regexp-r1 regexp)))))))
•             (if (empty? low) '() low))]
•            [((union-regexp? regexp) (gen-word (pick-regexp regexp)))]]
•
•   (check-pred is-bin-nums? (gen-word BIN-NUMS))
•   (check-pred is-bin-nums? (gen-word BIN-NUMS))
•   (check-pred is-bin-nums? (gen-word BIN-NUMS))
•   (check-pred is-bin-nums? (gen-word BIN-NUMS 30))
•   (check-pred is-bin-nums? (gen-word BIN-NUMS 50))

```

# Regular Expressions

- `;; regexp [natnum] → word`  
`;; Purpose: Nondeterministically generate a word in the`  
`;; language of the given regexp. The maximum`  
`;; repetitions for a Kleene star is the given`  
`;; natnum if provided. Otherwise, it is 20`  
`(define (gen-word rexp . reps)`
  - `(cond [(empty? rexp) EMP]`
  - `[(singleton-regexp? rexp) (convert-singleton rexp)]`
  - `[(kleenestar-regexp? rexp)`
    - `(let*`
      - `[(reps (pick-reps MAX-KLEENESTAR-REPS))]`
      - `(low (flatten`
        - `(filter`
          - `(λ (w) (not (eq? w EMP)))`
          - `(build-list`
            - `reps`
              - `(λ (i)`
                - `(generate (kleenestar-regexp-r1 rexp))))))])`
        - `(if (empty? low) EMP low))]`
      - `[(union-regexp? rexp) (gen-word (pick-regexp rexp))]`
      - `[else`
        - `(let [[(w1 (gen-word (concat-regexp-r1 rexp)))]`
          - `(w2 (gen-word (concat-regexp-r2 rexp)))]`
        - `(cond [(and (eq? w1 EMP) (eq? w2 EMP)) EMP]`
          - `[(eq? w1 EMP) w2]`
          - `[(eq? w2 EMP) w1]`
          - `[else (append w1 w2)])]]))`
      - `(check-pred is-bin-nums? (gen-word BIN-NUMS))`
      - `(check-pred is-bin-nums? (gen-word BIN-NUMS))`
      - `(check-pred is-bin-nums? (gen-word BIN-NUMS))`
      - `(check-pred is-bin-nums? (gen-word BIN-NUMS 30))`
      - `(check-pred is-bin-nums? (gen-word BIN-NUMS 50))`
    - `(check-pred is-ends-with-a? (gen-word ENDS-WITH-A))`

# Regular Expressions

- Make sure that all the tests pass

# Regular Expressions

- Make sure that all the tests pass
- Take time to appreciate what has been achieved
- A regular expression, simultaneously, is a description of,  $L$ , a regular language and a description of a construction algorithm for the words in  $L$
- *A construction algorithm specifies a language*
- The function designed is so versatile that it is part of FSM:  
`gen-regexp-word`

# Regular Expressions

- HOMEWORK: 10

# Regular Expressions

- Regular expressions may be used to describe data such as internet addresses, proteins, decimal numbers, and patterns to search for in text among others

# Regular Expressions

- Regular expressions may be used to describe data such as internet addresses, proteins, decimal numbers, and patterns to search for in text among others
- To illustrate the use of regular expressions we explore the problem of generating passwords

# Regular Expressions

- A password is a string that:
  - Has length  $\geq 10$
  - Includes at least one lowercase letter
  - Includes at least one uppercase letter
  - Includes at least one special character: \$, &, !, and \*

# Regular Expressions

- A password is a string that:

- Has length  $\geq 10$
- Includes at least one lowercase letter
- Includes at least one uppercase letter
- Includes at least one special character: \$, &, !, and \*

- The needed sets:

```
(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 x y z))  
(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 X Y Z))  
(define spcls '$ & ! *)
```

# Regular Expressions

- A password is a string that:

- Has length  $\geq 10$
- Includes at least one lowercase letter
- Includes at least one uppercase letter
- Includes at least one special character: \$, &, !, and \*

- The needed sets:

```
(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 x y z))
(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 X Y Z))
(define spcls '$ & ! *)
```

- The needed singleton-regexps:

```
(define lc (map
            (lambda (lcl) (singleton-regexp (symbol->string lcl)))
            lowers))
(define uc (map
            (lambda (ucl) (singleton-regexp (symbol->string ucl)))
            uppers))
(define spc (map
            (lambda (sc) (singleton-regexp (symbol->string sc)))
            spcls))
```

# Regular Expressions

- Six different orderings the required elements may appear in:

|       |       |       |
|-------|-------|-------|
| L U S | U L S | S U L |
| L S U | U S L | S L U |

- Arbitrary number of elements before and after each
- What do LOWER, UPPER, SPCHS, and ARBTRY need to be?

# Regular Expressions

- Six different orderings the required elements may appear in:

|       |       |       |
|-------|-------|-------|
| L U S | U L S | S U L |
| L S U | U S L | S L U |

- Arbitrary number of elements before and after each
- What do LOWER, UPPER, SPCHS, and ARBTRY need to be?
- ```
(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))))
```
- We need an auxiliary function

# Regular Expressions

- We can define each sub-language:

```
(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 Expressions

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

# Regular Expressions

- The language of passwords is a word in any of the languages defined for the different orderings of required elements
- It is defined using a union regular expression:

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

# Regular Expressions

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

# Regular Expressions

- DESIGN IDEA
- The constructor for a password takes no input and returns a string
- A potential new password is locally defined

# Regular Expressions

- DESIGN IDEA
- The constructor for a password takes no input and returns a string
- A potential new password is locally defined
- A word is generated by applying FSM's gen-regexp-word to PASSWD and then converting the generated word 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 word is generated.

# Regular Expressions

- DESIGN IDEA
- The constructor for a password takes no input and returns a string
- A potential new password is locally defined
- A word is generated by applying FSM's gen-regexp-word to PASSWD and then converting the generated word 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 word is generated.
- In order to prevent generated passwords from getting unwieldy long gen-regexp-word is given 5 as the maximum number of repetitions for a Kleene star regular expression

# Regular Expressions

- ```
;; --> string
;; Purpose: Generate a valid password
(define (generate-password))
```

# Regular Expressions

- ```
;;  --> string
;; Purpose: Generate a valid password
(define (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 Expressions

- ```
;;  --> string
;; Purpose: Generate a valid password
(define (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))
```

# Regular Expressions

- ```
;; --> string
;; Purpose: Generate a valid password
(define (generate-password)
```
- ```
(let [(new-passwd (passwd->string
                      (gen-regexp-word PASSWD 5)))]
  (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 Expressions

- ```
;; string → (listof symbol)
;; Purpose: Convert the given string to a list of symbols
(define (str->los str)
  (map (λ (c) (string->symbol (string c)))
        (string->list str)))

;; Tests
(check-equal? (str->los "") '())
(check-equal? (str->los "a!Cop") '(a ! C o p))
```

# Regular Expressions

- ```
;; string → (listof symbol)
;; Purpose: Convert the given string to a list of symbols
(define (str->los str)
  (map (λ (c) (string->symbol (string c)))
        (string->list str)))

;; Tests
(check-equal? (str->los "") '())
(check-equal? (str->los "a!Cop") '(a ! C o p))
```
- ```
;; word → string
;; Purpose: Convert the given word to a string
(define (passwd->string passwd)
  (list->string
    (map (λ (s)
            (first (string->list (symbol->string s))))
         passwd)))

;;Tests
(check-equal? (passwd->string '(a j h B ! ! y y t c))
               "ajhb!!yytc")
(check-equal? (passwd->string '($ u t q x ! J i n * K C))
               "$utqx!Jin*KC")
```

# Regular Expressions

- ```
;; (listof regexp) → union-regexp
;; Purpose: Create a union-regexp using the given list of
;;           regular expressions
(define (create-union-regexp L)
  (cond [(< (length L) 2)
         (error "list too short")]
        [(empty? (rest (rest L)))
         (union-regexp (first L) (second L))]
        [else
         (union-regexp (first L)
                      (create-union-regexp (rest L))))]

;; Tests
(check-equal?
  (create-union-regexp (list (first lc) (first uc)))
  (union-regexp (singleton-regexp "a")
                (singleton-regexp "A")))
(check-equal?
  (create-union-regexp
    (list (first lc) (fourth uc) (third spc)))
  (union-regexp (singleton-regexp "a")
                (union-regexp (singleton-regexp "D")
                              (singleton-regexp "!"))))
```

# Regular Expressions

- Run the program and make sure all the tests pass.

# Regular Expressions

- Run the program and make sure all the tests pass.
- These are sample passwords generated:

```
> (generate-password)
"*j$xL!&CjMK"
> (generate-password)
"&$&*&u&lBK&G*&$&U!E!"
> (generate-password)
"!$*D!&$F!f!"
> (generate-password)
"$!*!e0Y$!!*$$"
> (generate-password)
"$!bG&0&U&&!"
> (generate-password)
*xm!**yIT"
```

- The passwords generated appear fairly robust

# Regular Expressions

- HOMEWORK: 9, 13, 14
- QUIZ: 12 (due in 1 week)

# Deterministic Finite Automata

- Regular expressions define how to generate words in a regular language
- How can we decide if it is a member of a language?

# Deterministic Finite Automata

- Regular expressions define how to generate words in a regular language
- How can we decide if it is a member of a language?
- For this it is desirable to have some type of device or machine that takes as input a word and returns 'accept' if the given word is in the language and 'reject' if the given word is not in the language
- We need a model of a computer to determine if a word is part of a language
- How should such a machine operate?

# Deterministic Finite Automata

- Regular expressions define how to generate words in a regular language
- How can we decide if it is a member of a language?
- For this it is desirable to have some type of device or machine that takes as input a word and returns 'accept' if the given word is in the language and 'reject' if the given word is not in the language
- We need a model of a computer to determine if a word is part of a language
- How should such a machine operate?
- Analyzing how words in the language of a regular expression are generated can provide some insight
- Consider how a word is generated for the following regular expression:

```
(concat-regexp
  (union-regexp (singleton-regexp "a")
               (singleton-regexp "b"))
  (concat-regexp (kleenestar-regexp (singleton-regexp "a"))
                (singleton-regexp "b"))))
```

- Word generation traverses the structure of the regular expression
- An a or a b is generated
- An arbitrary number of as are generated
- A b is generated
- What does this tell you?

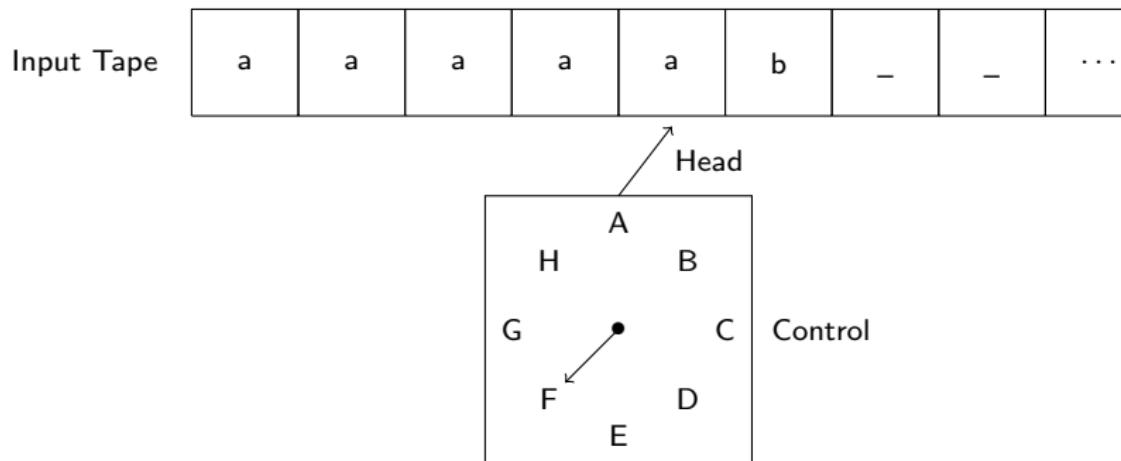
# Deterministic Finite Automata

- Regular expressions define how to generate words in a regular language
- How can we decide if it is a member of a language?
- For this it is desirable to have some type of device or machine that takes as input a word and returns 'accept' if the given word is in the language and 'reject' if the given word is not in the language
- We need a model of a computer to determine if a word is part of a language
- How should such a machine operate?
- Analyzing how words in the language of a regular expression are generated can provide some insight
- Consider how a word is generated for the following regular expression:

```
(concat-regexp
  (union-regexp (singleton-regexp "a")
               (singleton-regexp "b"))
  (concat-regexp (kleenestar-regexp (singleton-regexp "a"))
                (singleton-regexp "b"))))
```

- Word generation traverses the structure of the regular expression
- An a or a b is generated
- An arbitrary number of as are generated
- A b is generated
- What does this tell you?
- The elements of a word are generated from left to right
- Suggests that a word may be traversed from left to right to determine if it is in a language

# Deterministic Finite Automata



- The machine outlined above is called a *finite-state automaton* (or *finite-state machine*)
- A (very) restricted model of a computer
- Only capable of accepting or rejecting words
- Has no memory other than what exists in the processing module: it can only remember the state that it is

# Deterministic Finite Automata

## Definition

A deterministic finite-state automaton,  $\text{dfa}$ , is a (`make-dfa S Σs F δ ['no-dead']`)

- $\delta$  is a transition function: must contain a transition,  $(A \ a \ B)$ , for every element in  $S \times \Sigma$

# Deterministic Finite Automata

## Definition

A deterministic finite-state automaton,  $\text{dfa}$ , is a (`make-dfa S Σs F δ ['no-dead']`)

- $\delta$  is a transition function: must contain a transition,  $(A \ a \ B)$ , for every element in  $S \times \Sigma$
- The constructor automatically adds a *dead state*,  $ds$  (denoted by the FSM constant `DEAD`), and any missing transitions
- For any missing transition the added transition moves the machine to the dead state
- To inhibit the addition of the dead state the optional argument '`no-dead`' may be given to the constructor

# Deterministic Finite Automata

- A computation for a dfa,  $M$ , is denoted by a list of configurations that  $M$  traverses to consume the input word
- A configuration is a two-list that has the unconsumed part of the input word and the state the machine

# Deterministic Finite Automata

- A computation for a dfa,  $M$ , is denoted by a list of configurations that  $M$  traverses to consume the input word
- A configuration is a two-list that has the unconsumed part of the input word and the state the machine
- A transition made (or step taken) by the machine is denoted using  $\vdash$
- $C_i \vdash C_j$  is valid for  $M$  if and only if  $M$  can move from  $C_i$  to  $C_j$  using a single transition

# Deterministic Finite Automata

- A computation for a dfa,  $M$ , is denoted by a list of configurations that  $M$  traverses to consume the input word
- A configuration is a two-list that has the unconsumed part of the input word and the state the machine
- A transition made (or step taken) by the machine is denoted using  $\vdash$
- $C_i \vdash C_j$  is valid for  $M$  if and only if  $M$  can move from  $C_i$  to  $C_j$  using a single transition
- Zero or more moves by  $M$  is denoted using  $\vdash^*$ .
- $C_i \vdash^* C_j$  is valid for  $M$  if and only if  $M$  can move from  $C_i$  to  $C_j$  using zero or more transitions

# Deterministic Finite Automata

- A computation for a dfa,  $M$ , is denoted by a list of configurations that  $M$  traverses to consume the input word
- A configuration is a two-list that has the unconsumed part of the input word and the state the machine
- A transition made (or step taken) by the machine is denoted using  $\vdash$
- $C_i \vdash C_j$  is valid for  $M$  if and only if  $M$  can move from  $C_i$  to  $C_j$  using a single transition
- Zero or more moves by  $M$  is denoted using  $\vdash^*$ .
- $C_i \vdash^* C_j$  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  $M$  if the following is a valid computation:  
 $(w\ s) \vdash^* ('() q)$ , where  $q \in F$
- The language accepted by  $M$ ,  $L(M)$ , is the set of all strings accepted by  $M$

# Deterministic Finite Automata

- **Selectors**

- (sm-states m): Returns the states of the given machine
- (sm-sigma m): Returns the alphabet of the given machine
- (sm-rules m): Returns the transition relation of the given machine
- (sm-start m): Returns the start state of the given machine
- (sm-finals m): Returns the final states of the given machine
- (sm-type m): Returns a symbol denoting the machine type

# Deterministic Finite Automata

- Selectors

- (sm-states m): Returns the states of the given machine
- (sm-sigma m): Returns the alphabet of the given machine
- (sm-rules m): Returns the transition relation of the given machine
- (sm-start m): Returns the start state of the given machine
- (sm-finals m): Returns the final states of the given machine
- (sm-type m): Returns a symbol denoting the machine type

- Observers

- (sm-apply m w [n]): Applies the given machine to the given word. It returns either 'accept' or 'reject.'
- (sm-showtransitions m w [n]): Applies the given machine to the given word. It returns a list for the computation performed.

# Deterministic Finite Automata

- Selectors

- (sm-states  $m$ ): Returns the states of the given machine
- (sm-sigma  $m$ ): Returns the alphabet of the given machine
- (sm-rules  $m$ ): Returns the transition relation of the given machine
- (sm-start  $m$ ): Returns the start state of the given machine
- (sm-finals  $m$ ): Returns the final states of the given machine
- (sm-type  $m$ ): Returns a symbol denoting the machine type

- Observers

- (sm-apply  $m$   $w$  [ $n$ ]): Applies the given machine to the given word. It returns either 'accept' or 'reject'.
- (sm-showtransitions  $m$   $w$  [ $n$ ]): Applies the given machine to the given word. It returns a list for the computation performed.

- Testers

- (sm-test  $m$  [ $n$ ]): Applies  $m$  to 100 randomly generated words and returns a list of the results. The optional natural number specifies the number of tests to perform.
- (sm-sameresult?  $m1$   $m2$   $w$ ): Applies the two given machines,  $m1$  and  $m2$ , to,  $w$ , the given word and tests if the same result is obtained.
- (sm-testequiv?  $m1$   $m2$  [ $n$ ]): Applies the two given machines,  $m1$  and  $m2$ , to the same 100 randomly generated words and tests if they produce the same results. If they do true is returned. Otherwise, a list of the words for which the results differ is returned. The optional natural number specifies the number of words to test.

# Deterministic Finite Automata

- Machine Visualization

(sm-graph m): Returns a *transition diagram* rendered as a directed graph for the given machine.

(sm-visualize m ) [ (s p)\* ] : Starts the FSM visualization tool for the given machine. The optional two-lists contain a state of the given machine and a predicate invariant (we will soon discuss this in more detail).

- Example: abb-b.rkt

# Deterministic Finite Automata

- Design Recipe for 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 using unit tests and random testing
  - ⑦ Design, implement, and test an invariant predicate for each state
  - ⑧ Prove  $L = L(M)$

# Deterministic Finite Automata

- $L = \{w \mid w \in \{a\ b\}^* \wedge w \text{ has an even number of } a \text{ and an odd number of } b\}$

# Deterministic Finite Automata

- $L = \{w \mid w \in \{a\ b\}^* \wedge w \text{ has an even number of } a \text{ and an odd number of } b\}$
- ;; Name: EVEN-A-ODD-B  
;;  
;;  $\Sigma: \{a\ b\}$

# Deterministic Finite Automata

- $L = \{w \mid w \in \{a\ b\}^* \wedge w \text{ has an even number of } a \text{ and an odd number of } b\}$
- ;; Name: EVEN-A-ODD-B  
;;  
;;  $\Sigma: \{a\ b\}$
- ;; Tests for EVEN-A-ODD-B  
(check-equal? (sm-apply EVEN-A-ODD-B '()) 'reject)  
(check-equal? (sm-apply EVEN-A-ODD-B '(a b b a)) 'reject)  
(check-equal? (sm-apply EVEN-A-ODD-B '(b a b b a a))  
'reject)  
(check-equal? (sm-apply EVEN-A-ODD-B '(a b)) 'reject)  
(check-equal? (sm-apply EVEN-A-ODD-B '(a b b b b))  
'reject)  
(check-equal? (sm-apply EVEN-A-ODD-B '(b a b b a a b))  
'reject)  
(check-equal? (sm-apply EVEN-A-ODD-B '(b)) 'accept)  
(check-equal? (sm-apply EVEN-A-ODD-B '(a a b)) 'accept)  
(check-equal? (sm-apply EVEN-A-ODD-B '(a a a b a b b))  
'accept)

# Deterministic Finite Automata

- States
- As a word is processed the consumed input may contain:
  - ① an even number of a and an even number of b
  - ② an odd number of a and an odd number of b
  - ③ an even number of a and an odd number of b
  - ④ odd number of a and even number of b

# Deterministic Finite Automata

- States
- As a word is processed the consumed input may contain:
  - ① an even number of a and an even number of b
  - ② an odd number of a and an odd number of b
  - ③ an even number of a and an odd number of b
  - ④ odd number of a and even number of b
- When processing starts the consumed input has an even number of as and an even number of bs

# Deterministic Finite Automata

- States
- As a word is processed the consumed input may contain:
  - ① an even number of a and an even number of b
  - ② an odd number of a and an odd number of b
  - ③ an even number of a and an odd number of b
  - ④ odd number of a and even number of b
- When processing starts the consumed input has an even number of as and an even number of bs
- The state that represents that the consumed input has an even number of as and an odd number of bs must be the only final state.

# Deterministic Finite Automata

- States
- As a word is processed the consumed input may contain:
  - ① an even number of a and an even number of b
  - ② an odd number of a and an odd number of b
  - ③ an even number of a and an odd number of b
  - ④ odd number of a and even number of b
- When processing starts the consumed input has an even number of as and an even number of bs
- The state that represents that the consumed input has an even number of as and an odd number of bs must be the only final state.
- The states may be documented as follows:

```
;; States
;; S: even number of a and even number of b, start state
;; M: odd number of a and odd number of b
;; N: even number of a and odd number of b, final state
;; P: odd number of a and even number of b
```

# Deterministic Finite Automata

- Transition Function
  - $(S \ a \ P)$
  - $(S \ b \ N)$

# Deterministic Finite Automata

- Transition Function
  - $(S \ a \ P)$
  - $(S \ b \ N)$
- $(M \ a \ N)$
- $(M \ b \ P)$

# Deterministic Finite Automata

- Transition Function
  - $(S \ a \ P)$   
 $(S \ b \ N)$
  - $(M \ a \ N)$   
 $(M \ b \ P)$
  - $(N \ a \ M)$   
 $(N \ b \ S)$

# Deterministic Finite Automata

- Transition Function
  - $(S \ a \ P)$   
 $(S \ b \ N)$
  - $(M \ a \ N)$   
 $(M \ b \ P)$
  - $(N \ a \ M)$   
 $(N \ b \ S)$
  - $(P \ a \ S)$   
 $(P \ b \ M)$

# Deterministic Finite Automata

- Implementation

```
(define EVEN-A-ODD-B (make-dfa '(S M N P)
                                    '(a b)
                                    'S
                                    '(N)
                                    '((S a P)
                                      (S b N)
                                      (M a N)
                                      (M b P)
                                      (N a M)
                                      (N b S)
                                      (P a S)
                                      (P b M))
                                    'no-dead))
```

# Deterministic Finite Automata

- Implementation

```
(define EVEN-A-ODD-B (make-dfa '(S M N P)
                                     '(a b)
                                     'S
                                     '(N)
                                     '((S a P)
                                       (S b N)
                                       (M a N)
                                       (M b P)
                                       (N a M)
                                       (N b S)
                                       (P a S)
                                       (P b M))
                                     'no-dead))
```

- > (sm-test EVEN-A-ODD-B 20)  
'(((b a a a) reject)  
((a a b a a b b) accept)  
((b b a) reject)  
((a b a) accept)  
(( ) reject)  
((a a a a) reject)  
((b b b a a b b) accept)  
((b b b a b a b) accept)

:

# Deterministic Finite Automata

- State Invariants

- ```
;; word → Boolean
```

```
;; Purpose: Determine if given word has an even number  
;;           of a and an even number of b
```

```
(define (S-INV ci)  
    (and (even? (length (filter (λ (s) (eq? s 'a)) ci)))  
         (even? (length (filter (λ (s) (eq? s 'b)) ci)))))  
;; Tests for S-INV  
(check-equal? (S-INV '(a)) #f)  
(check-equal? (S-INV '(a b b b a)) #f)  
(check-equal? (S-INV '()) #t)  
(check-equal? (S-INV '(a a b b)) #t)
```

# Deterministic Finite Automata

- State Invariants

- ```
;; word → Boolean
```

```
;; Purpose: Determine if given word has an even number
;;           of a and an even number of b
```

```
(define (S-INV ci)
  (and (even? (length (filter (λ (s) (eq? s 'a)) ci)))
        (even? (length (filter (λ (s) (eq? s 'b)) ci)))))
```

```
;; Tests for S-INV
```

```
(check-equal? (S-INV '(a)) #f)
(check-equal? (S-INV '(a b b b a)) #f)
(check-equal? (S-INV '()) #t)
(check-equal? (S-INV '(a a b b)) #t)
```

- ```
;; word → Boolean
```

```
;; Purpose: Determine if given word has an odd number
;;           of a and an odd number of b
```

```
(define (M-INV ci)
  (and (odd? (length (filter (λ (s) (eq? s 'a)) ci)))
        (odd? (length (filter (λ (s) (eq? s 'b)) ci)))))
```

```
;; Tests for M-INV
```

```
(check-equal? (M-INV '(a)) #f)
(check-equal? (M-INV '(a b b b a)) #f)
(check-equal? (M-INV '(a b b b a a b)) #f)
(check-equal? (M-INV '(b a)) #t)
(check-equal? (M-INV '(b a a b a b)) #t)
```

# Deterministic Finite Automata

- ```
;; word → Boolean
;; Purpose: Determine if given word has an even number
;;           of a and an odd number of b
(define (N-INV ci)
  (and (even? (length (filter (λ (s) (eq? s 'a)) ci)))
        (odd? (length (filter (λ (s) (eq? s 'b)) ci))))
;; Tests for N-INV
(check-equal? (N-INV '()) #f)
(check-equal? (N-INV '(a b a b a)) #f)
(check-equal? (N-INV '(a b b a a b)) #f)
(check-equal? (N-INV '(b a a)) #t)
(check-equal? (N-INV '(a b a a b a b b b)) #t)
```

# Deterministic Finite Automata

- ```
;; word → Boolean
;; Purpose: Determine if given word has an even number
;;           of a and an odd number of b
(define (N-INV ci)
  (and (even? (length (filter (λ (s) (eq? s 'a)) ci)))
       (odd? (length (filter (λ (s) (eq? s 'b)) ci)))))
;; Tests for N-INV
(check-equal? (N-INV '()) #f)
(check-equal? (N-INV '(a b a b a)) #f)
(check-equal? (N-INV '(a b b a a b)) #f)
(check-equal? (N-INV '(b a a)) #t)
(check-equal? (N-INV '(a b a a b a b b b)) #t)
```
- ```
;; word → Boolean
;; Purpose: Determine if given word has an odd number
;;           of a and an even number of b
(define (P-INV ci)
  (and (odd? (length (filter (λ (s) (eq? s 'a)) ci)))
       (even? (length (filter (λ (s) (eq? s 'b)) ci)))))
;; Tests for P-INV
(check-equal? (P-INV '()) #f)
(check-equal? (P-INV '(a b)) #f)
(check-equal? (P-INV '(a b b a a b a)) #f)
(check-equal? (P-INV '(b a b)) #t)
(check-equal? (P-INV '(a b a a b b b)) #t)
```

# Deterministic Finite Automata

- For the required proofs we use the following notation:

$M = \text{EVEN-A-ODD-B}$

$\Sigma = (\text{sm-sigma } M)$

$F = (\text{sm-finals } M)$

$w \in \Sigma^*$

$c_i = \text{the consumed input}$

# Deterministic Finite Automata

## Theorem

*The state invariants hold when  $M$  is applied to  $w$ .*

- Proof by induction on the number of transitions,  $n$ ,  $M$  makes to consume  $w$ .

# Deterministic Finite Automata

## Theorem

*The state invariants hold when M is applied to w.*

- Proof by induction on the number of transitions,  $n$ , M makes to consume  $w$ .
- Base Case:  $n = 0$   
If  $n$  is 0 then the consumed input is '()' and M is in  $S$ . This means the consumed input has an even number of as and an even number of bs (0 of each). Therefore, S-INV holds.

# Deterministic Finite Automata

## Theorem

*The state invariants hold when  $M$  is applied to  $w$ .*

- Proof by induction on the number of transitions,  $n$ ,  $M$  makes to consume  $w$ .
- Base Case:  $n = 0$   
If  $n$  is 0 then the consumed input is '()' and  $M$  is in  $S$ . This means the consumed input has an even number of as and an even number of bs (0 of each). Therefore, S-INV holds.
- Inductive Step:  
Assume: State invariants hold for  $n = k$ .  
Show: State invariants hold for  $n = k+1$ .

# Deterministic Finite Automata

## Theorem

*The state invariants hold when M is applied to w.*

- Proof by induction on the number of transitions, n, M makes to consume w.
- Base Case: n = 0  
If n is 0 then the consumed input is '()' and M is in S. This means the consumed input has an even number of as and an even number of bs (0 of each). Therefore, S-INV holds.
- Inductive Step:  
Assume: State invariants hold for n = k.  
Show: State invariants hold for n = k+1.
- If n=k+1 then the consumed input cannot be '()' given that the machine must have consumed at least one symbol. Therefore, we can state that  $c_i=x_a$  such that  $|c_i|=k+1$ ,  $x \in \Sigma^*$  and  $a \in \Sigma$ . M's computation to consume  $c_i$  has k+1 steps:

$$(x a s) \vdash^k (a r) \vdash ('() q), \text{ where } r, q \in S$$

- Given that  $|x|=k$  the inductive hypothesis informs us that the state invariants hold when x is consumed by M

# Deterministic Finite Automata

## Theorem

*The state invariants hold when M is applied to w.*

- Proof by induction on the number of transitions, n, M makes to consume w.
- Base Case: n = 0  
If n is 0 then the consumed input is '()' and M is in S. This means the consumed input has an even number of as and an even number of bs (0 of each). Therefore, S-INV holds.
- Inductive Step:  
Assume: State invariants hold for n = k.  
Show: State invariants hold for n = k+1.
- If n=k+1 then the consumed input cannot be '()' given that the machine must have consumed at least one symbol. Therefore, we can state that  $c_i = x a$  such that  $|c_i| = k+1$ ,  $x \in \Sigma^*$  and  $a \in \Sigma$ . M's computation to consume  $c_i$  has k+1 steps:

$$(x a s) \vdash^k (a r) \vdash ('() q), \text{ where } r, q \in S$$

- Given that  $|x|=k$  the inductive hypothesis informs us that the state invariants hold when x is consumed by M
- We must show that the state invariants hold for the k+1 transition into q

# Deterministic Finite Automata

- (S a P): Assume S-INV holds. Consuming an  $a$  means  $c_i$  has an odd number of  $a$ s and an even number  $b$ s. Therefore, P-INV holds.

# Deterministic Finite Automata

- (S a P): Assume S-INV holds. Consuming an  $a$  means  $c_i$  has an odd number of  $a$ s and an even number  $b$ s. Therefore, P-INV holds.
- (S b N): Assume S-INV holds. Consuming a  $b$  means  $c_i$  has an even number of  $a$ s and an odd number  $b$ s. Therefore, N-INV holds.

# Deterministic Finite Automata

- (S a P): Assume S-INV holds. Consuming an  $a$  means  $c_i$  has an odd number of  $a$ s and an even number  $b$ s. Therefore, P-INV holds.
- (S b N): Assume S-INV holds. Consuming a  $b$  means  $c_i$  has an even number of  $a$ s and an odd number  $b$ s. Therefore, N-INV holds.
- (M a N): Assume M-INV holds. Consuming an  $a$  means  $c_i$  has an even number of  $a$ s and an odd number  $b$ s. Therefore, N-INV holds.

# Deterministic Finite Automata

- (S a P): Assume S-INV holds. Consuming an  $a$  means  $c_i$  has an odd number of  $a$ s and an even number  $b$ s. Therefore, P-INV holds.
- (S b N): Assume S-INV holds. Consuming a  $b$  means  $c_i$  has an even number of  $a$ s and an odd number  $b$ s. Therefore, N-INV holds.
- (M a N): Assume M-INV holds. Consuming an  $a$  means  $c_i$  has an even number of  $a$ s and an odd number  $b$ s. Therefore, N-INV holds.
- (M b P): Assume M-INV holds. Consuming a  $b$  means  $c_i$  has an odd number of  $a$ s and an even number  $b$ s. Therefore, P-INV holds.

# Deterministic Finite Automata

- (S a P): Assume S-INV holds. Consuming an  $a$  means  $c_i$  has an odd number of  $a$ s and an even number  $b$ s. Therefore, P-INV holds.
- (S b N): Assume S-INV holds. Consuming a  $b$  means  $c_i$  has an even number of  $a$ s and an odd number  $b$ s. Therefore, N-INV holds.
- (M a N): Assume M-INV holds. Consuming an  $a$  means  $c_i$  has an even number of  $a$ s and an odd number  $b$ s. Therefore, N-INV holds.
- (M b P): Assume M-INV holds. Consuming a  $b$  means  $c_i$  has an odd number of  $a$ s and an even number  $b$ s. Therefore, P-INV holds.
- (N a M): Assume N-INV holds. Consuming an  $a$  means  $c_i$  has an odd number of  $a$ s and an odd number  $b$ s. Therefore, M-INV holds.

# Deterministic Finite Automata

- (S a P): Assume S-INV holds. Consuming an a means ci has an odd number of as and an even number bs. Therefore, P-INV holds.
- (S b N): Assume S-INV holds. Consuming a b means ci has an even number of as and an odd number bs. Therefore, N-INV holds.
- (M a N): Assume M-INV holds. Consuming an a means ci has an even number of as and an odd number bs. Therefore, N-INV holds.
- (M b P): Assume M-INV holds. Consuming an b means ci has an odd number of as and an even number bs. Therefore, P-INV holds.
- (N a M): Assume N-INV holds. Consuming an a means ci has an odd number of as and an odd number bs. Therefore, M-INV holds.
- (N b S): Assume N-INV holds. Consuming an b means ci has an even number of as and an even number bs. Therefore, S-INV holds.

# Deterministic Finite Automata

- (S a P): Assume S-INV holds. Consuming an a means ci has an odd number of as and an even number bs. Therefore, P-INV holds.
- (S b N): Assume S-INV holds. Consuming a b means ci has an even number of as and an odd number bs. Therefore, N-INV holds.
- (M a N): Assume M-INV holds. Consuming an a means ci has an even number of as and an odd number bs. Therefore, N-INV holds.
- (M b P): Assume M-INV holds. Consuming an b means ci has an odd number of as and an even number bs. Therefore, P-INV holds.
- (N a M): Assume N-INV holds. Consuming an a means ci has an odd number of as and an odd number bs. Therefore, M-INV holds.
- (N b S): Assume N-INV holds. Consuming an b means ci has an even number of as and an even number bs. Therefore, S-INV holds.
- (P a S): Assume P-INV holds. Consuming an a means ci has an even number of as and an even number bs. Therefore, S-INV holds.

# Deterministic Finite Automata

- (S a P): Assume S-INV holds. Consuming an a means ci has an odd number of as and an even number bs. Therefore, P-INV holds.
- (S b N): Assume S-INV holds. Consuming a b means ci has an even number of as and an odd number bs. Therefore, N-INV holds.
- (M a N): Assume M-INV holds. Consuming an a means ci has an even number of as and an odd number bs. Therefore, N-INV holds.
- (M b P): Assume M-INV holds. Consuming an b means ci has an odd number of as and an even number bs. Therefore, P-INV holds.
- (N a M): Assume N-INV holds. Consuming an a means ci has an odd number of as and an odd number bs. Therefore, M-INV holds.
- (N b S): Assume N-INV holds. Consuming an b means ci has an even number of as and an even number bs. Therefore, S-INV holds.
- (P a S): Assume P-INV holds. Consuming an a means ci has an even number of as and an even number bs. Therefore, S-INV holds.
- (P b M): Assume P-INV holds. Consuming an b means ci has an odd number of as and an odd number bs. Therefore, M-INV holds

# Deterministic Finite Automata

- The proof that  $L(EVEN-A-ODD-B) = L$  is divided into two lemmas (i.e., two parts):
  - ①  $w \in L \Leftrightarrow w \in L(M)$
  - ②  $w \notin L \Leftrightarrow w \notin L(M)$

# Deterministic Finite Automata

Regular  
Expressions

Deterministic  
Finite  
Automata

Nondeterministic  
Finite  
Automata

Finite-State  
Automata and  
Regular  
Expressions

Regular  
Grammars

Pumping  
Theorem for  
Regular  
Languages

## Lemma

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

- ( $\Rightarrow$ ) Assume  $w \in L$ .

$w \in L$  means that  $w$  has an even number of as and an odd number of bs. The proof that state invariants hold when  $w$  is consumed means that  $M$  can only halt in  $N$ , which is a final state. Therefore,  $w \in L(M)$ .

# Deterministic Finite Automata

Regular  
Expressions

Deterministic  
Finite  
Automata

Nondeterministic  
Finite  
Automata

Finite-State  
Automata and  
Regular  
Expressions

Regular  
Grammars

Pumping  
Theorem for  
Regular  
Languages

## Lemma

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

- ( $\Rightarrow$ ) Assume  $w \in L$ .  
 $w \in L$  means that  $w$  has an even number of as and an odd number of bs.  
The proof that state invariants hold when  $w$  is consumed means that  $M$  can only halt in  $N$ , which is a final state. Therefore,  $w \in L(M)$ .
- ( $\Leftarrow$ ) Assume  $w \in L(M)$ .  
 $w \in L(M)$  means that  $M$  halts in  $N$ .  $N$ 's invariant guarantees that  $w$  has an even number of as and an odd number of bs. Therefore,  $w \in L$ .

# Deterministic Finite Automata

## Lemma

$$w \notin L \Leftrightarrow w \notin L(M)$$

- ( $\Rightarrow$ ) Assume  $w \notin L$ .

$w \notin L$  means that  $w$  does not have an even number of as and an odd number of bs. This means  $M$  does not halt in  $N$  after consuming  $w$ . Given that the state invariants always hold, this means that  $w \notin L(M)$ .

# Deterministic Finite Automata

## Lemma

$$w \notin L \Leftrightarrow w \notin L(M)$$

- ( $\Rightarrow$ ) Assume  $w \notin L$ .  
 $w \notin L$  means that  $w$  does not have an even number of as and an odd number of bs. This means  $M$  does not halt in  $N$  after consuming  $w$ . Given that the state invariants always hold, this means that  $w \notin L(M)$ .
- ( $\Leftarrow$ ) Assume  $w \notin L(M)$ .  
 $M$  does not halt in  $N$  (the only final state). Given that the state invariants always hold, this means that  $w$  does not have an even number of as and an odd number of bs. Therefore,  $w \notin L$ .

# Deterministic Finite Automata

## Theorem

$$L = L(\text{EVEN-A-ODD-B})$$

- The two previous lemmas establish the theorem.

# Deterministic Finite Automata

- HOMEWORK: 1, 2, 4, 5, 7, 12
- QUIZ: 17 (due in 1 week)

# Deterministic Finite Automata

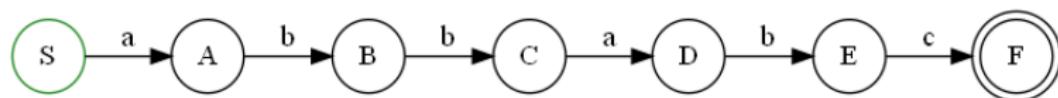
- Pattern Detection (i.e., Ctrl-F)

# Deterministic Finite Automata

- Pattern Detection (i.e., Ctrl-F)
- Given a pattern,  $patt$ , and an alphabet,  $\sigma$ , the DFA built needs  $|patt|+1$  states
- These states form the DFA's backbone and lead from the starting to the final state consuming the pattern

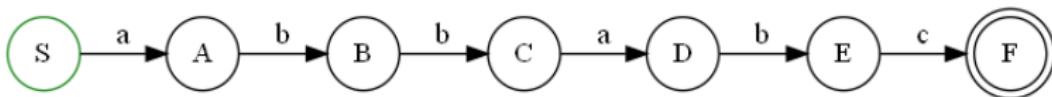
# Deterministic Finite Automata

- Pattern Detection (i.e., Ctrl-F)
- Given a pattern,  $patt$ , and an alphabet,  $\sigma$ , the DFA built needs  $|patt|+1$  states
- These states form the DFA's backbone and lead from the starting to the final state consuming the pattern
- The DFA backbone for the pattern '(a b b a b c)' has the following structure:



- Not the complete DFA because it is missing the transitions for when the next symbol in a given word does not match the next symbol in the pattern
- If the machine is in state E and the next symbol in the input word is not 'c', what state should the machine move to?

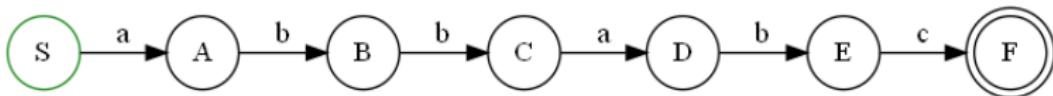
# Deterministic Finite Automata



- Reason about what the states mean (i.e., their invariant properties). For the machine's backbone above we can state:

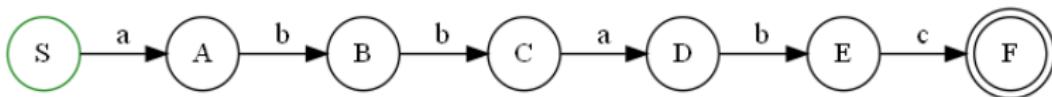
- S Nothing in the pattern has been matched
- A a has been matched
- B ab has been matched
- C abb has been matched
- D abba has been matched
- E abbab has been matched
- F abbabc has been matched

# Deterministic Finite Automata



- Reason about what the states mean (i.e., their invariant properties). For the machine's backbone above we can state:
  - S Nothing in the pattern has been matched
  - A a has been matched
  - B ab has been matched
  - C abb has been matched
  - D abba has been matched
  - E abbab has been matched
  - F abbabc has been matched
- What state should the machine be in after reading '(a b b a b b)'?

# Deterministic Finite Automata



- Reason about what the states mean (i.e., their invariant properties). For the machine's backbone above we can state:

- S Nothing in the pattern has been matched
- A a has been matched
- B ab has been matched
- C abb has been matched
- D abba has been matched
- E abbab has been matched
- F abbabc has been matched

- What state should the machine be in after reading '(a b b a b b)'?
- Observe that longest suffix of the read input that matches the beginning of the pattern is '(a b b)'
- The machine needs to move to state C.

# Deterministic Finite Automata

Regular  
Expressions

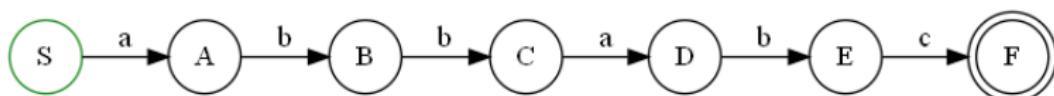
Deterministic  
Finite  
Automata

Nondeterministic  
Finite  
Automata

Finite-State  
Automata and  
Regular  
Expressions

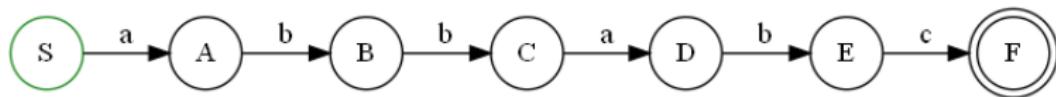
Regular  
Grammars

Pumping  
Theorem for  
Regular  
Languages



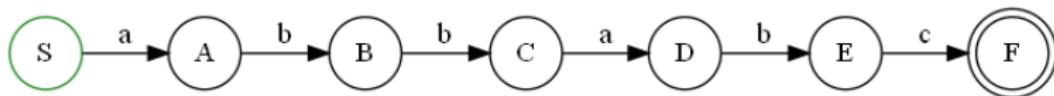
- How are the transition rules computed for when the next input symbol does not match the next symbol in the pattern?
- We say that the part of the pattern matched for each state represents the core prefix of the state
- For B the core prefix is '(a b)
- For E the core prefix is '(a b b a b)

# Deterministic Finite Automata



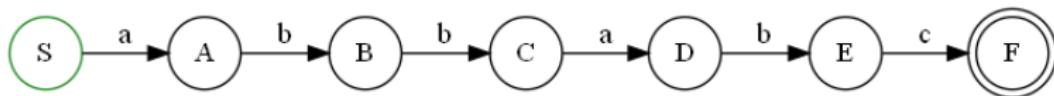
- We use core prefixes to compute the transitions needed for each state

# Deterministic Finite Automata



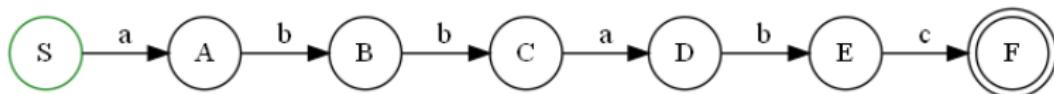
- We use core prefixes to compute the transitions needed for each state
- Assume the states are kept in a list such that the states appear in the direction of the arrows: `states = '(S A B C D E F)`

# Deterministic Finite Automata



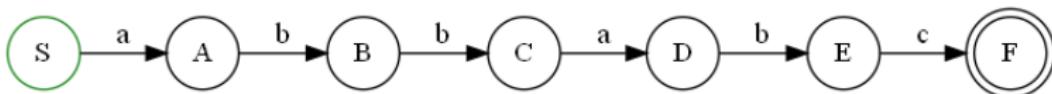
- We use core prefixes to compute the transitions needed for each state
- Assume the states are kept in a list such that the states appear in the direction of the arrows: states = '(S A B C D E F)
- What are the needed transitions out of E?

# Deterministic Finite Automata



- We use core prefixes to compute the transitions needed for each state
- Assume the states are kept in a list such that the states appear in the direction of the arrows: states = '(S A B C D E F)
- What are the needed transitions out of E?
- E's core prefix is '(a b b a b)
- The next input symbol may be a, b, or c
- For each, identify longest suffix that matches the pattern's beginning

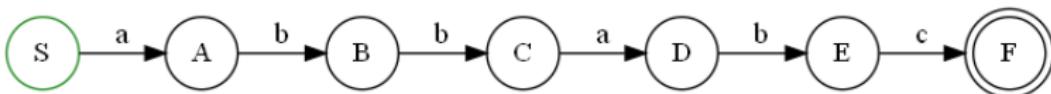
# Deterministic Finite Automata



- We use core prefixes to compute the transitions needed for each state
- Assume the states are kept in a list such that the states appear in the direction of the arrows: states = '(S A B C D E F)
- What are the needed transitions out of E?
- E's core prefix is '(a b b a b)
- The next input symbol may be a, b, or c
- For each, identify longest suffix that matches the pattern's beginning
- For a we have:

Last 6 symbols of consumed input: '(a b b a b a)

# Deterministic Finite Automata



- We use core prefixes to compute the transitions needed for each state
- Assume the states are kept in a list such that the states appear in the direction of the arrows: states = '(S A B C D E F)
- What are the needed transitions out of E?
- E's core prefix is '(a b b a b)
- The next input symbol may be a, b, or c
- For each, identify longest suffix that matches the pattern's beginning
- For a we have:

Last 6 symbols of consumed input: '(a b b a b a)
- Compare successively shorter suffixes with the pattern's beginning until a match is found or the suffix is empty:

Pattern: '(a b b a b c)

Suffix: '(a b b a b a) → does not match

'(b b a b a) → does not match

'(b a b a) → does not match

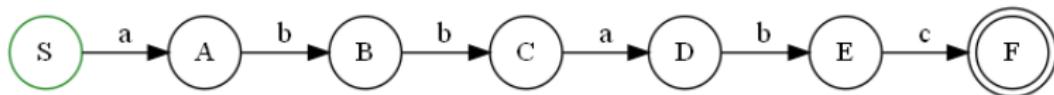
'(a b a) → does not match

'(b a) → does not match

'(a) → match

- The longest matching suffix with the pattern's beginning is A's core prefix
- This means that the machine must transition to A: (E a A)

# Deterministic Finite Automata



- For b:

Pattern: '(a b b a b c)

Suffix: '(a b b a b b) → does not match

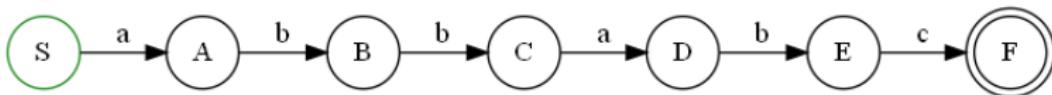
'(b b a b b) → does not match

'(b a b b) → does not match

'(a b b) → match

- The longest matching suffix with the pattern's beginning is C's core prefix
- The machine needs to transition to C: (E b C)

# Deterministic Finite Automata



- For b:

Pattern: '(a b b a b c)

Suffix: '(a b b a b b) → does not match

'(b b a b b) → does not match

'(b a b b) → does not match

'(a b b) → match

- The longest matching suffix with the pattern's beginning is C's core prefix
- The machine needs to transition to C: (E b C)
- For c:

Pattern: '(a b b a b c)

Suffix: '(a b b a b c) → match

- The longest matching suffix with the pattern's beginning is F's core prefix
- The needed transition is (E c F)

# Deterministic Finite Automata

Regular  
Expressions

Deterministic  
Finite  
Automata

Nondeterministic  
Finite  
Automata

Finite-State  
Automata and  
Regular  
Expressions

Regular  
Grammars

Pumping  
Theorem for  
Regular  
Languages

- `states = '(S A B C D E F)`
- Have you noticed the pattern for the destination state in each of the computed transition rules?

# Deterministic Finite Automata

Regular  
Expressions

Deterministic  
Finite  
Automata

Nondeterministic  
Finite  
Automata

Finite-State  
Automata and  
Regular  
Expressions

Regular  
Grammars

Pumping  
Theorem for  
Regular  
Languages

- `states = '(S A B C D E F)`
- Have you noticed the pattern for the destination state in each of the computed transition rules?
- It is always `(list-ref states (length lsuffix))`, where `lsuffix` is the longest matching suffix
- $$\begin{aligned} (\text{list-ref states } (\text{length } '(\text{a}))) &= \text{A} \\ (\text{list-ref states } (\text{length } '(\text{a b b}))) &= \text{C} \\ (\text{list-ref states } (\text{length } '(\text{a b b a b c}))) &= \text{F} \end{aligned}$$

# Deterministic Finite Automata

- For a given pattern,  $patt$ , and a given input alphabet,  $\sigma$ , the goal is to build a dfa for the following language:

$$L = \{w \mid w \text{ contains } patt\}$$

# Deterministic Finite Automata

- For a given pattern, *patt*, and a given input alphabet, *sigma*, the goal is to build a dfa for the following language:

$$L = \{w \mid w \text{ contains } patt\}$$

- The constructor needs to:
  - Generate the states for the new dfa
  - Compute the core prefix for each state
  - Compute the transitions for the new dfa

# Deterministic Finite Automata

- For a given pattern, *patt*, and a given input alphabet, *sigma*, the goal is to build a dfa for the following language:  
$$L = \{w \mid w \text{ contains } patt\}$$
- The constructor needs to:
  - Generate the states for the new dfa
  - Compute the core prefix for each state
  - Compute the transitions for the new dfa
- We choose to define the first state in the list of generated states as the starting state and the last state generated as the final state
- To generate a state the FSM function `generate-symbol` is used

# Deterministic Finite Automata

- For a given pattern,  $patt$ , and a given input alphabet,  $\sigma$ , the goal is to build a dfa for the following language:  
$$L = \{w \mid w \text{ contains } patt\}$$
- The constructor needs to:
  - Generate the states for the new dfa
  - Compute the core prefix for each state
  - Compute the transitions for the new dfa
- We choose to define the first state in the list of generated states as the starting state and the last state generated as the final state
- To generate a state the FSM function `generate-symbol` is used
- The generation of the core prefixes may be done so as to correspond with the list of generated states.
- The core prefix for the  $i^{\text{th}}$  state is given by taking the first  $i$  elements of the pattern.

# Deterministic Finite Automata

- For a given pattern,  $patt$ , and a given input alphabet,  $\sigma$ , the goal is to build a dfa for the following language:

$$L = \{w \mid w \text{ contains } patt\}$$

- The constructor needs to:
  - Generate the states for the new dfa
  - Compute the core prefix for each state
  - Compute the transitions for the new dfa
- We choose to define the first state in the list of generated states as the starting state and the last state generated as the final state
- To generate a state the FSM function `generate-symbol` is used
- The generation of the core prefixes may be done so as to correspond with the list of generated states.
- The core prefix for the  $i^{\text{th}}$  state is given by taking the first  $i$  elements of the pattern.
- To generate the transition function the needed transitions for each state,  $s$ , may be generated using the states, the input alphabet, the core prefix for  $s$ , and the pattern

# Deterministic Finite Automata

- ;; word alphabet → dfa  
;; Purpose: Build a dfa for  $L = \text{all words that contain the given pattern}$   
;;  
(define (build-pattern-dfa patt sigma))

# Deterministic Finite Automata

- ;; word alphabet → dfa  
;; Purpose: Build a dfa for  $L = \text{all words that contain the given pattern}$   
;;  
(define (build-pattern-dfa patt sigma))
- ;; Tests for build-pattern-dfa  
(define M (build-pattern-dfa '(a b b a) '(a b)))  
(define N (build-pattern-dfa '(a d) '(a b c d)))  
  
(check-equal? (sm-apply M '()) 'reject)  
(check-equal? (sm-apply M '(a a b b b a)) 'reject)  
(check-equal? (sm-apply M '(b b b a a a b b)) 'reject)  
(check-equal? (sm-apply M '(a b b a)) 'accept)  
(check-equal? (sm-apply M '(b b a a a b b a b b a)) 'accept)  
(check-equal? (sm-apply M '(a b b b a b b a)) 'accept)

# Deterministic Finite Automata

- ```
;; word alphabet → dfa
;; Purpose: Build a dfa for L = all words that contain the
;;                                     given pattern
(define (build-pattern-dfa patt sigma)
  (let* [(sts (build-list (add1 (length patt)))
               (λ (n) (generate-symbol 'A '(A))))]
```
- ```
;; Tests for build-pattern-dfa
(define M (build-pattern-dfa '(a b b a) '(a b)))
(define N (build-pattern-dfa '(a d) '(a b c d)))

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

# Deterministic Finite Automata

- ```
;; word alphabet → dfa
;; Purpose: Build a dfa for L = all words that contain the
;; given pattern
(define (build-pattern-dfa patt sigma)
  (let* [(sts (build-list (add1 (length patt)))
               (λ (n) (generate-symbol 'A '(A))))
         (core-prefixes (build-list (add1 (length patt))
                                   (λ (i) (take patt i))))]
```
- ```
;; Tests for build-pattern-dfa
(define M (build-pattern-dfa '(a b b a) '(a b)))
(define N (build-pattern-dfa '(a d) '(a b c d)))

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

# Deterministic Finite Automata

- ```
;; word alphabet → dfa
;; Purpose: Build a dfa for L = all words that contain the
;; given pattern
(define (build-pattern-dfa patt sigma))
```
- ```
(let* [(sts (build-list (add1 (length patt))
                           (λ (n) (generate-symbol 'A '(A)))))])
```
- ```
(core-prefixes (build-list (add1 (length patt))
                               (λ (i) (take patt i)))))
```
- ```
(deltas (append-map
              (λ (s cp)
                (gen-state-trans s sts sigma cp patt)))
              sts
              core-prefixes))]
```
- ```
;; Tests for build-pattern-dfa
(define M (build-pattern-dfa '(a b b a) '(a b)))
(define N (build-pattern-dfa '(a d) '(a b c d)))
```

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

## Deterministic Finite Automata

- ```

• ;; word alphabet → dfa
• ;; Purpose: Build a dfa for L = all words that contain the
• ;;                                     given pattern
• ;;
• (define (build-pattern-dfa patt sigma)
•   (let* [(sts (build-list (add1 (length patt))
•                         (λ (n) (generate-symbol 'A '(A)))))
•          (core-prefixes (build-list (add1 (length patt))
•                               (λ (i) (take patt i))))
•          (deltas (append-map
•                   (λ (s cp)
•                     (gen-state-trans s sts sigma cp patt)))
•                  sts
•                  core-prefixes))]
•     (make-dfa sts sigma (first sts) (list (last sts)) deltas 'no-dead)
•   ;; Tests for build-pattern-dfa
•   (define M (build-pattern-dfa '(a b b a) '(a b)))
•   (define N (build-pattern-dfa '(a d) '(a b c d)))
•
•   (check-equal? (sm-apply M '()) 'reject)
•   (check-equal? (sm-apply M '(a a b b b a)) 'reject)
•   (check-equal? (sm-apply M '(b b b a a a b b)) 'reject)
•   (check-equal? (sm-apply M '(a b b a)) 'accept)
•   (check-equal? (sm-apply M '(b b a a a b b a b b a)) 'accept)
•   (check-equal? (sm-apply M '(a b b b a b b a)) 'accept)

```

# Deterministic Finite Automata

- `;; state (listof state) alphabet word word → (listof dfa-rule)`  
;; Purpose: Generate failed match transitions for the  
;; given state  
`(define (gen-state-trans s states sigma cp patt)`

# Deterministic Finite Automata

- ```
;; state (listof state) alphabet word word → (listof dfa-rule)
;; Purpose: Generate failed match transitions for the
;;           given state
(define (gen-state-trans s states sigma cp patt)
```
- ```
;; Tests for gen-state-trans
(check-equal?
  (gen-state-trans 'E
    '(S A B C D E F)
    '(a b c)
    '(a b b a b)
    '(a b b a b c))
  '((E a A) (E b C) (E c F)))

(check-equal?
  (gen-state-trans 'S
    '(S A B C D E F)
    '(a b c)
    '()
    '(a b b a b c))
  '((S a A) (S b S) (S c S)))
```

:

# Deterministic Finite Automata

- `;; state (listof state) alphabet word word → (listof dfa-rule)`  
`;; Purpose: Generate failed match transitions for the`  
`;; given state`  
`(define (gen-state-trans s states sigma cp patt))`
- `(map (λ (a)`  
`(gen-state-tran s (append cp (list a)) patt states a))`  
`sigma))`
- `;; Tests for gen-state-trans`  
`(check-equal?`  
`(gen-state-trans 'E`  
`'(S A B C D E F)`  
`'(a b c)`  
`'(a b b a b)`  
`'(a b b a b c))`  
`'((E a A) (E b C) (E c F)))`
- `(check-equal?`  
`(gen-state-trans 'S`  
`'(S A B C D E F)`  
`'(a b c)`  
`'()`  
`'(a b b a b c))`  
`'((S a A) (S b S) (S c S))))`

:

# Deterministic Finite Automata

- ```
;; word word word (listof state) symbol → dfa-rule
;; Purpose: Generate dfa rule for given state and given word
;;           to match in the given pattern
(define (gen-state-tran s to-match patt states last-read)
```

# Deterministic Finite Automata

- ```
;; word word word (listof state) symbol → dfa-rule
;; Purpose: Generate dfa rule for given state and given word
;;           to match in the given pattern
(define (gen-state-tran s to-match patt states last-read)
```

- ```
;; Tests for gen-state-tran
(check-equal?
  (gen-state-tran
    'C '(a b b b) '(a b b a b c) '(S A B C D E F) 'b)
    '(C b S))
  (check-equal?
    (gen-state-tran
      'S '(b) '(a b b a b c) '(S A B C D E F) 'b)
      '(S b S)) ...)
```

# Deterministic Finite Automata

- `;; word word word (listof state) symbol → dfa-rule`  
`;; Purpose: Generate dfa rule for given state and given word`  
`;; to match in the given pattern`  
`(define (gen-state-tran s to-match patt states last-read)`
- `(cond [(empty? to-match) (list s last-read (first states))]`

- `;; Tests for gen-state-tran`  
`(check-equal?`  
`(gen-state-tran`  
`'C '(a b b b) '(a b b a b c) '(S A B C D E F) 'b)`  
`'(C b S))`  
`(check-equal?`  
`(gen-state-tran`  
`'S '(b) '(a b b a b c) '(S A B C D E F) 'b)`  
`'(S b S)) ...`

# Deterministic Finite Automata

- ```
;; word word word (listof state) symbol → dfa-rule
;; Purpose: Generate dfa rule for given state and given word
;;           to match in the given pattern
(define (gen-state-tran s to-match patt states last-read)
  (cond [(empty? to-match) (list s last-read (first states))]
        [(> (length to-match) (length patt))
         (list s last-read s)]))
```

- ```
;; Tests for gen-state-tran
(check-equal?
  (gen-state-tran
    'C '(a b b b) '(a b b a b c) '(S A B C D E F) 'b)
    '(C b S))
  (check-equal?
    (gen-state-tran
      'S '(b) '(a b b a b c) '(S A B C D E F) 'b)
      '(S b S)) ...))
```

# Deterministic Finite Automata

- `;; word word word (listof state) symbol → dfa-rule`  
`;; Purpose: Generate dfa rule for given state and given word`  
`;; to match in the given pattern`  
`(define (gen-state-tran s to-match patt states last-read)`
  - `(cond [(empty? to-match) (list s last-read (first states))]`
  - `[(> (length to-match) (length patt))`  
`(list s last-read s)]`
  - `[(equal? to-match (take patt (length to-match)))`  
`(list s`  
`(last to-match)`  
`(list-ref states (length to-match))))]`

- `;; Tests for gen-state-tran`  
`(check-equal?`  
`(gen-state-tran`  
`'C '(a b b b) '(a b b a b c) '(S A B C D E F) 'b)`  
`'(C b S))`  
`(check-equal?`  
`(gen-state-tran`  
`'S '(b) '(a b b a b c) '(S A B C D E F) 'b)`  
`'(S b S)) ...`

# Deterministic Finite Automata

- ```
;; word word word (listof state) symbol → dfa-rule
;; Purpose: Generate dfa rule for given state and given word
;;           to match in the given pattern
(define (gen-state-tran s to-match patt states last-read)
  (cond [(empty? to-match) (list s last-read (first states))]
        [(> (length to-match) (length patt))
         (list s last-read s)]
        [(equal? to-match (take patt (length to-match)))
         (list s
               (last to-match)
               (list-ref states (length to-match)))]
        [else (gen-state-tran s
                               (rest to-match)
                               patt
                               states
                               last-read)])))
```
- ```
;; Tests for gen-state-tran
(check-equal?
  (gen-state-tran
    'C '(a b b b) '(a b b a b c) '(S A B C D E F) 'b)
  '(C b S))
(check-equal?
  (gen-state-tran
    'S '(b) '(a b b a b c) '(S A B C D E F) 'b)
  '(S b S)) ...)
```

# Deterministic Finite Automata

Regular  
Expressions

Deterministic  
Finite  
Automata

Nondeterministic  
Finite  
Automata

Finite-State  
Automata and  
Regular  
Expressions

Regular  
Grammars

Pumping  
Theorem for  
Regular  
Languages

- This algorithm is the basis for the efficient and widely implemented Knuth-Morris-Pratt (KMP) algorithm
- The KMP algorithm is a string matching algorithm that looks for occurrences of a string in a block of text

# Deterministic Finite Automata

Regular  
Expressions

Deterministic  
Finite  
Automata

Nondeterministic  
Finite  
Automata

Finite-State  
Automata and  
Regular  
Expressions

Regular  
Grammars

Pumping  
Theorem for  
Regular  
Languages

- This algorithm is the basis for the efficient and widely implemented Knuth-Morris-Pratt (KMP) algorithm
- The KMP algorithm is a string matching algorithm that looks for occurrences of a string in a block of text
- Given that most programming languages do not have a `dfa` type like FSM, the KMP algorithm represents the `dfa` differently
- It uses a vector of indices into the pattern to represent where matching ought to continue when the next element in the text does not match the next element in the pattern
- You are strongly encouraged to review the KMP algorithm.

# Deterministic Finite Automata

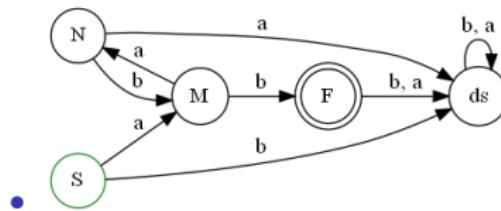
- HOMEWORK: 19
- Quiz: 20 (due in 1 week)

# Nondeterministic Finite Automata

- dfa can decide a language whose words are built using concatenation and Kleene star

# Nondeterministic Finite Automata

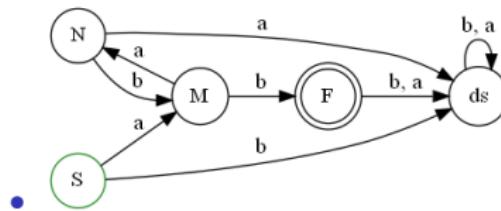
- dfa can decide a language whose words are built using concatenation and Kleene star



- A transition in a dfa represents concatenating an alphabet symbol
- A loop is concatenating a value generated by a Kleene star—the loop may be entered 0 or more times

# Nondeterministic Finite Automata

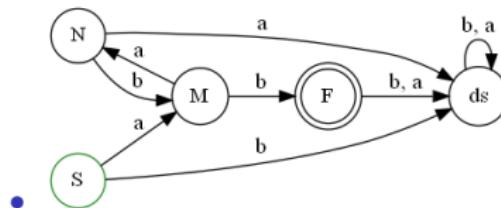
- dfa can decide a language whose words are built using concatenation and Kleene star



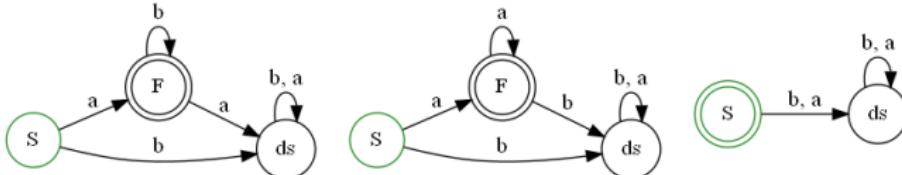
- A transition in a dfa represents concatenating an alphabet symbol
- A loop is concatenating a value generated by a Kleene star—the loop may be entered 0 or more times
- What about deciding a regular language that requires union?

# Nondeterministic Finite Automata

- dfa can decide a language whose words are built using concatenation and Kleene star



- A transition in a dfa represents concatenating an alphabet symbol
- A loop is concatenating a value generated by a Kleene star—the loop may be entered 0 or more times
- What about deciding a regular language that requires union?
- $L = ab^* \cup aa^* \cup \epsilon$
- There are three types of words that may be generated.
- It is not difficult to build a dfa for the language represented by each regular expression choice in the union:



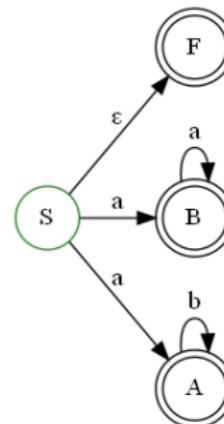
- It is difficult, however, to see how  $L$  can be decided by a dfa

# Nondeterministic Finite Automata

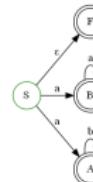
- A new model of a computer is needed
- One that allows a machine to change state in a manner that is not fully determined by the transition relation
- When the machine has a choice it nondeterministically chooses which transition (or transitions) to use

# Nondeterministic Finite Automata

- A new model of a computer is needed
- One that allows a machine to change state in a manner that is not fully determined by the transition relation
- When the machine has a choice it nondeterministically chooses which transition (or transitions) to use
- For instance, consider a finite-state machine for  $L$ :

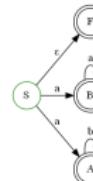


# Nondeterministic Finite Automata



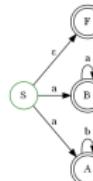
- Two new characteristics:
  - May change state without consuming anything
  - From a given state there may be more than one transition
- The transition relation is not a function
- Given the same input the machine may carry out different computations

# Nondeterministic Finite Automata



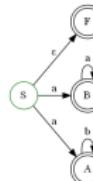
- Two new characteristics:
  - May change state without consuming anything
  - From a given state there may be more than one transition
- The transition relation is not a function
- Given the same input the machine may carry out different computations
- Processing ' $(a\ b\ b)$ ':
$$((a\ b\ b)\ S) \vdash ((a\ b\ b)\ F)$$
$$((a\ b\ b)\ S) \vdash ((b\ b)\ B)$$
$$((a\ b\ b)\ S) \vdash ((b\ b)\ A) \vdash ((b)\ A) \vdash (\emptyset\ A)$$
- The first two computations reject because input not consumed
- The third computation accepts ' $(a\ b\ b)$ '
- Based on this, is ' $(a\ b\ b)$ ' in  $L$  or not?

# Nondeterministic Finite Automata



- Two new characteristics:
  - May change state without consuming anything
  - From a given state there may be more than one transition
- The transition relation is not a function
- Given the same input the machine may carry out different computations
- Processing ' $(a\ b\ b)$ ':
$$((a\ b\ b)\ S) \vdash ((a\ b\ b)\ F)$$
$$((a\ b\ b)\ S) \vdash ((b\ b)\ B)$$
$$((a\ b\ b)\ S) \vdash ((b\ b)\ A) \vdash ((b)\ A) \vdash (\epsilon\ A)$$
- The first two computations reject because input not consumed
- The third computation accepts ' $(a\ b\ b)$ '
- Based on this, is ' $(a\ b\ b)$ ' in  $L$  or not?
- A word is in the language of a nondeterministic finite-state machine if there is at least one computation that leads to accept

# Nondeterministic Finite Automata



- Two new characteristics:
  - May change state without consuming anything
  - From a given state there may be more than one transition
- The transition relation is not a function
- Given the same input the machine may carry out different computations
- Processing ' $(a\ b\ b)$ ':
$$((a\ b\ b)\ S) \vdash ((a\ b\ b)\ F)$$
$$((a\ b\ b)\ S) \vdash ((b\ b)\ B)$$
$$((a\ b\ b)\ S) \vdash ((b\ b)\ A) \vdash ((b)\ A) \vdash (\ )\ A$$
- The first two computations reject because input not consumed
- The third computation accepts ' $(a\ b\ b)$ '
- Based on this, is ' $(a\ b\ b)$ ' in  $L$  or not?
- A word is in the language of a nondeterministic finite-state machine if there is at least one computation that leads to accept
- You may assume that if the input word is in the machine's language then every nondeterministic choice made during a computation is part of a computation that leads to the machine accepting the word
- Machines with such "intuition" sound very powerful

# Nondeterministic Finite Automata

## Definition

A nondeterministic finite-state automaton, ndfa, is a  $(\text{make-ndfa } K \Sigma S F \delta)$

- $\delta$  is a transition relation, not a function, that may have  $\epsilon$ -transitions and multiple transitions from a state on the same alphabet element

# Nondeterministic Finite Automata

## Definition

A nondeterministic finite-state automaton, **ndfa**, is a  $(\text{make-ndfa } K \Sigma S F \delta)$

- $\delta$  is a transition relation, not a function, that may have  $\epsilon$ -transitions and multiple transitions from a state on the same alphabet element
- An **ndfa** (transition) rule is defined as follows:

$(Q \xrightarrow{a} R)$ , where  $Q, R \in K \wedge a \in \{\Sigma \cup \{\epsilon\}\}$

# Nondeterministic Finite Automata

## Definition

A nondeterministic finite-state automaton,  $\text{ndfa}$ , is a  $(\text{make-ndfa } K \Sigma S F \delta)$

- $\delta$  is a transition relation, not a function, that may have  $\epsilon$ -transitions and multiple transitions from a state on the same alphabet element
- An  $\text{ndfa}$  (transition) rule is defined as follows:  
$$(Q \xrightarrow{a} R), \text{ where } Q, R \in K \wedge a \in \{\Sigma \cup \{\epsilon\}\}$$
- Given a configuration  $(m \ r)$ , where  $m \in \Sigma^*$  and  $r \in K$ , the machine may have no transitions it may follow and halts or may move to one of several different configurations because of  $\epsilon$ -transitions or multiple transitions from  $r$  on the same alphabet element

# Nondeterministic Finite Automata

## Definition

A nondeterministic finite-state automaton,  $\text{ndfa}$ , is a  $(\text{make-ndfa } K \Sigma S F \delta)$

- $\delta$  is a transition relation, not a function, that may have  $\epsilon$ -transitions and multiple transitions from a state on the same alphabet element
- An  $\text{ndfa}$  (transition) rule is defined as follows:  
$$(Q \xrightarrow{a} R), \text{ where } Q, R \in K \wedge a \in \{\Sigma \cup \{\epsilon\}\}$$
- Given a configuration  $(m \ r)$ , where  $m \in \Sigma^*$  and  $r \in K$ , the machine may have no transitions it may follow and halts or may move to one of several different configurations because of  $\epsilon$ -transitions or multiple transitions from  $r$  on the same alphabet element
- A word,  $w$ , is accepted by an  $\text{ndfa}$ ,  $N$ , if there exists a computation such that:  
$$(w \ s) \vdash^* ((\cdot) \ f), \text{ where } f \in F$$
- The language of  $N$ ,  $L(N)$ , is all the words accepted by  $N$ .

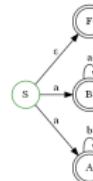
# Nondeterministic Finite Automata

## Definition

A nondeterministic finite-state automaton,  $\text{ndfa}$ , is a  $(\text{make-ndfa } K \Sigma S F \delta)$

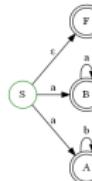
- $\delta$  is a transition relation, not a function, that may have  $\epsilon$ -transitions and multiple transitions from a state on the same alphabet element
- An  $\text{ndfa}$  (transition) rule is defined as follows:  
$$(Q \xrightarrow{a} R), \text{ where } Q, R \in K \wedge a \in \{\Sigma \cup \{\epsilon\}\}$$
- Given a configuration  $(m \ r)$ , where  $m \in \Sigma^*$  and  $r \in K$ , the machine may have no transitions it may follow and halts or may move to one of several different configurations because of  $\epsilon$ -transitions or multiple transitions from  $r$  on the same alphabet element
- A word,  $w$ , is accepted by an  $\text{ndfa}$ ,  $N$ , if there exists a computation such that:  
$$(w \ s) \vdash^* ((\cdot) \ f), \text{ where } f \in F$$
- The language of  $N$ ,  $L(N)$ , is all the words accepted by  $N$ .
- Every  $\text{dfa}$  is an  $\text{ndfa}$

# Nondeterministic Finite Automata



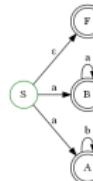
- #lang fsm  
;; L = { $\epsilon$ }  $\cup$  aa\*  $\cup$  ab\*  
(define LNDFA (make-ndfa '(S A B F)

# Nondeterministic Finite Automata



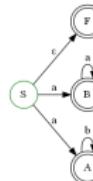
- ```
#lang fsm
;; L = {ε} ∪ aa* ∪ ab*
(define LNDFA (make-ndfa '(S A B F)
```
- ```
;; Tests for LNDFA
(check-equal? (sm-apply LNDFA '(a b a)) 'reject)
(check-equal? (sm-apply LNDFA '(b b b b b)) 'reject)
(check-equal? (sm-apply LNDFA '(a b b b b a a a)) 'reject)
(check-equal? (sm-apply LNDFA '()) 'accept)
(check-equal? (sm-apply LNDFA '(a)) 'accept)
(check-equal? (sm-apply LNDFA '(a a a a)) 'accept)
(check-equal? (sm-apply LNDFA '(a b b)) 'accept)
```

# Nondeterministic Finite Automata



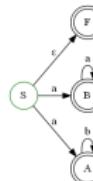
- ```
#lang fsm
;; L = {ε} ∪ aa* ∪ ab*
(define LNDFA (make-ndfa '(S A B F)
                           '(a b))
```
- ```
;; Tests for LNDFA
(check-equal? (sm-apply LNDFA '(a b a)) 'reject)
(check-equal? (sm-apply LNDFA '(b b b b b)) 'reject)
(check-equal? (sm-apply LNDFA '(a b b b b a a a)) 'reject)
(check-equal? (sm-apply LNDFA '()) 'accept)
(check-equal? (sm-apply LNDFA '(a)) 'accept)
(check-equal? (sm-apply LNDFA '(a a a a)) 'accept)
(check-equal? (sm-apply LNDFA '(a b b)) 'accept)
```

# Nondeterministic Finite Automata



- ```
#lang fsm
;; L = {ε} ∪ aa* ∪ ab*
(define LNDFA (make-ndfa '(S A B F)
                           '(a b)
                           'S))
```
- ```
;; Tests for LNDFA
(check-equal? (sm-apply LNDFA '(a b a)) 'reject)
(check-equal? (sm-apply LNDFA '(b b b b b)) 'reject)
(check-equal? (sm-apply LNDFA '(a b b b b a a a)) 'reject)
(check-equal? (sm-apply LNDFA '()) 'accept)
(check-equal? (sm-apply LNDFA '(a)) 'accept)
(check-equal? (sm-apply LNDFA '(a a a a)) 'accept)
(check-equal? (sm-apply LNDFA '(a b b)) 'accept)
```

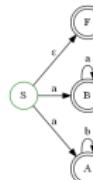
# Nondeterministic Finite Automata



- ```
#lang fsm
;; L = {ε} ∪ aa* ∪ ab*
(define LNDFA (make-ndfa '(S A B F)
                            '(a b)
                            'S
                            '(A B F))
```
- 
- 
- 
- 
- 
- ;; Tests for LNDFA

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

# Nondeterministic Finite Automata



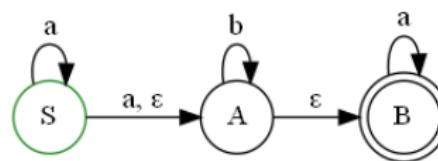
- ```
#lang fsm
;; L = {ε} ∪ aa* ∪ ab*
(define LNDFA (make-ndfa '(S A B F)
                           '(a b)
                           'S
                           '(A B F)
                           `((S a A) (S a B) (S ,EMP F)
                             (A b A)
                             (B a B))))
```
- ```
;; Tests for LNDFA
(check-equal? (sm-apply LNDFA '(a b a)) 'reject)
(check-equal? (sm-apply LNDFA '(b b b b b)) 'reject)
(check-equal? (sm-apply LNDFA '(a b b b b a a a)) 'reject)
(check-equal? (sm-apply LNDFA '()) 'accept)
(check-equal? (sm-apply LNDFA '(a)) 'accept)
(check-equal? (sm-apply LNDFA '(a a a a)) 'accept)
(check-equal? (sm-apply LNDFA '(a b b)) 'accept)
```
- The only transitions listed are those that are on a path to an accepting state

# Nondeterministic Finite Automata

- Designing an ndfa can prove easier than designing a dfa

# Nondeterministic Finite Automata

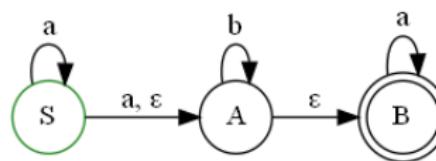
- Designing an ndfa can prove easier than designing a dfa
- Care must be taken when reasoning about the machine:



- What state does the machine move to if it is in S and consumes an a?

# Nondeterministic Finite Automata

- Designing an ndfa can prove easier than designing a dfa
- Care must be taken when reasoning about the machine:



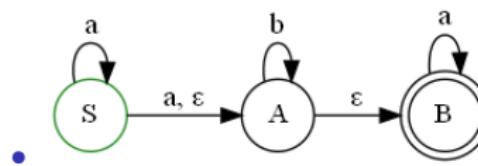
- What state does the machine move to if it is in S and consumes an a?
- The machine may end in S, A, or B
- State invariant must hold for any state in a computation that leads to accept

# Nondeterministic Finite Automata

- To aid us in reasoning about ndfas we define the *empties* of a state R:  
$$E(R) = \{\{R\} \cup \{P \mid ((\epsilon R) \vdash^* (\epsilon P))\}\}$$
- That is,  $E(R)$  contains R and all states reachable from R by only following  $\epsilon$ -transitions

# Nondeterministic Finite Automata

- To aid us in reasoning about ndfas we define the *empties* of a state R:  
$$E(R) = \{\{R\} \cup \{P \mid ((\epsilon R) \vdash^* (\epsilon P))\}\}$$
- That is,  $E(R)$  contains R and all states reachable from R by only following  $\epsilon$ -transitions



$$E(S) = \{S \ A \ B\}$$

$$E(A) = \{A \ B\}$$

$$E(B) = \{B\}$$

# Nondeterministic Finite Automata

- $L = \{w \mid a \notin w \vee b \notin w \vee c \notin w\}$

# Nondeterministic Finite Automata

- $L = \{w \mid a \notin w \vee b \notin w \vee c \notin w\}$
- Name: AT-LEAST-ONE-MISSING
- $\Sigma = \{a \ b \ c\}$

# Nondeterministic Finite Automata

- $L = \{w \mid a \notin w \vee b \notin w \vee c \notin w\}$

- Name: AT-LEAST-ONE-MISSING

- $\Sigma = \{a \ b \ c\}$

- ;; Tests for AT-LEAST-ONE-MISSING

```
(check-equal? (sm-apply AT-LEAST-ONE-MISSING '(a b c)) 'reject)
(check-equal? (sm-apply AT-LEAST-ONE-MISSING '(b b a b c b a))
               'reject)
(check-equal? (sm-apply AT-LEAST-ONE-MISSING '(b a c)) 'reject)
(check-equal? (sm-apply AT-LEAST-ONE-MISSING '()) 'accept)
(check-equal? (sm-apply AT-LEAST-ONE-MISSING '(a)) 'accept)
(check-equal? (sm-apply AT-LEAST-ONE-MISSING '(b)) 'accept)
(check-equal? (sm-apply AT-LEAST-ONE-MISSING '(c)) 'accept)
(check-equal? (sm-apply AT-LEAST-ONE-MISSING '(c c a a))
               'accept)
(check-equal? (sm-apply AT-LEAST-ONE-MISSING '(b b c b b b))
               'accept)
(check-equal? (sm-apply AT-LEAST-ONE-MISSING '(a a a b b b))
               'accept)
```

# Nondeterministic Finite Automata

- Design Idea and conditions
- Nondeterministically decides to process the word as if a is missing, as if b is missing, or as if c is missing

# Nondeterministic Finite Automata

- Design Idea and conditions
- Nondeterministically decides to process the word as if a is missing, as if b is missing, or as if c is missing
- The consumed input,  $c_i$ , must satisfy one of four conditions: nothing is consumed or for each  $x \in (\text{sm}-\sigma)$  AT-LEAST-ONE-MISSING)  $x \notin c_i$

# Nondeterministic Finite Automata

- Design Idea and conditions
- Nondeterministically decides to process the word as if a is missing, as if b is missing, or as if c is missing
- The consumed input,  $c_i$ , must satisfy one of four conditions: nothing is consumed or for each  $x \in (\text{sm-sigma AT-LEAST-ONE-MISSING})$   $x \notin c_i$
- Four states are needed:

```
;; States
;; S: the consumed input is empty, starting state
;; A: the consumed input does not contain a, final state
;; B: the consumed input does not contain b, final state
;; C: the consumed input does not contain c, final state
```

# Nondeterministic Finite Automata

- Transition Relation
- In S nondeterministically move to either A, B, or C
- The transition relation is:

$\sim((S, \text{EMP } A)$   
 $(S, \text{EMP } B)$   
 $(S, \text{EMP } C)$

# Nondeterministic Finite Automata

- Transition Relation
- In S nondeterministically move to either A, B, or C
- In A process arbitrary number of bs or cs

- The transition relation is:

$\sim((S, \text{EMP } A)$   
 $(S, \text{EMP } B)$   
 $(S, \text{EMP } C))$

- $(A \xrightarrow{b} A)$   
 $(A \xrightarrow{c} A)$

# Nondeterministic Finite Automata

- Transition Relation
- In S nondeterministically move to either A, B, or C
- In A process arbitrary number of bs or cs
- In B processes arbitrary number of as or cs
- The transition relation is:
  - $\sim((S, \text{EMP } A)$   
 $(S, \text{EMP } B)$   
 $(S, \text{EMP } C)$
  - $(A \xrightarrow{b} A)$   
 $(A \xrightarrow{c} A)$
  - $(B \xrightarrow{a} B)$   
 $(B \xrightarrow{c} B)$

# Nondeterministic Finite Automata

- Transition Relation
- In S nondeterministically move to either A, B, or C
- In A process arbitrary number of bs or cs
- In B processes arbitrary number of as or cs
- In C process arbitrary number of as or bs.
- The transition relation is:

$\sim((S, \text{EMP } A)$   
 $(S, \text{EMP } B)$   
 $(S, \text{EMP } C))$

- $(A \xrightarrow{b} A)$   
 $(A \xrightarrow{c} A)$
- $(B \xrightarrow{a} B)$   
 $(B \xrightarrow{c} B)$
- $(C \xrightarrow{a} C)$   
 $(C \xrightarrow{b} C))$

# Nondeterministic Finite Automata

- ```
;; States
;; S: the consumed input is empty, starting state
;; A: the consumed input does not contain a, final state
;; B: the consumed input does not contain b, final state
;; C: the consumed input does not contain c, final state
```
- ```
;; word → Boolean
;; Purpose: Determine if the given word is empty
(define (S-INV ci) (empty? ci))

;; Test for S-INV
(check-equal? (S-INV '()) #t)
(check-equal? (S-INV '(a b)) #f)
```

# Nondeterministic Finite Automata

- ```
;; States
;; S: the consumed input is empty, starting state
;; A: the consumed input does not contain a, final state
;; B: the consumed input does not contain b, final state
;; C: the consumed input does not contain c, final state
```
- ```
;; word → Boolean
;; Purpose: Determine if the given word is empty
(define (S-INV ci) (empty? ci))

;; Test for S-INV
(check-equal? (S-INV '()) #t)
(check-equal? (S-INV '(a b)) #f)
```
- ```
;; word → Boolean
;; Purpose: Determine if the given word does not contain a
(define (A-INV ci) (empty? (filter (λ (a) (eq? a 'a)) ci)))

;; Test for A-INV
(check-equal? (A-INV '(a)) #f)
(check-equal? (A-INV '(a c b)) #f)
(check-equal? (A-INV '(c c b a b)) #f)
(check-equal? (A-INV '(b)) #t)
(check-equal? (A-INV '(c c b c b)) #t)
(check-equal? (A-INV '()) #t)
```

# Nondeterministic Finite Automata

- ```
;; States
;; S: the consumed input is empty, starting state
;; A: the consumed input does not contain a, final state
;; B: the consumed input does not contain b, final state
;; C: the consumed input does not contain c, final state
```
- ```
;; word → Boolean
;; Purpose: Determine if the given word does not contain b
(define (B-INV ci) (empty? (filter (λ (a) (eq? a 'b)) ci)))
```
- ```
;; Test for B-INV
(check-equal? (B-INV '(b)) #f)
(check-equal? (B-INV '(a c b)) #f)
(check-equal? (B-INV '(a a b a b)) #f)
(check-equal? (B-INV '(c)) #t)
(check-equal? (B-INV '(c c a c c a a a)) #t)
(check-equal? (B-INV '()) #t)
```

# Nondeterministic Finite Automata

- ```
;; States
;; S: the consumed input is empty, starting state
;; A: the consumed input does not contain a, final state
;; B: the consumed input does not contain b, final state
;; C: the consumed input does not contain c, final state
```
- ```
;; word → Boolean
;; Purpose: Determine if the given word does not contain b
(define (B-INV ci) (empty? (filter (λ (a) (eq? a 'b)) ci)))
```
- ```
;; Test for B-INV
(check-equal? (B-INV '(b)) #f)
(check-equal? (B-INV '(a c b)) #f)
(check-equal? (B-INV '(a a b a b)) #f)
(check-equal? (B-INV '(c)) #t)
(check-equal? (B-INV '(c c a c c a a a)) #t)
(check-equal? (B-INV '()) #t)
```
- ```
;; word → Boolean
;; Purpose: Determine if the given word does not contain c
(define (C-INV ci) (empty? (filter (λ (a) (eq? a 'c)) ci)))
```

```
;; Test for C-INV
(check-equal? (C-INV '(c)) #f)
(check-equal? (C-INV '(a b c b)) #f)
(check-equal? (C-INV '(c c b a b)) #f)
(check-equal? (C-INV '(b)) #t)
```

# Nondeterministic Finite Automata

- Validate AT-LEAST-ONE-MISSING's:

```
(sm-visualize AT-LEAST-ONE-MISSING
  (list 'S S-INV)
  (list 'A A-INV)
  (list 'B B-INV)
  (list 'C C-INV))
```

# Nondeterministic Finite Automata

- We must show that  $R$ 's invariant implies the invariant for every state in  $E(R)$  that can lead to accept

# Nondeterministic Finite Automata

- We must show that  $R$ 's invariant implies the invariant for every state in  $E(R)$  that can lead to accept
- Why are we only concerned with nondeterministic transitions that may lead to an accept?

# Nondeterministic Finite Automata

- We must show that  $R$ 's invariant implies the invariant for every state in  $E(R)$  that can lead to accept
- Why are we only concerned with nondeterministic transitions that may lead to an accept?
- An ndfa never makes a nondeterministic transition unless it furthers a computation that leads to an accept
- Nondeterministic choices not made do no concern us

# Nondeterministic Finite Automata

- We must show that  $R$ 's invariant implies the invariant for every state in  $E(R)$  that can lead to accept
- Why are we only concerned with nondeterministic transitions that may lead to an accept?
- An ndfa never makes a nondeterministic transition unless it furthers a computation that leads to an accept
- Nondeterministic choices not made do no concern us
- We only need to reason about nondeterministic transitions that can lead to accept
- Nondeterministic transitions that cannot lead to an accept are never made and, thus, not part of any computation

# Nondeterministic Finite Automata

- Assume:
  - $M = \text{AT-LEAST-ONE-MISSING}$
  - $w \in (\Sigma - \sigma)^*$
  - $F = (\Sigma - \text{finals } M)$
  - $c_i$  is the consumed input

# Nondeterministic Finite Automata

## Theorem

*The state invariants hold when M is applied to w.*

- BASE CASE
- When M starts, S-INV holds because  $c_i = '()$

# Nondeterministic Finite Automata

## Theorem

*The state invariants hold when M is applied to w.*

- BASE CASE
- When M starts, S-INV holds because  $c_i = '()$
- If  $w \in L(M)$  then M nondeterministically moves to A, B, or C using an  $\epsilon$ -transition.

# Nondeterministic Finite Automata

## Theorem

*The state invariants hold when M is applied to w.*

- BASE CASE
- When M starts, S-INV holds because  $c_i = '()$
- If  $w \in L(M)$  then M nondeterministically moves to A, B, or C using an  $\epsilon$ -transition.
- Given that S-INV holds, we know that  $c_i = '()$

# Nondeterministic Finite Automata

## Theorem

*The state invariants hold when M is applied to w.*

- BASE CASE
- When M starts, S-INV holds because  $c_i = '()$
- If  $w \in L(M)$  then M nondeterministically moves to A, B, or C using an  $\epsilon$ -transition.
- Given that S-INV holds, we know that  $c_i = '()$
- $(S \in A)$ ,  $(S \in B)$ , and  $(S \in C)$  add nothing to the input and  $c_i = \text{elist}$  after using any of them

# Nondeterministic Finite Automata

## Theorem

*The state invariants hold when M is applied to w.*

- BASE CASE
- When M starts, S-INV holds because  $c_i = '()$
- If  $w \in L(M)$  then M nondeterministically moves to A, B, or C using an  $\epsilon$ -transition.
- Given that S-INV holds, we know that  $c_i = '()$
- $(S \in A)$ ,  $(S \in B)$ , and  $(S \in C)$  add nothing to the input and  $c_i = \text{elist}$  after using any of them
- A-INV holds because  $c_i$  contains zero as

# Nondeterministic Finite Automata

## Theorem

*The state invariants hold when M is applied to w.*

- BASE CASE
- When M starts, S-INV holds because  $c_i = '()$
- If  $w \in L(M)$  then M nondeterministically moves to A, B, or C using an  $\epsilon$ -transition.
- Given that S-INV holds, we know that  $c_i = '()$
- $(S \in A)$ ,  $(S \in B)$ , and  $(S \in C)$  add nothing to the input and  $c_i = \epsilon$  list after using any of them
- A-INV holds because  $c_i$  contains zero as
- B-INV holds because  $c_i$  contains zero bs

# Nondeterministic Finite Automata

## Theorem

*The state invariants hold when M is applied to w.*

- BASE CASE
- When M starts, S-INV holds because  $c_i = '()$
- If  $w \in L(M)$  then M nondeterministically moves to A, B, or C using an  $\epsilon$ -transition.
- Given that S-INV holds, we know that  $c_i = '()$
- $(S \in A)$ ,  $(S \in B)$ , and  $(S \in C)$  add nothing to the input and  $c_i = \epsilon$  list after using any of them
- A-INV holds because  $c_i$  contains zero as
- B-INV holds because  $c_i$  contains zero bs
- C-INV holds because  $c_i$  contains zero cs.

# Nondeterministic Finite Automata

## Theorem

*The state invariants hold when  $M$  is applied to  $w$ .*

- INDUCTIVE STEP
- (A b A): By inductive hypothesis A-INV holds. A-INV guarantees that the consumed input does not contain an a. Consuming a b means that the consumed input remains without an a. Therefore, A-INV holds after the transition.

# Nondeterministic Finite Automata

## Theorem

*The state invariants hold when M is applied to w.*

- INDUCTIVE STEP
- (A b A): By inductive hypothesis A-INV holds. A-INV guarantees that the consumed input does not contain an a. Consuming a b means that the consumed input remains without an a. Therefore, A-INV holds after the transition.
- (A c A): By inductive hypothesis A-INV holds. A-INV guarantees that the consumed input does not contain an a. Consuming a c means that the consumed input remains without an a. Therefore, A-INV holds after the transition.

# Nondeterministic Finite Automata

## Theorem

*The state invariants hold when M is applied to w.*

- INDUCTIVE STEP
- (A b A): By inductive hypothesis A-INV holds. A-INV guarantees that the consumed input does not contain an a. Consuming a b means that the consumed input remains without an a. Therefore, A-INV holds after the transition.
- (A c A): By inductive hypothesis A-INV holds. A-INV guarantees that the consumed input does not contain an a. Consuming a c means that the consumed input remains without an a. Therefore, A-INV holds after the transition.
- (B a B): By inductive hypothesis B-INV holds. B-INV guarantees that the consumed input does not contain an b. Consuming an a means that the consumed input remains without an b. Therefore, B-INV holds after the transition.

# Nondeterministic Finite Automata

## Theorem

*The state invariants hold when  $M$  is applied to  $w$ .*

- INDUCTIVE STEP
- (A b A): By inductive hypothesis A-INV holds. A-INV guarantees that the consumed input does not contain an a. Consuming a b means that the consumed input remains without an a. Therefore, A-INV holds after the transition.
- (A c A): By inductive hypothesis A-INV holds. A-INV guarantees that the consumed input does not contain an a. Consuming a c means that the consumed input remains without an a. Therefore, A-INV holds after the transition.
- (B a B): By inductive hypothesis B-INV holds. B-INV guarantees that the consumed input does not contain an b. Consuming an a means that the consumed input remains without an b. Therefore, B-INV holds after the transition.
- (B c B): By inductive hypothesis B-INV holds. B-INV guarantees that the consumed input does not contain an b. Consuming a c means that the consumed input remains without a b. Therefore, B-INV holds after the transition.

# Nondeterministic Finite Automata

## Theorem

*The state invariants hold when  $M$  is applied to  $w$ .*

- INDUCTIVE STEP
- (C a C): By inductive hypothesis C-INV holds. C-INV guarantees that the consumed input does not contain a c. Consuming an a means that the consumed input remains without a c. Therefore, C-INV holds after the transition.

# Nondeterministic Finite Automata

## Theorem

*The state invariants hold when  $M$  is applied to  $w$ .*

- INDUCTIVE STEP
- (C a C): By inductive hypothesis C-INV holds. C-INV guarantees that the consumed input does not contain a c. Consuming an a means that the consumed input remains without a c. Therefore, C-INV holds after the transition.
- (C b C): By inductive hypothesis C-INV holds. C-INV guarantees that the consumed input does not contain a c. Consuming an b means that the consumed input remains without a c. Therefore, C-INV holds after the transition.

# Nondeterministic Finite Automata

## Lemma

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

- ( $\Rightarrow$ ) Assume  $w \in L$ .
- $w \in L$  means that  $w$  is missing a, b, or c. Given that the state invariants always hold, this means M consumes all its input in either A, B, or C. Given that  $A, B, C \in F$ ,  $w \in L(M)$ .

# Nondeterministic Finite Automata

## Lemma

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

- ( $\Rightarrow$ ) Assume  $w \in L$ .
- $w \in L$  means that  $w$  is missing a, b, or c. Given that the state invariants always hold, this means  $M$  consumes all its input in either A, B, or C. Given that  $A, B, C \in F$ ,  $w \in L(M)$ .
- ( $\Leftarrow$ ) Assume  $w \in L(M)$ .
- $w \in L(M)$  means that  $M$  consumes all its input and halts in A, B, or C. Given that the state invariants always hold we may conclude that  $w$  is missing a, b, or c. Therefore,  $w \in L$ .

# Nondeterministic Finite Automata

## Lemma

$$w \notin L \Leftrightarrow w \notin L(M)$$

- ( $\Rightarrow$ ) Assume  $w \notin L$ .
- $w \notin L$  means that  $w$  has at least one  $a$ , one  $b$ , and one  $c$ . This means  $M$  does not move out of  $S$  because there is no nondeterministic transition possible that lead to accept. Given that  $M$  halts without consuming all of its input,  $w$  is rejected and, therefore,  $w \notin L(M)$ .

# Nondeterministic Finite Automata

## Lemma

$$w \notin L \Leftrightarrow w \notin L(M)$$

- ( $\Rightarrow$ ) Assume  $w \notin L$ .
- $w \notin L$  means that  $w$  has at least one  $a$ , one  $b$ , and one  $c$ . This means  $M$  does not move out of  $S$  because there is no nondeterministic transition possible that lead to accept. Given that  $M$  halts without consuming all of its input,  $w$  is rejected and, therefore,  $w \notin L(M)$ .
- ( $\Leftarrow$ ) Assume  $w \notin L(M)$ .
- $w \notin L(M)$  means that  $M$  halts without consuming all of  $w$ . Given that the state invariants always hold,  $w$  must have at least one  $a$ , one  $b$ , and one  $c$ . Thus,  $w \notin L$ .

# Nondeterministic Finite Automata

## Theorem

$$L = L(AT-LEAST-ONE-MISSING)$$

- The previous two lemmas establish the theorem.

# Nondeterministic Finite Automata

- HOMEWORK: 2–5
- BONUS QUIZ: 1 (due in 1 week)

# Nondeterministic Finite Automata

- Does endowing a finite-state machine with nondeterminism give us more computational power?

# Nondeterministic Finite Automata

- Does endowing a finite-state machine with nondeterminism give us more computational power?
- To simulate an `ndfa` a `dfa` needs to simulate all computations of the `ndfa` simultaneously
- At first glance this may sound like preposterous
- Do we need a `dfa` to be in multiple states at the same time?

# Nondeterministic Finite Automata

- Does endowing a finite-state machine with nondeterminism give us more computational power?
- To simulate an ndfa a dfa needs to simulate all computations of the ndfa simultaneously
- At first glance this may sound like preposterous
- Do we need a dfa to be in multiple states at the same time?
- Consider an ndfa,  $N = (\text{make-ndfa } S \ \Sigma s \ F \ \delta)$ , making a transition:  
 $(P \ a \ R_1)$   
⋮  
 $(P \ a \ R_n)$
- After consuming a, what states can N be in?

# Nondeterministic Finite Automata

- Does endowing a finite-state machine with nondeterminism give us more computational power?
- To simulate an ndfa a dfa needs to simulate all computations of the ndfa simultaneously
- At first glance this may sound like preposterous
- Do we need a dfa to be in multiple states at the same time?
- Consider an ndfa,  $N = (\text{make-ndfa } S \ \Sigma s \ F \ \delta)$ , making a transition:  
 $(P \ a \ R_1)$   
 $\vdots$   
 $(P \ a \ R_n)$
- After consuming a, what states can N be in?  
 $E(R_1) \cup E(R_2) \cup \dots \cup E(R_n)$

# Nondeterministic Finite Automata

- Does endowing a finite-state machine with nondeterminism give us more computational power?
- To simulate an ndfa a dfa needs to simulate all computations of the ndfa simultaneously
- At first glance this may sound like preposterous
- Do we need a dfa to be in multiple states at the same time?
- Consider an ndfa,  $N = (\text{make-ndfa } S \ \Sigma s \ F \ \delta)$ , making a transition:

$(P \ a \ R_1)$

$\vdots$

$(P \ a \ R_n)$

- After consuming a, what states can N be in?
- $E(R_1) \cup E(R_2) \cup \dots \cup E(R_n)$
- Observe that the above is an element in  $2^S$
- Call it Z

# Nondeterministic Finite Automata

- Does endowing a finite-state machine with nondeterminism give us more computational power?
- To simulate an ndfa a dfa needs to simulate all computations of the ndfa simultaneously

- At first glance this may sound like preposterous
- Do we need a dfa to be in multiple states at the same time?

- Consider an ndfa,  $N = (\text{make-ndfa } S \ \Sigma s \ F \ \delta)$ , making a transition:

$(P \ a \ R_1)$

⋮

$(P \ a \ R_n)$

- After consuming  $a$ , what states can  $N$  be in?
$$E(R_1) \cup E(R_2) \cup \dots \cup E(R_n)$$
- Observe that the above is an element in  $2^S$
- Call it  $Z$
- Think of  $Z$  as a *super state* for a dfa that represents all the states  $N$  may be in
- To simulate all possible computations that may be performed by  $N$  a dfa transitions between super states
- After consuming all the input, it accepts if it is in a super state that contains a final state in  $N$ . Otherwise, it rejects.

# Nondeterministic Finite Automata

- It is necessary to show how such a dfa, M, is constructed and to show that  $L(M) = L(N)$
- Showing how to build something (like a dfa) and showing that the construction is correct (like  $L(M) = L(N)$ ) is called a *constructive proof*
- A constructive proof has a construction algorithm and a proof of its correctness.

# Nondeterministic Finite Automata

- Let  $N = (\text{make-ndfa } S \ \Sigma s \ F \ \delta)$
- Building a dfa from  $N$  hinges on computing a transition function between super states for a dfa
- The dfa is constructed as follows:

```
(make-dfa <encoding of super states>
          Σ
          <encoding of E(s)>
          <encoding of super states that contain f ∈ F>
          <transition function between encoded super states>)
```

# Nondeterministic Finite Automata

- Let  $N = (\text{make-ndfa } S \Sigma s F \delta)$
- Building a dfa from  $N$  hinges on computing a transition function between super states for a dfa
- The dfa is constructed as follows:

```
(make-dfa <encoding of super states>
          Σ
          <encoding of E(s)>
          <encoding of super states that contain f ∈ F>
          <transition function between encoded super states>)
```

- To compute the transition function process,  $P$ , each known super state
- At the beginning, the only known super state is  $E(s)$

# Nondeterministic Finite Automata

- Let  $N = (\text{make-ndfa } S \Sigma s F \delta)$
- Building a dfa from  $N$  hinges on computing a transition function between super states for a dfa
- The dfa is constructed as follows:

```
(make-dfa <encoding of super states>
          Σ
          <encoding of E(s)>
          <encoding of super states that contain f ∈ F>
          <transition function between encoded super states>)
```

- To compute the transition function process,  $P$ , each known super state
- At the beginning, the only known super state is  $E(s)$
- For each state,  $p ∈ P$ , each alphabet element,  $a$ , is processed to compute, possibly new, super states
- The union of the super states obtained from processing  $a$  for each  $p ∈ P$  is the super state the dfa moves to from  $P$  on an  $a$

# Nondeterministic Finite Automata

- Let  $N = (\text{make-ndfa } S \Sigma s F \delta)$
- Building a dfa from  $N$  hinges on computing a transition function between super states for a dfa
- The dfa is constructed as follows:

```
(make-dfa <encoding of super states>
          Σ
          <encoding of E(s)>
          <encoding of super states that contain f ∈ F>
          <transition function between encoded super states>)
```

- To compute the transition function process,  $P$ , each known super state
- At the beginning, the only known super state is  $E(s)$
- For each state,  $p ∈ P$ , each alphabet element,  $a$ , is processed to compute, possibly new, super states
- The union of the super states obtained from processing  $a$  for each  $p ∈ P$  is the super state the dfa moves to from  $P$  on an  $a$
- Let  $P = \{p_1 p_2 p_3\}$
- Let  $(p_1 a r), (p_1 a s), (p_3 a t) ∈ \delta$

# Nondeterministic Finite Automata

- Let  $N = (\text{make-ndfa } S \ \Sigma s \ F \ \delta)$
- Building a dfa from  $N$  hinges on computing a transition function between super states for a dfa
- The dfa is constructed as follows:

```
(make-dfa <encoding of super states>
          Σ
          <encoding of E(s)>
          <encoding of super states that contain f ∈ F>
          <transition function between encoded super states>)
```

- To compute the transition function process,  $P$ , each known super state
- At the beginning, the only known super state is  $E(s)$
- For each state,  $p ∈ P$ , each alphabet element,  $a$ , is processed to compute, possibly new, super states
- The union of the super states obtained from processing  $a$  for each  $p ∈ P$  is the super state the dfa moves to from  $P$  on an  $a$
- Let  $P = \{p_1 \ p_2 \ p_3\}$
- Let  $(p_1 \ a \ r), (p_1 \ a \ s), (p_3 \ a \ t) ∈ δ$
- On an  $a$ , from  $p_1$   $N$  may transition to any state in  $E(r) ∪ E(s)$
- From  $p_2$   $N$  may transition nowhere
- From  $p_3$   $N$  may transition to any state in  $E(t)$

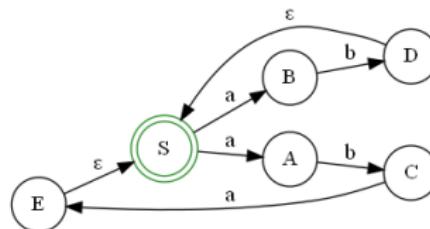
# Nondeterministic Finite Automata

- Let  $N = (\text{make-ndfa } S \Sigma s F \delta)$
- Building a dfa from  $N$  hinges on computing a transition function between super states for a dfa
- The dfa is constructed as follows:

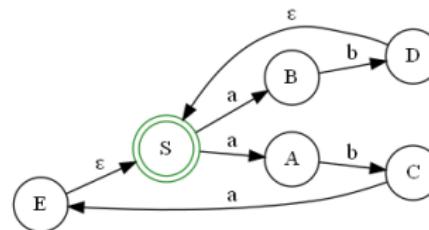
```
(make-dfa <encoding of super states>
          Σ
          <encoding of E(s)>
          <encoding of super states that contain f ∈ F>
          <transition function between encoded super states>)
```

- To compute the transition function process,  $P$ , each known super state
- At the beginning, the only known super state is  $E(s)$
- For each state,  $p ∈ P$ , each alphabet element,  $a$ , is processed to compute, possibly new, super states
- The union of the super states obtained from processing  $a$  for each  $p ∈ P$  is the super state the dfa moves to from  $P$  on an  $a$
- Let  $P = \{p_1 p_2 p_3\}$
- Let  $(p_1 a r), (p_1 a s), (p_3 a t) ∈ δ$
- On an  $a$ , from  $p_1$   $N$  may transition to any state in  $E(r) ∪ E(s)$
- From  $p_2$   $N$  may transition nowhere
- From  $p_3$   $N$  may transition to any state in  $E(t)$
- We may describe a transition in the dfa as follows:  
 $((p_1 p_2 p_3) a (E(r) ∪ E(s) ∪ E(t)))$
- That is, the dfa transitions from super state  $P$  on an  $a$  to a super state  $Q$ , where  $Q = (E(r) ∪ E(s) ∪ E(t))$

# Nondeterministic Finite Automata

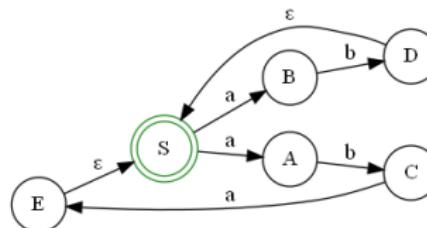


# Nondeterministic Finite Automata



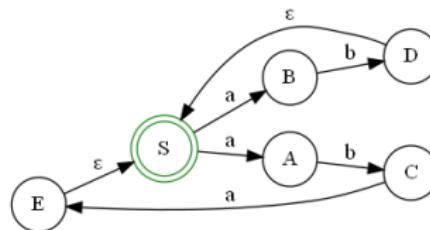
- | state | $E(state)$ |
|-------|------------|
| S     | (S)        |
| A     | (A)        |
| B     | (B)        |
| C     | (C)        |
| D     | (D S)      |
| E     | (E S)      |

# Nondeterministic Finite Automata



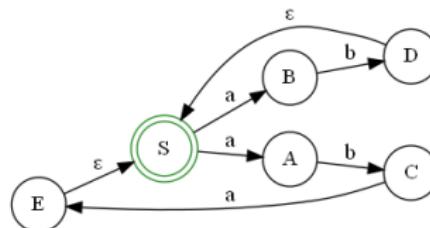
- | state | $E(state)$ |
|-------|------------|
| S     | (S)        |
| A     | (A)        |
| B     | (B)        |
| C     | (C)        |
| D     | (D S)      |
| E     | (E S)      |
- $((S) \ a \ (A \ B))$   
 $((S) \ b \ ())$

# Nondeterministic Finite Automata



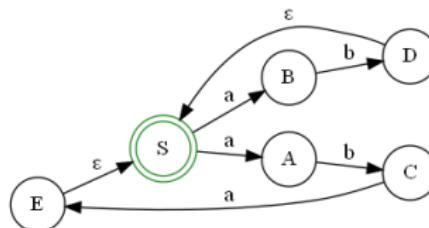
- | state | $E(state)$ |
|-------|------------|
| S     | (S)        |
| A     | (A)        |
| B     | (B)        |
| C     | (C)        |
| D     | (D S)      |
| E     | (E S)      |
- $((S) \ a \ (A \ B))$   
 $((S) \ b \ ())$
- $((A \ B) \ a \ ())$   
 $((A \ B) \ b \ (C \ D \ S))$

# Nondeterministic Finite Automata



- | state | $E(state)$ |
|-------|------------|
| S     | (S)        |
| A     | (A)        |
| B     | (B)        |
| C     | (C)        |
| D     | (D S)      |
| E     | (E S)      |
- $((S) \ a \ (A \ B))$   
 $((S) \ b \ ())$
- $((A \ B) \ a \ ())$   
 $((A \ B) \ b \ (C \ D \ S))$
- $(() \ a \ ())$   
 $(() \ b \ ())$

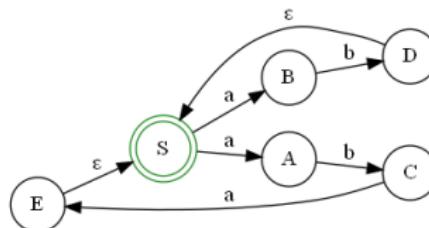
# Nondeterministic Finite Automata



state	E(state)
S	(S)
A	(A)
B	(B)
C	(C)
D	(D S)
E	(E S)

- ((S) a (A B))  
    ((S) b ( ))
- ((A B) a ( ))  
    ((A B) b (C D S))
- (( ) a ( ))  
    (( ) b ( ))
- ((C D S) a (E S A B))  
    ((C D S) b ( ))

# Nondeterministic Finite Automata



- | state | E(state) |
|-------|----------|
| S     | (S)      |
| A     | (A)      |
| B     | (B)      |
| C     | (C)      |
| D     | (D S)    |
| E     | (E S)    |
- $((S) \ a \ (A \ B))$   
 $((S) \ b \ ())$
  - $((A \ B) \ a \ ())$   
 $((A \ B) \ b \ (C \ D \ S))$
  - $(() \ a \ ())$   
 $(() \ b \ ())$
  - $((C \ D \ S) \ a \ (E \ S \ A \ B))$   
 $((C \ D \ S) \ b \ ())$
  - $((E \ S \ A \ B) \ a \ (A \ B))$   
 $((E \ S \ A \ B) \ b \ (C \ D \ S))$

# Nondeterministic Finite Automata



Super State	a	b
$(S) = S$	$(A B) A$	$() \text{ DEAD}$
$(A B) = A$	$() \text{ DEAD}$	$(C D S) B$
$(C D S) = B$	$(E S A B) C$	$() \text{ DEAD}$
$(E S A B) = C$	$(A B) A$	$(C D S) B$
$() = \text{DEAD}$	$() \text{ DEAD}$	$() \text{ DEAD}$

# Nondeterministic Finite Automata

Super State	a	b
(S) = <b>S</b>	(A B) <b>A</b>	() <b>DEAD</b>
(A B) = <b>A</b>	() <b>DEAD</b>	(C D S) <b>B</b>
(C D S) = <b>B</b>	(E S A B) <b>C</b>	() <b>DEAD</b>
(E S A B) = <b>C</b>	(A B) <b>A</b>	(C D S) <b>B</b>
() = <b>DEAD</b>	() <b>DEAD</b>	() <b>DEAD</b>

- (define D (make-dfa ` (S A B C ,DEAD)  
                 '(a b)  
                 'S  
                 '(S B C)  
                 ` ((S a A) (S b ,DEAD)  
                     (A a ,DEAD) (A b B)  
                     (B a C) (B b ,DEAD)  
                     (C a A) (C b B)  
                     (,DEAD a ,DEAD) (,DEAD b ,DEAD))))  
;; Tests for D  
(check-equal? (sm-testequiv? D ND 500) #t)  
(check-equal? (sm-testequiv? (ndfa->dfa ND) D 500) #t)

# Nondeterministic Finite Automata

Regular  
Expressions

Deterministic  
Finite  
Automata

Nondeterministic  
Finite  
Automata

Finite-State  
Automata and  
Regular  
Expressions

Regular  
Grammars

Pumping  
Theorem for  
Regular  
Languages

- Implementation

# Nondeterministic Finite Automata

- Implementation

- - ;; Data Definitions
  - ;;
  - ;; An ndfa transition rule, ndfa-rule, is a  
;; (list state symbol state)
  - ;;
  - ;; A super state, ss, is a (listof state)
  - ;;
  - ;; A super state dfa rule, ss-dfa-rule, is a  
;; (list ss symbol ss)
  - ;;
  - ;; An empties table, emps-tbl, is a  
;; (listof (list state ss))
  - ;;
  - ;; A super state name table, ss-name-table, is a  
;; (listof (list ss state))

# Nondeterministic Finite Automata

- ```
;; ndfa → dfa
;; Purpose: Convert the given ndfa to an equivalent dfa
(define (ndfa2dfa M)
  (if (eq? (sm-type M) 'dfa)
      M
      (convert (sm-states M)
                (sm-sigma M)
                (sm-start M)
                (sm-finals M)
                (sm-rules M)))))

;; Tests for ndfa2dfa
(define M (ndfa2dfa AT-LEAST-ONE-MISSING))
(check-equal? (sm-testequiv? AT-LEAST-ONE-MISSING M 500) #t)
(check-equal? (sm-testequiv? M (ndfa->dfa AT-LEAST-ONE-MISSING)

(define N (ndfa2dfa ND))
(check-equal? (sm-testequiv? ND N 500) #t)
(check-equal? (sm-testequiv? N (ndfa->dfa ND) 500) #t))
```

# Nondeterministic Finite Automata

- `;; (listof state) alphabet state (listof state) (list-of ndfa-rule)`  
`;; → dfa`  
`;; Purpose: Create a dfa from the given ndfa components`  
`(define (convert states sigma start finals rules))`

# Nondeterministic Finite Automata

- ;; Tests for convert

(check-equal?

```
(sm-testequiv? (convert '(S A B) '(a b) 'S '(A B) '((S a A)
(S a B)
(A a A)
(B b B)))
```

```
(make-ndfa '(S A B) '(a b) 'S '(A B) '((S a A)
(S a B)
(A a A)
(B b B)))
```

500)

#t)

(check-equal?

```
(sm-testequiv? (convert '(S A) '(a b) 'S '(S A) '((S a S)
(S a A)
(A b A)
(A a A)))
```

```
(make-ndfa '(S A) '(a b) 'S '(S A) '((S a S)
(S a A)
(A b A)
(A a A)))
```

500)

#t)

# Nondeterministic Finite Automata

- (define (convert states sigma start finals rules)

Regular  
Expressions

Deterministic  
Finite  
Automata

Nondeterministic  
Finite  
Automata

Finite-State  
Automata and  
Regular  
Expressions

Regular  
Grammars

Pumping  
Theorem for  
Regular  
Languages

# Nondeterministic Finite Automata

- (define (convert states sigma start finals rules))
- (let\* [(empties (compute-empties-tbl states rules))

# Nondeterministic Finite Automata

- (define (convert states sigma start finals rules)
  - (let\*
    - [(empties (compute-empties-tbl states rules))
  - (ss-dfa-rules
    - (compute-ss-dfa-rules (list (extract-empties start empties))
      - sigma empties rules '())

# Nondeterministic Finite Automata

- (define (convert states sigma start finals rules)
  - (let\*
    - [(empties (compute-empties-tbl states rules))]
  - (ss-dfa-rules
    - (compute-ss-dfa-rules (list (extract-empties start empties))
      - sigma empties rules '())
  - (super-states (remove-duplicates
    - (append-map (λ (r) (list (first r) (third r)))
      - ss-dfa-rules))

# Nondeterministic Finite Automata

- (define (convert states sigma start finals rules)
  - (let\*
    - [(empties (compute-empties-tbl states rules))]
  - (ss-dfa-rules
    - (compute-ss-dfa-rules (list (extract-empties start empties)  
sigma empties rules '()))
  - (super-states (remove-duplicates
    - (append-map (λ (r) (list (first r) (third r)))  
ss-dfa-rules)))
  - (ss-name-tbl (compute-ss-name-tbl super-states)))]

# Nondeterministic Finite Automata

- (define (convert states sigma start finals rules)
  - (let\*
    - [(empties (compute-empties-tbl states rules))]
  - (ss-dfa-rules
    - (compute-ss-dfa-rules (list (extract-empties start empties))
      - sigma empties rules '())
  - (super-states (remove-duplicates
    - (append-map (λ (r) (list (first r) (third r)))
      - ss-dfa-rules))
  - (ss-name-tbl (compute-ss-name-tbl super-states)))]
  - (make-dfa (map (λ (ss) (second (assoc ss ss-name-tbl)))
    - super-states)
    - sigma
    - (second (assoc (first super-states) ss-name-tbl))

# Nondeterministic Finite Automata

- (define (convert states sigma start finals rules)
  - (let\*
    - [(empties (compute-empties-tbl states rules))]
  - (ss-dfa-rules
    - (compute-ss-dfa-rules (list (extract-empties start empties))
      - sigma empties rules '())
  - (super-states (remove-duplicates
    - (append-map (λ (r) (list (first r) (third r)))
      - ss-dfa-rules))
  - (ss-name-tbl (compute-ss-name-tbl super-states)))]
  - (make-dfa (map (λ (ss) (second (assoc ss ss-name-tbl)))
    - super-states)
    - sigma
    - (second (assoc (first super-states) ss-name-tbl))
  - (map (λ (ss) (second (assoc ss ss-name-tbl)))
    - (filter (λ (ss) (ormap (λ (s) (member s finals))
      - super-states))

# Nondeterministic Finite Automata

- (define (convert states sigma start finals rules)
  - (let\*
    - [(empties (compute-empties-tbl states rules))]
  - (ss-dfa-rules
    - (compute-ss-dfa-rules (list (extract-empties start empties))
      - sigma empties rules '())
  - (super-states (remove-duplicates
    - (append-map (λ (r) (list (first r) (third r)))
      - ss-dfa-rules))
  - (ss-name-tbl (compute-ss-name-tbl super-states)))]
  - (make-dfa (map (λ (ss) (second (assoc ss ss-name-tbl)))
    - super-states)
    - sigma
    - (second (assoc (first super-states) ss-name-tbl))
  - (map (λ (ss) (second (assoc ss ss-name-tbl)))
    - (filter (λ (ss) (ormap (λ (s) (member s finals))
      - super-states))
  - (map (λ (r) (list (second (assoc (first r) ss-name-tbl))
    - (second r)
    - (second (assoc (third r) ss-name-tbl))
  - ss-dfa-rules)
  - 'no-dead))))

# Nondeterministic Finite Automata

- `;; (listof state) rules → emps-tbl`  
`;; Purpose: Compute empties table for all given states`  
`(define (compute-empties-tbl states rules)`

# Nondeterministic Finite Automata

- `;; (listof state) rules → emps-tbl`  
`;; Purpose: Compute empties table for all given states`  
`(define (compute-empties-tbl states rules)`
- `;; Tests for compute-empties-tbl`  
`(check-equal? (compute-empties-tbl`  
 `'(X Y Z)`  
 ``((X ,EMP Y) (Y a Z) (Z ,EMP X)))`  
 ``((X (Y X)) (Y (Y)) (Z (Y X Z))))`
- `(check-equal?`  
 `(compute-empties-tbl`  
 `'(W X Y Z)`  
 ``((W ,EMP X) (X ,EMP Y) (Y a Z) Z ,EMP Y) (Z b Z)))`  
 ``((W (Y X W)) (X (Y X)) (Y (Y)) (Z (Y Z))))`

# Nondeterministic Finite Automata

- (define (compute-empties-tbl states rules)

# Nondeterministic Finite Automata

- (define (compute-empties-tbl states rules)

- (map (λ (st) (list st (compute-empties (list st) rules '())))) states))

# Nondeterministic Finite Automata

- (define (compute-empties-tbl states rules)
  - ```
;; (listof state) (listof ndfa-rules) (listof state) → (listof state)
;; Purpose: Compute the empties for the states left to explore in the first
;;           given (listof state)
;; Accumulator Invariants:
;;   to-search = unvisited states reachable by consuming no input
;;   visited = visited states reachable by consuming no input
(define (compute-empties to-search rules visited)
  (if (empty? to-search)
      visited
      (let* [(curr (first to-search))
             (curr-e-rules
              (get-e-trans curr (append to-search visited) rules))]
        (compute-empties (append (rest to-search) (map third curr-e-rules))
                        rules
                        (cons curr visited))))))
```
  - (map (λ (st) (list st (compute-empties (list st) rules '()))) states))

# Nondeterministic Finite Automata

- (define (compute-empties-tbl states rules)
  - ;; state (listof state) (listof ndfa-rule) → (listof ndfa-rule)  
;; Purpose: Extract empty transitions to non-generated states for the  
;; given state  
(define (get-e-trans state gen-states rules)
    - (filter (λ (r) (and (eq? (first r) state)  
(eq? (second r) EMP)  
(not (member (third r) gen-states))))  
rules))
  - ;; (listof state) (listof ndfa-rules) (listof state) → (listof state)  
;; Purpose: Compute the empties for the states left to explore in the first  
;; given (listof state)  
;; Accumulator Invariants:  
;; to-search = unvisited states reachable by consuming no input  
;; visited = visited states reachable by consuming no input  
(define (compute-empties to-search rules visited)
    - (if (empty? to-search)  
visited  
(let\* [(curr (first to-search))  
(curr-e-rules  
          (get-e-trans curr (append to-search visited) rules))]  
(compute-empties (append (rest to-search) (map third curr-e-rules))  
rules  
(cons curr visited))))
  - (map (λ (st) (list st (compute-empties (list st) rules '())))) states))

# Nondeterministic Finite Automata

- 

```
;; state emps-tbl → ss
;; Purpose: Extract the empties of the given state
;; Assume: Given state is in the given list of states
(define (extract-empties st empties)
  (second (first (filter (λ (e) (eq? (first e) st))
                           empties)))))

;; Tests for extract-empties
(check-equal? (extract-empties 'A '((S (S B))
                                         (F (F))
                                         (A (A C D))
                                         (C (C))
                                         (D (D)))))
               '(A C D))

(check-equal? (extract-empties 'Z '((Z (Z S))
                                         (S ()))))
               '(Z S))
```

# Nondeterministic Finite Automata

- ```
;; (listof ss) alphabet emps-tbl (listof ndfa-rule) (listof ss)
;;   → (listof ss-dfa-rule)
;; Purpose: Compute the supper state dfa rules
;; Accumulator Invariants:
;;           ssts = the super states explored
;;           to-search-ssts = the super states that must still be explored
(define (compute-ss-dfa-rules to-search-ssts sigma empties rules ssts)
  :
  :
```

# Nondeterministic Finite Automata

- ```
;; (listof ss) alphabet emps-tbl (listof ndfa-rule) (listof ss)
;;                                         → (listof ss-dfa-rule)
;; Purpose: Compute the supper state dfa rules
;; Accumulator Invariants:
;;           ssts = the super states explored
;;           to-search-ssts = the super states that must still be explored
(define (compute-ss-dfa-rules to-search-ssts sigma empties rules ssts)
  :
  :
  • (if (empty? to-search-ssts)
        '()
```

# Nondeterministic Finite Automata

- ```
;; (listof ss) alphabet emps-tbl (listof ndfa-rule) (listof ss)
;;   → (listof ss-dfa-rule)
;; Purpose: Compute the supper state dfa rules
;; Accumulator Invariants:
;;           ssts = the super states explored
;;           to-search-ssts = the super states that must still be explored
(define (compute-ss-dfa-rules to-search-ssts sigma empties rules ssts)
  :
  :
  • (if (empty? to-search-ssts)
        '()
      • (let* [(curr-ss (first to-search-ssts))
              (reachables (find-reachables curr-ss sigma rules empties))]
```

# Nondeterministic Finite Automata

- ```
;; (listof ss) alphabet emps-tbl (listof ndfa-rule) (listof ss)
;;                                         → (listof ss-dfa-rule)
;; Purpose: Compute the supper state dfa rules
;; Accumulator Invariants:
;;           ssts = the super states explored
;;           to-search-ssts = the super states that must still be explored
(define (compute-ss-dfa-rules to-search-ssts sigma empties rules ssts)
  :
  :
  • (if (empty? to-search-ssts)
      '()
      • (let* [(curr-ss (first to-search-ssts))
              (reachables (find-reachables curr-ss sigma rules empties))]
          • (to-super-states
              (build-list (length sigma) (λ (i) (get-reachable i reachables))))
```

# Nondeterministic Finite Automata

- ```
;; (listof ss) alphabet emps-tbl (listof ndfa-rule) (listof ss)
;;   → (listof ss-dfa-rule)
;; Purpose: Compute the supper state dfa rules
;; Accumulator Invariants:
;;           ssts = the super states explored
;;           to-search-ssts = the super states that must still be explored
(define (compute-ss-dfa-rules to-search-ssts sigma empties rules ssts)
  :
  :
  • (if (empty? to-search-ssts)
      '()
      • (let* [(curr-ss (first to-search-ssts))
              (reachables (find-reachables curr-ss sigma rules empties))]
          • (to-super-states
              (build-list (length sigma) (λ (i) (get-reachable i reachables))))
          • (new-rules (map (λ (sst a) (list curr-ss a sst))
                            to-super-states
                            sigma))]
```

# Nondeterministic Finite Automata

- ```
;; (listof ss) alphabet emps-tbl (listof ndfa-rule) (listof ss)
;;                                         → (listof ss-dfa-rule)
;; Purpose: Compute the supper state dfa rules
;; Accumulator Invariants:
;;           ssts = the super states explored
;;           to-search-ssts = the super states that must still be explored
(define (compute-ss-dfa-rules to-search-ssts sigma empties rules ssts)
  :
  :
  • (if (empty? to-search-ssts)
      '()
      • (let* [(curr-ss (first to-search-ssts))
              (reachables (find-reachables curr-ss sigma rules empties))]
          • (to-super-states
              (build-list (length sigma) (λ (i) (get-reachable i reachables))))
          • (new-rules (map (λ (sst a) (list curr-ss a sst))
                            to-super-states
                            sigma))]
          • (append
              new-rules
              (compute-ss-dfa-rules
                (append (rest to-search-ssts)
                        (filter (λ (ss)
                                  (not (member ss (append to-search-ssts ssts))))
                                to-super-states)
                sigma
                empties
                rules
                (cons curr-ss ssts)))))))
```

# Nondeterministic Finite Automata

- ```
;; ss alphabet (listof ndfa-rule) emps-tbl
;;
;; Purpose: Compute reachable super states from given
;;           super state
(define (find-reachables ss sigma rules empties)
  (map (λ (st)
          (find-reachables-from-st st sigma rules empties))
        ss))
```

# Nondeterministic Finite Automata

- ```
;; state alphabet (listof ndfa-rule) emps-tbl → (listof ss)
;; Purpose: Find the reachable super state from the given state
;;           for each element of the given alphabet
(define (find-reachables-from-st st sigma rules empties)
  (map (λ (a)
           (find-reachables-from-st-on-a st a rules empties))
        sigma))
```
- ```
;; ss alphabet (listof ndfa-rule) emps-tbl → (listof (listof ss))
;; Purpose: Compute reachable super states from given
;;           super state
(define (find-reachables ss sigma rules empties)
  (map (λ (st)
           (find-reachables-from-st st sigma rules empties))
        ss))
```

# Nondeterministic Finite Automata

- ```
;; state symbol (listof ndfa-rule) emps-tbl → ss
;; Purpose: Find the reachable super state from the given state
;;           and the given alphabet element
(define (find-reachables-from-st-on-a st a rules empties)
  (let* [(rls (filter
                (λ (r)
                  (and (eq? (first r) st) (eq? (second r) a)))
                rules))
         (to-states (map third rls))]
    (remove-duplicates
      (append-map (λ (st) (extract-empties st empties))
                  to-states))))
```
- ```
;; state alphabet (listof ndfa-rule) emps-tbl → (listof ss)
;; Purpose: Find the reachable super state from the given state
;;           for each element of the given alphabet
(define (find-reachables-from-st st sigma rules empties)
  (map (λ (a)
         (find-reachables-from-st-on-a st a rules empties)))
       sigma))
```
- ```
;; ss alphabet (listof ndfa-rule) emps-tbl → (listof (listof ss))
;; Purpose: Compute reachable super states from given
;;           super state
(define (find-reachables ss sigma rules empties)
  (map (λ (st)
         (find-reachables-from-st st sigma rules empties))
       ss))
```

# Nondeterministic Finite Automata

- ```
;; natnum (listof (listof ss)) → (listof ss)
;; Purpose: Return ss of ith (listof state) in each given
;;           list element
(define (get-reachable i reachables)
  (remove-duplicates (append-map
                      (λ (reached) (list-ref reached i))
                      reachables)))
```

# Nondeterministic Finite Automata

- `;; (listof ss) → ss-name-tbl`  
;; Purpose: Create a table for ss names  
`(define (compute-ss-name-tbl super-states)`

# Nondeterministic Finite Automata

- ```
;; (listof ss) → ss-name-tbl
;; Purpose: Create a table for ss names
(define (compute-ss-name-tbl super-states)
```
- ```
;; Tests for compute-ss-name-tbl
(check-pred (lambda (tbl)
    (and (list? tbl)
        (andmap (λ (e) (= (length e) 2)) tbl)
        (andmap (λ (e) (andmap symbol? (first e)))
            tbl)
        (andmap (λ (e) (symbol? (second e))) tbl)))
    (compute-ss-name-tbl '())))

(check-pred (lambda (tbl)
    (and (list? tbl)
        (andmap (λ (e) (= (length e) 2)) tbl)
        (andmap (λ (e) (andmap symbol? (first e)))
            tbl)
        (andmap (λ (e) (symbol? (second e))) tbl)))
    (compute-ss-name-tbl '((A B) (A B C) () (C))))
```

# Nondeterministic Finite Automata

- `;; (listof ss) → ss-name-tbl`  
`;; Purpose: Create a table for ss names`  
`(define (compute-ss-name-tbl super-states)`
- `(map (λ (ss)`  
 `(if (empty? ss)`  
 `(list ss DEAD)`  
 `(list ss (generate-symbol 'Z '(Z))))))`  
 `super-states))`
- `;; Tests for compute-ss-name-tbl`  
`(check-pred (lambda (tbl)`  
 `(and (list? tbl)`  
 `(andmap (λ (e) (= (length e) 2)) tbl)`  
 `(andmap (λ (e) (andmap symbol? (first e)))`  
 `tbl)`  
 `(andmap (λ (e) (symbol? (second e))) tbl))))`  
`(compute-ss-name-tbl '())))`
- `(check-pred (lambda (tbl)`  
 `(and (list? tbl)`  
 `(andmap (λ (e) (= (length e) 2)) tbl)`  
 `(andmap (λ (e) (andmap symbol? (first e)))`  
 `tbl)`  
 `(andmap (λ (e) (symbol? (second e))) tbl))))`  
`(compute-ss-name-tbl '((A B) (A B C) () (C))))`

# Nondeterministic Finite Automata

- Correctness Proof

# Nondeterministic Finite Automata

- Correctness Proof
- $ND = (\text{make-ndfa } S \Sigma A F \delta)$

$D = (\text{make-dfa } S' \Sigma A' F' \delta')$ . where

$S'$  = the states computed in convert

$A'$  = the starting state computed in convert

$F'$  = the final states computed in convert

$\delta'$  = the transition function computed in convert

- We need to prove  $L(ND) = L(D)$

# Nondeterministic Finite Automata

## Theorem

$(w Q) \vdash^*_{ND} ((P)) \Leftrightarrow (w Q') \vdash^*_D ((P'))$ , where  $Q' = E(Q) \wedge P \in P'$

- Base Case
- $\Rightarrow$  Assume  $(w Q) \vdash^*_{ND} ((P))$

Base Case:  $|w| = 0$   $|w| = 0 \Rightarrow w = ()$

By assumption,  $((Q)) \vdash^*_{ND} ((P))$ . This means that  $P \in E(Q)$

By construction of D,  $P \in Q'$ . This means that ND's computation is carried out as follows by D:  
 $((Q')) \vdash ((Q'))$

This establishes the base case.

# Nondeterministic Finite Automata

## Theorem

$(w Q) \vdash^*_{ND} ((P)) \Leftrightarrow (w Q') \vdash^*_D ((P'))$ , where  $Q' = E(Q) \wedge P \in P'$

- Base Case

- ( $\Rightarrow$ ) Assume  $(w Q) \vdash^*_{ND} ((P))$

Base Case:  $|w| = 0 \Rightarrow w = ()$

By assumption,  $((Q)) \vdash^*_{ND} ((P))$ . This means that  $P \in E(Q)$

By construction of D,  $P \in Q'$ . This means that ND's computation is carried out as follows by D:  
 $((Q')) \vdash ((Q'))$

This establishes the base case.

- Inductive Step:

Assume:  $(w Q) \vdash^*_{ND} ((P)) \Rightarrow (w Q') \vdash^*_D ((P'))$ , where  $Q' = E(Q) \wedge P \in P'$ , for  $|w|=k$   
Show:  $(w Q) \vdash^*_{ND} ((P)) \Rightarrow (w Q') \vdash^*_D ((P'))$ , where  $Q' = E(Q) \wedge P \in P'$ , for  $|w|=k+1$

$|w|=k+1 \Rightarrow w=(xa)$ , where  $x \in \Sigma^*$  and  $a \in \Sigma$ .

To prove the implication assume  $((xa) Q) \vdash^*_{ND} ((P))$ .

This means that ND's computation is:

$((xa) Q) \vdash^*_{ND} ((a) R) \vdash_{ND} ((T)) \vdash^*_{ND} ((P))$

That is, consuming x takes ND from Q to some intermediate state R. From R on an a ND goes to T. Then by  $\epsilon$ -transitions ND gets to P.

By inductive hypothesis:

$((xa) Q') \vdash^*_D ((a) R')$ , where  $R \in R'$

Observe that  $(R \ a \ T) \in \delta$  and  $P \in E(T)$ . Let  $P' \in E(T)$ . By construction of D, this means that the following is D's computation:

$((xa) Q') \vdash^*_D ((a) R') \vdash_D ((P'))$ , where  $P \in P'$ .

Clearly, we have that  $(w Q') \vdash^*_D ((P'))$ , where  $P \in P'$ . This completes the proof of the implication.

# Nondeterministic Finite Automata

## Theorem

$(w Q) \vdash^*_{ND} (\lambda P) \Leftrightarrow (w Q') \vdash^*_D (\lambda P')$ , where  $Q' = E(Q) \wedge P \in P'$

- ( $\Leftarrow$ ) Assume:  $(w Q') \vdash^*_D (\lambda P')$ , where  $Q' = E(Q) \wedge P \in P'$ , for  $|w|=k+1$ .  
Base Case:  $|w| = 0$   $|w| = 0 \Rightarrow w = \lambda$   
By assumption,  $Q' = P'$  because D is deterministic. This means that  $P \in E(Q)$ .  
By construction of D:  
 $(\lambda Q) \vdash^*_{ND} (\lambda P)$   
This establishes the base case.

# Nondeterministic Finite Automata

## Theorem

$(w Q) \vdash^*_{ND} ((P)) \Leftrightarrow (w Q') \vdash^*_D ((P'))$ , where  $Q' = E(Q) \wedge P \in P'$

- ( $\Leftarrow$ ) Assume:  $(w Q') \vdash^*_D ((P'))$ , where  $Q' = E(Q) \wedge P \in P'$ , for  $|w|=k+1$ .  
Base Case:  $|w|=0 \Rightarrow w=()$   
By assumption,  $Q' = P'$  because D is deterministic. This means that  $P \in E(Q)$ .  
By construction of D:  
 $((Q) \vdash^*_{ND} ((P))$   
This establishes the base case.
- Inductive Step:  
Assume:  $(w Q') \vdash^*_D ((P')) \Rightarrow (w Q) \vdash^*_{ND} ((P))$ , where  $Q' = E(Q) \wedge P \in P'$ , for  $|w|=k$   
Show:  $(w Q') \vdash^*_D ((P')) \Rightarrow (w Q) \vdash^*_{ND} ((P))$ , where  $Q' = E(Q) \wedge P \in P'$ , for  $|w|=k+1$   
 $|w|=k+1 \Rightarrow w=(xa)$ , where  $x \in \Sigma^*$  and  $a \in \Sigma$ .  
To prove the implication assume  $(w Q') \vdash^*_D ((P'))$ , where  $Q' = E(Q) \wedge P \in P'$ , for  $|w|=k+1$   
This means that D's computation is:  
 $((xa) Q') \vdash^*_D ((a) R') \vdash_D ((P'))$   
By construction of D, the above means:  
 $((xa) Q) \vdash^*_{ND} ((a) R) \vdash_{ND} ((T)) \vdash^*_{ND} ((P))$ , where  $R \in R'$  and  $P \in E(T)=P'$ .  
This completes the proof of the theorem.

# Nondeterministic Finite Automata

## Lemma

$$w \in L(ND) \Leftrightarrow w \in L(D)$$

# Nondeterministic Finite Automata

## Lemma

$$w \in L(ND) \Leftrightarrow w \in L(D)$$

- ( $\Rightarrow$ ) Assume  $w \in L(ND)$

This means that:

$(w S) \vdash^*_{ND} (( P), \text{ where } P \in F)$ .

By Theorem, we have that:

$(w S') \vdash^*_{D'} (( P'), \text{ where } P' \in F')$ .

By construction of D,  $P' \in F'$ . Thus,  $w \in L(D)$

# Nondeterministic Finite Automata

## Lemma

$$w \in L(ND) \Leftrightarrow w \in L(D)$$

- ( $\Rightarrow$ ) Assume  $w \in L(ND)$

This means that:

$(w S) \vdash^*_{ND} (( P), \text{ where } P \in F)$ .

By Theorem, we have that:

$(w S') \vdash^*_{D} (( P'), \text{ where } P' \in P')$ .

By construction of D,  $P' \in F'$ . Thus,  $w \in L(D)$

- ( $\Leftarrow$ ) Assume  $w \in L(D)$

This means that:

$(w S') \vdash^*_{D} (( P'), \text{ where } P' \in F')$ .

By Theorem and construction of D, we have that:

$(w S) \vdash^*_{ND} (( P), \text{ where } P \in F \text{ and } P \in P')$ .

Thus,  $w \in L(ND)$

# Nondeterministic Finite Automata

## Lemma

$$w \notin L(ND) \Leftrightarrow w \notin L(D)$$

# Nondeterministic Finite Automata

## Lemma

$$w \notin L(ND) \Leftrightarrow w \notin L(D)$$

- ( $\Rightarrow$ ) Assume  $w \notin L(ND)$   
This means that for all ND computations:  
 $(w S) \vdash^*_{ND} (( P), \text{ where } P \notin F).$   
By Theorem, we have that:  
 $(w S') \vdash^*_{D'} (( P'), \text{ where } P' \in P').$   
By construction of D,  $P' \notin F'$ . Thus,  $w \notin L(D)$

# Nondeterministic Finite Automata

## Lemma

$$w \notin L(ND) \Leftrightarrow w \notin L(D)$$

- ( $\Rightarrow$ ) Assume  $w \notin L(ND)$

This means that for all ND computations:

$(w S) \vdash^*_{ND} (( P), \text{ where } P \notin F)$ .

By Theorem, we have that:

$(w S') \vdash^*_{D} (( P'), \text{ where } P \in P')$ .

By construction of D,  $P' \notin F'$ . Thus,  $w \notin L(D)$

- ( $\Leftarrow$ ) Assume  $w \notin L(D)$

This means that:

$(w S') \vdash^*_{D} (( P'), \text{ where } P' \notin F')$ .

By Theorem and construction of D, we have that:

$(w S) \vdash^*_{ND} (( P), \text{ where } P \notin F \text{ and } P \in P')$ .

Thus,  $w \notin L(ND)$ .

# Nondeterministic Finite Automata

## Theorem

$$L(ND) = L(D)$$

- Follows from the two previous lemmas.

# Nondeterministic Finite Automata

- It is a remarkable result that endowing dfas with nondeterminism yields no extra computational power

# Nondeterministic Finite Automata

- It is a remarkable result that endowing dfas with nondeterminism yields no extra computational power
- Does this mean that ndfas are worthless?

# Nondeterministic Finite Automata

- It is a remarkable result that endowing dfas with nondeterminism yields no extra computational power
- Does this mean that ndfas are worthless?
- For a computer scientist the answer is clearly no
- Why?

# Nondeterministic Finite Automata

- It is a remarkable result that endowing dfas with nondeterminism yields no extra computational power
- Does this mean that ndfas are worthless?
- For a computer scientist the answer is clearly no
- Why?
- Make the design process easier
- A useful abstraction programmers may use to design solutions using a dfa

# Nondeterministic Finite Automata

- QUIZ: 6 (due in 1 week)

# FSA and Regular Expressions

- Finite-state automata decide regular languages?

# FSA and Regular Expressions

- Finite-state automata decide regular languages?
- We have seen DFA examples that read concatenated symbols or that loop to read a collection of concatenated symbols an arbitrary number of times
- This suggests that the languages they decide may be closed under concatenation and Kleene star

# FSA and Regular Expressions

Regular  
Expressions

Deterministic  
Finite  
Automata

Nondeterministic  
Finite  
Automata

Finite-State  
Automata and  
Regular  
Expressions

Regular  
Grammars

Pumping  
Theorem for  
Regular  
Languages

- Finite-state automata decide regular languages?
- We have seen dfa examples that read concatenated symbols or that loop to read a collection of concatenated symbols an arbitrary number of times
- This suggests that the languages they decide may be closed under concatenation and Kleene star
- We have seen ndfa examples that suggest the languages they decide are closed under union
- These are operations used by regular expressions!

# FSA and Regular Expressions

- We ought to be able to combine the languages decided by finite-state automatas using concatenation, union, and Kleene star to create machines for bigger languages
- Such an ability would provide programmers with a new set of constructors to create finite-state automatas
- Simplifying the amount of work a programmer must do to create “complex” finite-state automatas

# FSA and Regular Expressions

## Theorem

*The languages decided by finite-state automata are closed under:*

1. union
2. concatenation
3. Kleene star
4. complement
5. intersection

# FSA and Regular Expressions

## Theorem

*The languages decided by finite-state automata are closed under:*

1. union
2. concatenation
3. Kleene star
4. complement
5. intersection

- The proof is divided into 5 theorems
- They are all proven using a constructive proof

# FSA and Regular Expressions

- To test the constructors the following machines are defined:

```
;; L = ab*
(define ab* (make-ndfa '(S A) '(a b) 'S '(A)
                           '((S a A) (A b A))))
;; L = a(a U ab)b*
(define a-aUb-b* (make-ndfa '(Z H B C D F)
                                '(a b)
                                'Z
                                '(F)
                                `((Z a H)
                                   (Z a B) (H a D) (D ,EMP F)
                                   (B a C) (C b F) (F b F))))
;; L = aab*
(define aab* (make-ndfa '(W X Y) '(a b) 'W '(Y)
                           '((W a X) (X a Y) (Y b Y))))
;; L = a*
(define a* (make-dfa '(S D)
                      '(a b)
                      'S
                      '(S)
                      `((S a S) (S b D) (D a D) (D b D))
                      'no-dead))
```

# FSA and Regular Expressions

## Theorem

*The languages accepted by finite-state machines are closed under union.*

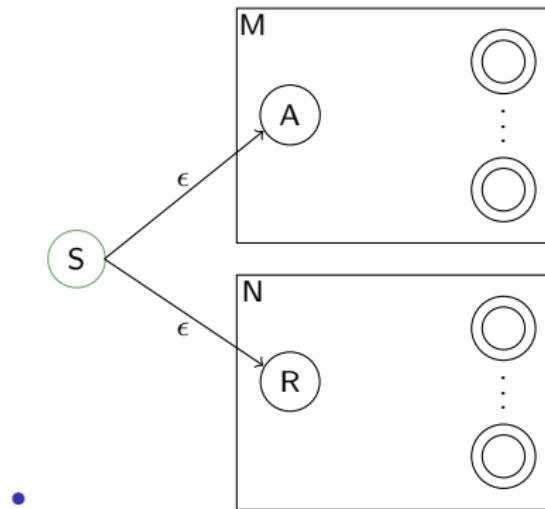
- Let the following be the two machines that decide the languages to union:  
 $M = (\text{make-ndfa } S_M \Sigma_M A F_M \delta_M)$   
 $N = (\text{make-ndfa } S_N \Sigma_N R F_N \delta_N)$
- We need to construct an ndfa that decides  $L = L(M) \cup L(N)$

# FSA and Regular Expressions

## Theorem

*The languages accepted by finite-state machines are closed under union.*

- Let the following be the two machines that decide the languages to union:  
 $M = (\text{make-ndfa } S_M \Sigma_M A F_M \delta_M)$   
 $N = (\text{make-ndfa } S_N \Sigma_N R F_N \delta_N)$
- We need to construct an ndfa that decides  $L = L(M) \cup L(N)$



# FSA and Regular Expressions

- `;; ndfa ndfa → ndfa`  
`;; Purpose: Construct ndfa for the union of given ndfas`  
`;; Assume: The intersection of states is empty`  
`(define (union-fsa M N)`

# FSA and Regular Expressions

- ```
;; ndfa ndfa → ndfa
;; Purpose: Construct ndfa for the union of given ndfas
;; Assume: The intersection of states is empty
(define (union-fsa M N))
```

- ```
;; Tests for union-fsa
(define ab*Ua-aUb-b* (union-fsa ab* a-aUb-b*))
(define ab*Uaab* (union-fsa ab* aab*))
```

# FSA and Regular Expressions

- ```
;; ndfa ndfa → ndfa
;; Purpose: Construct ndfa for the union of given ndfas
;; Assume: The intersection of states is empty
(define (union-fsa M N))
```

- ```
;; Tests for union-fsa
(define ab*Ua-aUb-b* (union-fsa ab* a-aUb-b*))
(define ab*Uaab* (union-fsa ab* aab*))
```
- ```
(check-equal? (sm-apply ab*Ua-aUb-b* '()) 'reject)
(check-equal? (sm-apply ab*Ua-aUb-b* '(a a a a)) 'reject)
(check-equal? (sm-apply ab*Ua-aUb-b* '(a b)) 'accept)
(check-equal? (sm-apply ab*Ua-aUb-b* '(a a b b)) 'accept)
(check-equal? (sm-testequiv? ab*Ua-aUb-b* (sm-union ab* ab*Ua-aUb-b*)))
(check-equal? (sm-apply ab*Uaab* '(a a a)) 'reject)
```

# FSA and Regular Expressions

- ```
;; ndfa ndfa → ndfa
;; Purpose: Construct ndfa for the union of given ndfas
;; Assume: The intersection of states is empty
(define (union-fsa M N)
  (let* [(new-start (generate-symbol
                     'S (append (sm-states M) (sm-states N))))]
```

- ```
;; Tests for union-fsa
(define ab*Ua-aUb-b* (union-fsa ab* a-aUb-b*))
(define ab*Uaab* (union-fsa ab* aab*))
```
- ```
(check-equal? (sm-apply ab*Ua-aUb-b* '()) 'reject)
(check-equal? (sm-apply ab*Ua-aUb-b* '(a a a a)) 'reject)
(check-equal? (sm-apply ab*Ua-aUb-b* '(a b)) 'accept)
(check-equal? (sm-apply ab*Ua-aUb-b* '(a a b b)) 'accept)
(check-equal? (sm-testequiv? ab*Ua-aUb-b* (sm-union ab* ab*Ua-aUb-b*))
  (check-equal? (sm-apply ab*Uaab* '(a a a)) 'reject))
```

# FSA and Regular Expressions

- ```
;; ndfa ndfa → ndfa
;; Purpose: Construct ndfa for the union of given ndfas
;; Assume: The intersection of states is empty
(define (union-fsa M N))
```
- ```
(let* [(new-start (generate-symbol
'S (append (sm-states M) (sm-states N))))]
```
- ```
(new-sigma (remove-duplicates
(append (sm-sigma M) (sm-sigma N))))
```

- ```
;; Tests for union-fsa
(define ab*Ua-aUb-b* (union-fsa ab* a-aUb-b*))
(define ab*Uaab* (union-fsa ab* aab*))
```
- ```
(check-equal? (sm-apply ab*Ua-aUb-b* '()) 'reject)
(check-equal? (sm-apply ab*Ua-aUb-b* '(a a a a)) 'reject)
(check-equal? (sm-apply ab*Ua-aUb-b* '(a b)) 'accept)
(check-equal? (sm-apply ab*Ua-aUb-b* '(a a b b)) 'accept)
(check-equal? (sm-testequiv? ab*Ua-aUb-b* (sm-union ab* ab*Ua-aUb-b*))
(check-equal? (sm-apply ab*Uaab* '(a a a)) 'reject)
```

# FSA and Regular Expressions

- ```
;; ndfa ndfa → ndfa
;; Purpose: Construct ndfa for the union of given ndfas
;; Assume: The intersection of states is empty
(define (union-fsa M N))
```
- ```
(let* [(new-start (generate-symbol
                     'S (append (sm-states M) (sm-states N))))]
```
- ```
(new-sigma (remove-duplicates
                     (append (sm-sigma M) (sm-sigma N)))))
```
- ```
(new-states (cons new-start
                     (append (sm-states M) (sm-states N)))))
```

- ```
;; Tests for union-fsa
(define ab*Ua-aUb-b* (union-fsa ab* a-aUb-b*))
(define ab*Uaab* (union-fsa ab* aab*))
```
- ```
(check-equal? (sm-apply ab*Ua-aUb-b* '()) 'reject)
(check-equal? (sm-apply ab*Ua-aUb-b* '(a a a a)) 'reject)
(check-equal? (sm-apply ab*Ua-aUb-b* '(a b)) 'accept)
(check-equal? (sm-apply ab*Ua-aUb-b* '(a a b b)) 'accept)
(check-equal? (sm-testequiv? ab*Ua-aUb-b* (sm-union ab* ab*Ua-aUb-b*))
(check-equal? (sm-apply ab*Uaab* '(a a a)) 'reject)
```

# FSA and Regular Expressions

- `;; ndfa ndfa → ndfa`  
`;; Purpose: Construct ndfa for the union of given ndfas`  
`;; Assume: The intersection of states is empty`  
`(define (union-fsa M N)`
- `(let* [(new-start (generate-symbol`  
                  `'S (append (sm-states M) (sm-states N))))`
- `(new-sigma (remove-duplicates`  
                  `(append (sm-sigma M) (sm-sigma N))))`
- `(new-states (cons new-start`  
                  `(append (sm-states M) (sm-states N))))`
- `(new-finals (append (sm-finals M) (sm-finals N))))`

- `;; Tests for union-fsa`  
`(define ab*Ua-aUb-b* (union-fsa ab* a-aUb-b*))`  
`(define ab*Uaab* (union-fsa ab* aab*))`
- `(check-equal? (sm-apply ab*Ua-aUb-b* '()) 'reject)`  
`(check-equal? (sm-apply ab*Ua-aUb-b* '(a a a a)) 'reject)`  
`(check-equal? (sm-apply ab*Ua-aUb-b* '(a b)) 'accept)`  
`(check-equal? (sm-apply ab*Ua-aUb-b* '(a a b b)) 'accept)`  
`(check-equal? (sm-testequiv? ab*Ua-aUb-b* (sm-union ab* ab*Ua-aUb-b*))`  
`(check-equal? (sm-apply ab*Uaab* '(a a a)) 'reject)`

# FSA and Regular Expressions

- ```
;; ndfa ndfa → ndfa
;; Purpose: Construct ndfa for the union of given ndfas
;; Assume: The intersection of states is empty
(define (union-fsa M N))
```
- ```
(let* [(new-start (generate-symbol
                      'S (append (sm-states M) (sm-states N))))]
```
- ```
(new-sigma (remove-duplicates
                      (append (sm-sigma M) (sm-sigma N)))))
```
- ```
(new-states (cons new-start
                      (append (sm-states M) (sm-states N)))))
```
- ```
(new-finals (append (sm-finals M) (sm-finals N))))
```
- ```
(new-rules (append (list (list new-start EMP (sm-start M))
                                (list new-start EMP (sm-start N)
                                      (sm-rules M)
                                      (sm-rules N))))]
```
- ```
;; Tests for union-fsa
(define ab*Ua-aUb-b* (union-fsa ab* a-aUb-b*))
(define ab*Uaab* (union-fsa ab* aab*))
```
- ```
(check-equal? (sm-apply ab*Ua-aUb-b* '()) 'reject)
(check-equal? (sm-apply ab*Ua-aUb-b* '(a a a a)) 'reject)
(check-equal? (sm-apply ab*Ua-aUb-b* '(a b)) 'accept)
(check-equal? (sm-apply ab*Ua-aUb-b* '(a a b b)) 'accept)
(check-equal? (sm-testequiv? ab*Ua-aUb-b* (sm-union ab* ab*Ua-aUb-b*))
(check-equal? (sm-apply ab*Uaab* '(a a a)) 'reject) ▶ ⏴ ⏵ ⏵ ⏵ ⏵ ⏵ ⏵ ⏵
```

# FSA and Regular Expressions

- ```
;; ndfa ndfa → ndfa
;; Purpose: Construct ndfa for the union of given ndfas
;; Assume: The intersection of states is empty
(define (union-fsa M N)
  (let* [(new-start (generate-symbol
                     'S (append (sm-states M) (sm-states N))))]
    (new-sigma (remove-duplicates
                (append (sm-sigma M) (sm-sigma N))))
    (new-states (cons new-start
                      (append (sm-states M) (sm-states N)))))
    (new-finals (append (sm-finals M) (sm Finals N)))
    (new-rules (append (list (list new-start EMP (sm-start M))
                             (list new-start EMP (sm-start N))
                             (sm-rules M)
                             (sm-rules N)))]
      (make-ndfa new-states new-sigma new-start new-finals new-rules)))
  ;; Tests for union-fsa
  (define ab*Ua-aUb-b* (union-fsa ab* a-aUb-b*))
  (define ab*Uaab* (union-fsa ab* aab*))
  (check-equal? (sm-apply ab*Ua-aUb-b* '()) 'reject)
  (check-equal? (sm-apply ab*Ua-aUb-b* '(a a a a)) 'reject)
  (check-equal? (sm-apply ab*Ua-aUb-b* '(a b)) 'accept)
  (check-equal? (sm-apply ab*Ua-aUb-b* '(a a b b)) 'accept)
  (check-equal? (sm-testequiv? ab*Ua-aUb-b* (sm-union ab* ab*Ua-aUb-b*)
    (check-equal? (sm-apply ab*Uaab* '(a a a)) 'reject)
```

# FSA and Regular Expressions

## Lemma

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

- Define three machines as follows:

$$M = (\text{make-ndfa } S \ \Sigma \ Z \ F \ \delta)$$

$$N = (\text{make-ndfa } S' \ \Sigma' \ Z' \ F' \ \delta')$$

$$U = (\text{union-fsa } M \ N) = (\text{make-ndfa } S'' \ \Sigma'' \ Z'' \ F'' \ \delta'')$$

- Let  $L = L(M) \cup L(N)$

# FSA and Regular Expressions

## Lemma

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

- ( $\Rightarrow$ ) We need to show that  $w \in L \Rightarrow w \in L(U)$

# FSA and Regular Expressions

## Lemma

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

- ( $\Rightarrow$ ) We need to show that  $w \in L \Rightarrow w \in L(U)$
- Assume  $w \in L$
- This means that  $w \in L(M)$  or  $w \in L(N)$
- By construction,  $U$  nondeterministically correctly chooses to simulate  $M$  or to simulate  $N$
- Thus,  $w \in L(U)$ .

# FSA and Regular Expressions

## Lemma

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

- ( $\Rightarrow$ ) We need to show that  $w \in L \Rightarrow w \in L(U)$
- Assume  $w \in L$
- This means that  $w \in L(M)$  or  $w \in L(N)$
- By construction,  $U$  nondeterministically correctly chooses to simulate  $M$  or to simulate  $N$
- Thus,  $w \in L(U)$ .
- ( $\Leftarrow$ ) We need to show that  $w \in L(U) \Rightarrow w \in L$
- Assume  $w \in L(U)$
- This means that there is a computation that consumes  $w$  such that:

$$(w \text{ } z'') \vdash^*_{\mathcal{U}} ((\cdot) \text{ } K), \text{ where } K \in F''$$

# FSA and Regular Expressions

## Lemma

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

- ( $\Rightarrow$ ) We need to show that  $w \in L \Rightarrow w \in L(U)$
- Assume  $w \in L$
- This means that  $w \in L(M)$  or  $w \in L(N)$
- By construction,  $U$  nondeterministically correctly chooses to simulate  $M$  or to simulate  $N$
- Thus,  $w \in L(U)$ .
- ( $\Leftarrow$ ) We need to show that  $w \in L(U) \Rightarrow w \in L$
- Assume  $w \in L(U)$
- This means that there is a computation that consumes  $w$  such that:  
$$(w \text{ } z'') \vdash^*_U ((\text{ }) \text{ } K), \text{ where } K \in F^{**}$$
- By  $U$ 's construction, either  $M$  or  $N$  is simulated
- $U$ 's final states are the final states of  $M$  and  $N$
- This means  $w \in L(M)$  or  $w \in L(N)$
- Thus,  $w \in L$

# FSA and Regular Expressions

## Lemma

$$w \notin L \Leftrightarrow w \notin L(U)$$

- ( $\Rightarrow$ ) We need to show that  $w \notin L \Rightarrow w \notin L(U)$

# FSA and Regular Expressions

## Lemma

$$w \notin L \Leftrightarrow w \notin L(U)$$

- ( $\Rightarrow$ ) We need to show that  $w \notin L \Rightarrow w \notin L(U)$
- Assume  $w \notin L$
- This means that  $w \notin L(M)$  and  $w \notin L(N)$
- By U's construction,  $F'' = F \cup F'$
- And all possible computations of U on w never reach a state in  $F''$
- Thus,  $w \notin L(U)$

# FSA and Regular Expressions

## Lemma

$$w \notin L \Leftrightarrow w \notin L(U)$$

- ( $\Rightarrow$ ) We need to show that  $w \notin L \Rightarrow w \notin L(U)$
- Assume  $w \notin L$
- This means that  $w \notin L(M)$  and  $w \notin L(N)$
- By U's construction,  $F'' = F \cup F'$
- And all possible computations of U on w never reach a state in  $F''$
- Thus,  $w \notin L(U)$
- ( $\Leftarrow$ ) We need to show that  $w \notin L(U) \Rightarrow w \notin L$ .

# FSA and Regular Expressions

## Lemma

$$w \notin L \Leftrightarrow w \notin L(U)$$

- ( $\Rightarrow$ ) We need to show that  $w \notin L \Rightarrow w \notin L(U)$
- Assume  $w \notin L$
- This means that  $w \notin L(M)$  and  $w \notin L(N)$
- By U's construction,  $F'' = F \cup F'$
- And all possible computations of U on w never reach a state in  $F''$
- Thus,  $w \notin L(U)$
- ( $\Leftarrow$ ) We need to show that  $w \notin L(U) \Rightarrow w \notin L$ .
- Assume  $w \notin L(U)$
- This means that all possible computations on w are described as follows:  
 $(w \; Z'') \vdash^*_U ((\cdot) \; K)$ , where  $K \notin F''$
- By U's construction, either M or N is simulated
- $F'' = F \cup F'$
- This means  $w \notin L(M)$  and  $w \notin L(N)$
- Thus,  $w \notin L$

# FSA and Regular Expressions

Regular  
Expressions

Deterministic  
Finite  
Automata

Nondeterministic  
Finite  
Automata

Finite-State  
Automata and  
Regular  
Expressions

Regular  
Grammars

Pumping  
Theorem for  
Regular  
Languages

## Theorem

*The languages accepted by finite-state machines are closed under union.*

- The theorem is established by the previous two lemmas

# FSA and Regular Expressions

- HOMEWORK: 1, 3, 4

# FSA and Regular Expressions

## Theorem

*The languages accepted by finite-state machines are closed concatenation.*

- Let the following be the two machines that decide the languages to concatenate:

$$\begin{aligned} M &= (\text{make-ndfa } S_M \ \Sigma_M \ A \ F_M \ \delta_M) \\ N &= (\text{make-ndfa } S_N \ \Sigma_N \ R \ F_N \ \delta_N) \end{aligned}$$

# FSA and Regular Expressions

## Theorem

*The languages accepted by finite-state machines are closed concatenation.*

- Let the following be the two machines that decide the languages to concatenate:

$$\begin{aligned} M &= (\text{make-ndfa } S_M \ \Sigma_M \ A \ F_M \ \delta_M) \\ N &= (\text{make-ndfa } S_N \ \Sigma_N \ R \ F_N \ \delta_N) \end{aligned}$$

- We need to construct an ndfa that decides  $L = L(M) \circ L(N)$

# FSA and Regular Expressions

## Theorem

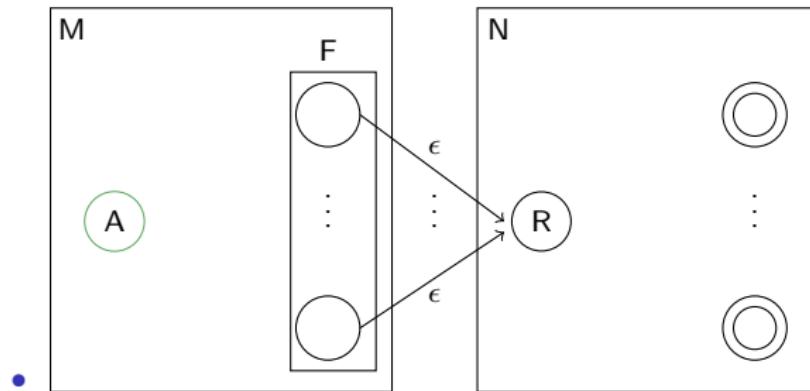
*The languages accepted by finite-state machines are closed concatenation.*

- Let the following be the two machines that decide the languages to concatenate:

$$M = (\text{make-ndfa } S_M \ \Sigma_M \ A \ F_M \ \delta_M)$$

$$N = (\text{make-ndfa } S_N \ \Sigma_N \ R \ F_N \ \delta_N)$$

- We need to construct an ndfa that decides  $L = L(M) \circ L(N)$



# FSA and Regular Expressions

- `;; ndfa ndfa → ndfa`  
;; Purpose: Construct ndfa for the concatenation of the languages  
;; given ndfas  
;; Assume: The intersection of the states is empty  
`(define (concat-fsa M N)`

# FSA and Regular Expressions

- ```
;; ndfa ndfa → ndfa
;; Purpose: Construct ndfa for the concatenation of the languages
;;           given ndfas
;; Assume: The intersection of the states is empty
(define (concat-fsa M N)
```
- ```
;; Tests for concat-fsa
(define ab*-o-a-aUb-b* (concat-fsa ab* a-aUb-b*))
(define ab*-o-aab* (concat-fsa ab* aab*))
```

# FSA and Regular Expressions

- ```
;; ndfa ndfa → ndfa
;; Purpose: Construct ndfa for the concatenation of the languages
;;           given ndfas
;; Assume: The intersection of the states is empty
(define (concat-fsa M N)
```
- ```
;; Tests for concat-fsa
(define ab*-o-a-aUb-b* (concat-fsa ab* a-aUb-b*))
(define ab*-o-aab* (concat-fsa ab* aab*))

;; Tests for concat-fsa
(define ab*-o-a-aUb-b* (concat-fsa ab* a-aUb-b*))
(define ab*-o-aab* (concat-fsa ab* aab*))

;; Tests for concat-fsa
(define ab*-o-a-aUb-b* (concat-fsa ab* a-aUb-b*))
(define ab*-o-aab* (concat-fsa ab* aab*))
```
- ```
(check-equal? (sm-apply ab*-o-a-aUb-b* '()) 'reject)
(check-equal? (sm-apply ab*-o-a-aUb-b* '(b b b)) 'reject)
(check-equal? (sm-apply ab*-o-a-aUb-b* '(a a b a b)) 'reject)
(check-equal? (sm-apply ab*-o-a-aUb-b* '(a b a a b)) 'accept)
(check-equal? (sm-apply ab*-o-a-aUb-b* '(a b b b a a)) 'accept)
(check-equal? (sm-testequiv? ab*-o-a-aUb-b* (sm-concat ab* a-aUb-b*))
  (check-equal? (sm-apply ab*-o-aab* '()) 'reject)
  (check-equal? (sm-apply ab*-o-aab* '(a b a)) 'reject))
```

# FSA and Regular Expressions

- ```
;; ndfa ndfa → ndfa
;; Purpose: Construct ndfa for the concatenation of the languages
;;           given ndfas
;; Assume: The intersection of the states is empty
(define (concat-fsa M N))
```
- ```
(let* [(new-start (sm-start M))
```
- ```
; Tests for concat-fsa
(define ab*-o-a-aUb-b* (concat-fsa ab* a-aUb-b*))
(define ab*-o-aab* (concat-fsa ab* aab*))

• (check-equal? (sm-apply ab*-o-a-aUb-b* '()) 'reject)
  (check-equal? (sm-apply ab*-o-a-aUb-b* '(b b b)) 'reject)
  (check-equal? (sm-apply ab*-o-a-aUb-b* '(a a b a b)) 'reject)
  (check-equal? (sm-apply ab*-o-a-aUb-b* '(a b a a b)) 'accept)
  (check-equal? (sm-apply ab*-o-a-aUb-b* '(a b b b a a)) 'accept)
  (check-equal? (sm-testequiv? ab*-o-a-aUb-b* (sm-concat ab* a-aUb-b*))
  (check-equal? (sm-apply ab*-o-aab* '()) 'reject)
  (check-equal? (sm-apply ab*-o-aab* '(a b a)) 'reject)
```

# FSA and Regular Expressions

- `;; ndfa ndfa → ndfa`

`;; Purpose: Construct ndfa for the concatenation of the languages given ndfas`

`;; Assume: The intersection of the states is empty`

`(define (concat-fsa M N)`

- `(let* [(new-start (sm-start M))`

- `(new-sigma (remove-duplicates (append (sm-sigma M) (sm-sigma N))))`

- `;; Tests for concat-fsa`

`(define ab*-o-a-aUb-b* (concat-fsa ab* a-aUb-b*))`

`(define ab*-o-aab* (concat-fsa ab* aab*))`

- `(check-equal? (sm-apply ab*-o-a-aUb-b* '()) 'reject)`

`(check-equal? (sm-apply ab*-o-a-aUb-b* '(b b b)) 'reject)`

`(check-equal? (sm-apply ab*-o-a-aUb-b* '(a a b a b)) 'reject)`

`(check-equal? (sm-apply ab*-o-a-aUb-b* '(a b a a b)) 'accept)`

`(check-equal? (sm-apply ab*-o-a-aUb-b* '(a b b b a a)) 'accept)`

`(check-equal? (sm-testequiv? ab*-o-a-aUb-b* (sm-concat ab* a-aUb-b*)) #t)`

`(check-equal? (sm-apply ab*-o-aab* '()) 'reject)`

`(check-equal? (sm-apply ab*-o-aab* '(a b a)) 'reject)`

# FSA and Regular Expressions

- `;; ndfa ndfa → ndfa`  
`;; Purpose: Construct ndfa for the concatenation of the languages`  
`;; given ndfas`  
`;; Assume: The intersection of the states is empty`  
`(define (concat-fsa M N)`
  - `(let* [(new-start (sm-start M))`
  - `(new-sigma (remove-duplicates (append (sm-sigma M) (sm-sigma N))))`
  - `(new-states (append (sm-states M) (sm-states N))))`
- `;; Tests for concat-fsa`  
`(define ab*-o-a-aUb-b* (concat-fsa ab* a-aUb-b*))`  
`(define ab*-o-aab* (concat-fsa ab* aab*))`
  - `(check-equal? (sm-apply ab*-o-a-aUb-b* '()) 'reject)`  
`(check-equal? (sm-apply ab*-o-a-aUb-b* '(b b b)) 'reject)`  
`(check-equal? (sm-apply ab*-o-a-aUb-b* '(a a b a b)) 'reject)`  
`(check-equal? (sm-apply ab*-o-a-aUb-b* '(a b a a b)) 'accept)`  
`(check-equal? (sm-apply ab*-o-a-aUb-b* '(a b b b a a)) 'accept)`  
`(check-equal? (sm-testequiv? ab*-o-a-aUb-b* (sm-concat ab* a-aUb-b*))`  
`(check-equal? (sm-apply ab*-o-aab* '()) 'reject)`  
`(check-equal? (sm-apply ab*-o-aab* '(a b a)) 'reject)`

# FSA and Regular Expressions

- `;; ndfa ndfa → ndfa`  
`;; Purpose: Construct ndfa for the concatenation of the languages`  
`;; given ndfas`  
`;; Assume: The intersection of the states is empty`  
`(define (concat-fsa M N)`
  - `(let* [(new-start (sm-start M))`
  - `(new-sigma (remove-duplicates (append (sm-sigma M) (sm-sigma N))))`
  - `(new-states (append (sm-states M) (sm-states N)))`
  - `(new-finals (sm-finals N))`
- `;; Tests for concat-fsa`  
`(define ab*-o-a-aUb-b* (concat-fsa ab* a-aUb-b*))`  
`(define ab*-o-aab* (concat-fsa ab* aab*))`
  - `(check-equal? (sm-apply ab*-o-a-aUb-b* '()) 'reject)`  
`(check-equal? (sm-apply ab*-o-a-aUb-b* '(b b b)) 'reject)`  
`(check-equal? (sm-apply ab*-o-a-aUb-b* '(a a b a b)) 'reject)`  
`(check-equal? (sm-apply ab*-o-a-aUb-b* '(a b a a b)) 'accept)`  
`(check-equal? (sm-apply ab*-o-a-aUb-b* '(a b b b a a)) 'accept)`  
`(check-equal? (sm-testequiv? ab*-o-a-aUb-b* (sm-concat ab* a-aUb-b*))`  
`(check-equal? (sm-apply ab*-o-aab* '()) 'reject)`  
`(check-equal? (sm-apply ab*-o-aab* '(a b a)) 'reject)`

# FSA and Regular Expressions

- `;; ndfa ndfa → ndfa`  
    `;; Purpose: Construct ndfa for the concatenation of the languages`  
    `;; given ndfas`  
    `;; Assume: The intersection of the states is empty`  
`(define (concat-fsa M N)`
  - `(let* [(new-start (sm-start M))`
  - `(new-sigma (remove-duplicates (append (sm-sigma M) (sm-sigma N))))`
  - `(new-states (append (sm-states M) (sm-states N)))`
  - `(new-finals (sm-finals M))`
  - `(new-rules (append (sm-rules M)`  
                        `(sm-rules N))`  
                        `(map (λ (f) (list f EMP (sm-start N)))`  
                        `(sm-finals M))))]`  
- `;; Tests for concat-fsa`  
`(define ab*-o-a-aUb-b* (concat-fsa ab* a-aUb-b*))`  
`(define ab*-o-aab* (concat-fsa ab* aab*))`
  - `(check-equal? (sm-apply ab*-o-a-aUb-b* '()) 'reject)`  
`(check-equal? (sm-apply ab*-o-a-aUb-b* '(b b b)) 'reject)`  
`(check-equal? (sm-apply ab*-o-a-aUb-b* '(a a b a b)) 'reject)`  
`(check-equal? (sm-apply ab*-o-a-aUb-b* '(a b a a b)) 'accept)`  
`(check-equal? (sm-apply ab*-o-a-aUb-b* '(a b b b a a)) 'accept)`  
`(check-equal? (sm-testequiv? ab*-o-a-aUb-b* (sm-concat ab* a-aUb-b*))`  
`(check-equal? (sm-apply ab*-o-aab* '()) 'reject)`  
`(check-equal? (sm-apply ab*-o-aab* '(a b a)) 'reject)`

# FSA and Regular Expressions

- `;; ndfa ndfa → ndfa`  
    `;; Purpose: Construct ndfa for the concatenation of the languages`  
    `;; given ndfas`  
    `;; Assume: The intersection of the states is empty`  
`(define (concat-fsa M N)`
- `(let* [(new-start (sm-start M))`
- `(new-sigma (remove-duplicates (append (sm-sigma M) (sm-sigma N))))`
- `(new-states (append (sm-states M) (sm-states N))))`
- `(new-finals (sm-finals M))`
- `(new-rules (append (sm-rules M)`  
                 `(sm-rules N))`  
                 `(map (λ (f) (list f EMP (sm-start N)))`  
                 `(sm-finals M))))]`
- `(make-ndfa new-states new-sigma new-start new-finals new-rules)`
- `; ; Tests for concat-fsa`  
`(define ab*-o-a-aUb-b* (concat-fsa ab* a-aUb-b*))`  
`(define ab*-o-aab* (concat-fsa ab* aab*))`
- `(check-equal? (sm-apply ab*-o-a-aUb-b* '()) 'reject)`  
`(check-equal? (sm-apply ab*-o-a-aUb-b* '(b b b)) 'reject)`  
`(check-equal? (sm-apply ab*-o-a-aUb-b* '(a a b a b)) 'reject)`  
`(check-equal? (sm-apply ab*-o-a-aUb-b* '(a b a a b)) 'accept)`  
`(check-equal? (sm-apply ab*-o-a-aUb-b* '(a b b b a a)) 'accept)`  
`(check-equal? (sm-testequiv? ab*-o-a-aUb-b* (sm-concat ab* a-aUb-b*))`  
`(check-equal? (sm-apply ab*-o-aab* '()) 'reject)`  
`(check-equal? (sm-apply ab*-o-aab* '(a b a)) 'reject)`

# FSA and Regular Expressions

- Define three machines as follows:

$$M = (\text{make-ndfa } S \ \Sigma \ Z \ F \ \delta)$$
$$N = (\text{make-ndfa } S' \ \Sigma' \ Z' \ F' \ \delta')$$
$$U = (\text{concat-fsa } M \ N) = (\text{make-ndfa } S'' \ \Sigma'' \ Z'' \ F'' \ \delta'')$$

- Let  $L = L(M) \circ L(N)$
- We proceed to prove that  $L = L(U)$

# FSA and Regular Expressions

## Lemma

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

- ( $\Rightarrow$ ) We need to show that  $w \in L \Rightarrow w \in L(U)$

# FSA and Regular Expressions

## Lemma

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

- ( $\Rightarrow$ ) We need to show that  $w \in L \Rightarrow w \in L(U)$
- Assume  $w \in L$
- This means  $w = xy$ , where  $x \in L(M)$  and  $y \in L(N)$
- By construction of  $U$ , the following is a valid computation:  
 $(xy \text{ } S'') \vdash^*_{U'} (y \text{ } R) \vdash (y \text{ } z') \vdash^*_{U'} ((\text{ }) \text{ } T)$ , where  $R \in F$  and  $T \in F'$
- By construction of  $U$ ,  $F'' = F'$
- Therefore,  $w \in L(U)$

# FSA and Regular Expressions

## Lemma

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

- ( $\Rightarrow$ ) We need to show that  $w \in L \Rightarrow w \in L(U)$
- Assume  $w \in L$
- This means  $w = xy$ , where  $x \in L(M)$  and  $y \in L(N)$
- By construction of  $U$ , the following is a valid computation:  
 $(xy \ S'') \vdash^*_{U'} (y \ R) \vdash (y \ z') \vdash^*_{U'} (( \ T)$ , where  $R \in F$  and  $T \in F'$
- By construction of  $U$ ,  $F'' = F'$
- Therefore,  $w \in L(U)$
- ( $\Leftarrow$ ) We need to show that  $w \in L(U) \Rightarrow w \in L$

# FSA and Regular Expressions

## Lemma

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

- ( $\Rightarrow$ ) We need to show that  $w \in L \Rightarrow w \in L(U)$
- Assume  $w \in L$
- This means  $w = xy$ , where  $x \in L(M)$  and  $y \in L(N)$
- By construction of  $U$ , the following is a valid computation:  
 $(xy \text{ } S'') \vdash^*_{U'} (y \text{ } R) \vdash (y \text{ } Z') \vdash^*_{U'} ((\text{ }) \text{ } T)$ , where  $R \in F$  and  $T \in F'$
- By construction of  $U$ ,  $F'' = F'$
- Therefore,  $w \in L(U)$
- ( $\Leftarrow$ ) We need to show that  $w \in L(U) \Rightarrow w \in L$
- Assume  $w \in L(U)$
- By construction of  $U$ , this means that for  $w = xy$  the following computation is valid:  
 $(xy \text{ } S'') \vdash^*_{U'} (y \text{ } R) \vdash (y \text{ } Z') \vdash^*_{U'} ((\text{ }) \text{ } T)$ , where  $R \in F$  and  $T \in F'$
- This implies that  $x \in L(M)$  and  $y \in L(N)$
- Thus,  $w \in L$

# FSA and Regular Expressions

Regular  
Expressions

Deterministic  
Finite  
Automata

Nondeterministic  
Finite  
Automata

**Finite-State  
Automata and  
Regular  
Expressions**

Regular  
Grammars

Pumping  
Theorem for  
Regular  
Languages

## Lemma

$$w \notin L \Leftrightarrow w \notin L(U)$$

# FSA and Regular Expressions

## Lemma

$$w \notin L \Leftrightarrow w \notin L(U)$$

- ( $\Rightarrow$ ) We need to show that  $w \notin L \Rightarrow w \notin L(U)$

# FSA and Regular Expressions

## Lemma

$$w \notin L \Leftrightarrow w \notin L(U)$$

- ( $\Rightarrow$ ) We need to show that  $w \notin L \Rightarrow w \notin L(U)$
- Assume  $w \notin L$
- This means  $w \neq xy$ , where  $x \in L(M)$  and  $y \in L(N)$
- By construction, all possible computations of  $U$  on  $w$  either do not reach a final state by consuming  $w$  or do not consume all of  $w$
- In both cases,  $w$  is rejected
- Therefore,  $w \notin L(U)$ .

# FSA and Regular Expressions

## Lemma

$$w \notin L \Leftrightarrow w \notin L(U)$$

- ( $\Rightarrow$ ) We need to show that  $w \notin L \Rightarrow w \notin L(U)$
- Assume  $w \notin L$
- This means  $w \neq xy$ , where  $x \in L(M)$  and  $y \in L(N)$
- By construction, all possible computations of  $U$  on  $w$  either do not reach a final state by consuming  $w$  or do not consume all of  $w$
- In both cases,  $w$  is rejected
- Therefore,  $w \notin L(U)$ .
- ( $\Leftarrow$ ) We need to show that  $w \notin L(U) \Rightarrow w \notin L$

# FSA and Regular Expressions

## Lemma

$$w \notin L \Leftrightarrow w \notin L(U)$$

- ( $\Rightarrow$ ) We need to show that  $w \notin L \Rightarrow w \notin L(U)$
- Assume  $w \notin L$
- This means  $w \neq xy$ , where  $x \in L(M)$  and  $y \in L(N)$
- By construction, all possible computations of  $U$  on  $w$  either do not reach a final state by consuming  $w$  or do not consume all of  $w$
- In both cases,  $w$  is rejected
- Therefore,  $w \notin L(U)$ .
- ( $\Leftarrow$ ) We need to show that  $w \notin L(U) \Rightarrow w \notin L$
- Assume  $w \notin L(U)$
- By construction, this means that  $w$  is rejected because  $U$  consumes  $w$  and does not reach a final state or  $U$  is unable to consume  $w$
- This implies that  $w$  cannot be written as  $xy$ , where  $x \in L(M)$  and  $y \in L(N)$
- Thus,  $w \notin L$

# FSA and Regular Expressions

Regular  
Expressions

Deterministic  
Finite  
Automata

Nondeterministic  
Finite  
Automata

Finite-State  
Automata and  
Regular  
Expressions

Regular  
Grammars

Pumping  
Theorem for  
Regular  
Languages

## Theorem

*The languages accepted by finite-state machines are closed concatenation.*

- The proof follows from the two previous lemmas

# FSA and Regular Expressions

- HOMEWORK: 5, 7, 8

# FSA and Regular Expressions

## Theorem

*The languages accepted by finite-state machines are closed under Kleene star.*

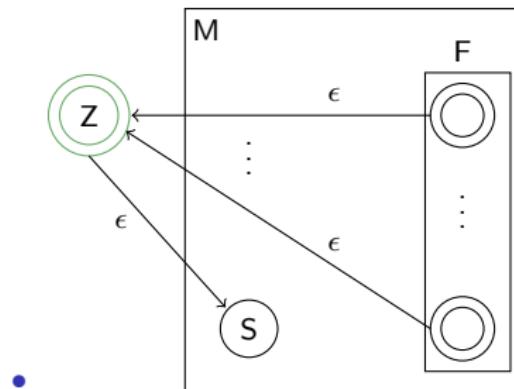
- Let the following be the machine that decides the language to be Kleene starred:  
 $M = (\text{make-ndfa } S \Sigma A F \delta)$
- We need to construct an ndfa that decides  $L = L(M)^*$

# FSA and Regular Expressions

## Theorem

*The languages accepted by finite-state machines are closed under Kleene star.*

- Let the following be the machine that decides the language to be Kleene starred:
- $M = (\text{make-ndfa } S \Sigma A F \delta)$
- We need to construct an ndfa that decides  $L = L(M)^*$



# FSA and Regular Expressions

- ;; ndfa → ndfa  
;; Purpose: Construct ndfa for the Kleene star of given ndfa  
`(define (kstar-fsa M)`

# FSA and Regular Expressions

- ;; ndfa → ndfa  
;; Purpose: Construct ndfa for the Kleene star of given ndfa  
(define (kstar-fsa M)

- ;; Tests for kstar-fsa  
(define a-aUb-b\*\*-\* (kstar-fsa a-aUb-b\*))  
(define ab\*\*-\* (kstar-fsa ab\*))  
  
(check-equal? (sm-apply a-aUb-b\*\*-\* '(b b b)) 'reject)  
(check-equal? (sm-apply a-aUb-b\*\*-\* '(a b a b a a a a)) 'reject)  
(check-equal? (sm-apply a-aUb-b\*\*-\* '()) 'accept)  
(check-equal? (sm-apply a-aUb-b\*\*-\* '(a a a a b b b b)) 'accept)  
(check-equal? (sm-apply a-aUb-b\*\*-\* '(a a b a a b b a a)) 'accept)  
(check-equal? (sm-testequiv? a-aUb-b\*\*-\* (sm-kleenestar a-aUb-b\*)) #  
  
(check-equal? (sm-apply ab\*\*-\* '(b)) 'reject)  
(check-equal? (sm-apply ab\*\*-\* '(b b b)) 'reject)

# FSA and Regular Expressions

- `;; ndfa → ndfa`  
`;; Purpose: Construct ndfa for the Kleene star of given ndfa`  
`(define (kstar-fsa M)`
- `(let* [(new-start (generate-symbol 'K (sm-states M)))`  
`(new-sigma (sm-sigma M))`
- `;; Tests for kstar-fsa`  
`(define a-aUb-b**-* (kstar-fsa a-aUb-b*))`  
`(define ab**-* (kstar-fsa ab*))`  
  
`(check-equal? (sm-apply a-aUb-b**-* '(b b b)) 'reject)`  
`(check-equal? (sm-apply a-aUb-b**-* '(a b a b a a a a)) 'reject)`  
`(check-equal? (sm-apply a-aUb-b**-* '()) 'accept)`  
`(check-equal? (sm-apply a-aUb-b**-* '(a a a a b b b b)) 'accept)`  
`(check-equal? (sm-apply a-aUb-b**-* '(a a b a a b b a a)) 'accept)`  
`(check-equal? (sm-testequiv? a-aUb-b**-* (sm-kleenestar a-aUb-b*)) #t)`  
  
`(check-equal? (sm-apply ab**-* '(b)) 'reject)`  
`(check-equal? (sm-apply ab**-* '(b b b)) 'reject)`

# FSA and Regular Expressions

- `;; ndfa → ndfa`  
`;; Purpose: Construct ndfa for the Kleene star of given ndfa`  
`(define (kstar-fsa M)`
- `(let* [(new-start (generate-symbol 'K (sm-states M)))`  
`(new-sigma (sm-sigma M))`
- `(new-states (cons new-start (sm-states M)))`  
`(new-finals (cons new-start (sm-finals M))))`
  
- `;; Tests for kstar-fsa`  
`(define a-aUb-b**-* (kstar-fsa a-aUb-b*))`  
`(define ab**-* (kstar-fsa ab*))`  
  
`(check-equal? (sm-apply a-aUb-b**-* '(b b b)) 'reject)`  
`(check-equal? (sm-apply a-aUb-b**-* '(a b a b a a a a)) 'reject)`  
`(check-equal? (sm-apply a-aUb-b**-* '()) 'accept)`  
`(check-equal? (sm-apply a-aUb-b**-* '(a a a a b b b b)) 'accept)`  
`(check-equal? (sm-apply a-aUb-b**-* '(a a b a a b b a a)) 'accept)`  
`(check-equal? (sm-testequiv? a-aUb-b**-* (sm-kleenestar a-aUb-b*)) #t)`  
  
`(check-equal? (sm-apply ab**-* '(b)) 'reject)`  
`(check-equal? (sm-apply ab**-* '(b b b)) 'reject)`

# FSA and Regular Expressions

- `;; ndfa → ndfa`  
`;; Purpose: Construct ndfa for the Kleene star of given ndfa`  
`(define (kstar-fsa M)`
- `(let* [(new-start (generate-symbol 'K (sm-states M)))`  
`(new-sigma (sm-sigma M))`
- `(new-states (cons new-start (sm-states M)))`  
`(new-finals (cons new-start (sm-finals M)))`
- `(new-rules (cons (list new-start EMP (sm-start M))`  
`append (sm-rules M)`  
`(map (λ (f) (list f EMP new-start`  
`(sm-finals M)))))]`
- `;; Tests for kstar-fsa`  
`(define a-aUb-b**-* (kstar-fsa a-aUb-b*))`  
`(define ab**-* (kstar-fsa ab*))`  
  
`(check-equal? (sm-apply a-aUb-b**-* '(b b b)) 'reject)`  
`(check-equal? (sm-apply a-aUb-b**-* '(a b a b a a a a)) 'reject)`  
`(check-equal? (sm-apply a-aUb-b**-* '()) 'accept)`  
`(check-equal? (sm-apply a-aUb-b**-* '(a a a a b b b b)) 'accept)`  
`(check-equal? (sm-apply a-aUb-b**-* '(a a b a a b b a a)) 'accept)`  
`(check-equal? (sm-testequiv? a-aUb-b**-* (sm-kleenestar a-aUb-b*)) #t)`  
  
`(check-equal? (sm-apply ab**-* '(b)) 'reject)`  
`(check-equal? (sm-apply ab**-* '(b b b)) 'reject)`

# FSA and Regular Expressions

- `;; ndfa → ndfa`  
`;; Purpose: Construct ndfa for the Kleene star of given ndfa`  
`(define (kstar-fsa M)`
- `(let* [(new-start (generate-symbol 'K (sm-states M)))`  
`(new-sigma (sm-sigma M))`
- `(new-states (cons new-start (sm-states M)))`  
`(new-finals (cons new-start (sm-finals M)))`
- `(new-rules (cons (list new-start EMP (sm-start M))`  
`append (sm-rules M)`  
`(map (λ (f) (list f EMP new-start`  
`(sm-finals M)))))]`
- `(make-ndfa new-states new-sigma new-start new-finals new-rules))`
- `; ; Tests for kstar-fsa`  
`(define a-aUb-b**-* (kstar-fsa a-aUb-b*))`  
`(define ab**-* (kstar-fsa ab*))`  
  
`(check-equal? (sm-apply a-aUb-b**-* '(b b b)) 'reject)`  
`(check-equal? (sm-apply a-aUb-b**-* '(a b a b a a a a)) 'reject)`  
`(check-equal? (sm-apply a-aUb-b**-* '()) 'accept)`  
`(check-equal? (sm-apply a-aUb-b**-* '(a a a a b b b b)) 'accept)`  
`(check-equal? (sm-apply a-aUb-b**-* '(a a b a a b b a a)) 'accept)`  
`(check-equal? (sm-testequiv? a-aUb-b**-* (sm-kleenestar a-aUb-b*)) #t)`  
  
`(check-equal? (sm-apply ab**-* '(b)) 'reject)`  
`(check-equal? (sm-apply ab**-* '(b b b)) 'reject)`

# FSA and Regular Expressions

- Define two machines as follows:

$M = (\text{make-ndfa } S \Sigma Z F \delta)$

$U = (\text{kstar-fsa } M) = (\text{make-ndfa } S' \Sigma' Z' F' \delta')$

# FSA and Regular Expressions

- Define two machines as follows:

$$M = (\text{make-ndfa } S \ \Sigma \ Z \ F \ \delta)$$
$$U = (\text{kstar-fsa } M) = (\text{make-ndfa } S' \ \Sigma' \ Z' \ F' \ \delta')$$

- Let  $L = L(M)^*$
- We proceed to prove that  $L = L(U)$

# FSA and Regular Expressions

## Lemma

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

- ( $\Rightarrow$ ) We need to show that  $w \in L \Rightarrow w \in L(U)$

# FSA and Regular Expressions

## Lemma

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

- ( $\Rightarrow$ ) We need to show that  $w \in L \Rightarrow w \in L(U)$
- Assume  $w \in L$ . This means that  $w = w_1 w_2 \dots w_n$ , where  $w_i \in L(M)$
- By construction of  $U$ , the following is a computation on  $w$ :  
 $((w_1 w_2 \dots w_n \ S') \vdash^* \$ \ U \$ ((w_2 \dots w_n) \ Y_1) \vdash^* _U ((\dots w_n) \ Y_2) \vdash^* _U ((\dots w_n) \ Y_n),$  where  $Y_i \in F'$
- Therefore,  $w \in L(U)$

# FSA and Regular Expressions

## Lemma

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

- ( $\Rightarrow$ ) We need to show that  $w \in L \Rightarrow w \in L(U)$
- Assume  $w \in L$ . This means that  $w = w_1 w_2 \dots w_n$ , where  $w_i \in L(M)$
- By construction of  $U$ , the following is a computation on  $w$ :  
 $((w_1 w_2 \dots w_n \ S') \vdash^* \$ \ U \$ ((w_2 \dots w_n) \ Y_1) \vdash^* _U ((\dots w_n) \ Y_2) \vdash^* _U (\dots Y_n),$  where  $Y_i \in F'$
- Therefore,  $w \in L(U)$
- ( $\Leftarrow$ ) We need to show that  $w \in L(U) \Rightarrow w \in L$

# FSA and Regular Expressions

## Lemma

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

- ( $\Rightarrow$ ) We need to show that  $w \in L \Rightarrow w \in L(U)$
- Assume  $w \in L$ . This means that  $w = w_1 w_2 \dots w_n$ , where  $w_i \in L(M)$
- By construction of  $U$ , the following is a computation on  $w$ :  
 $((w_1 w_2 \dots w_n \ S') \vdash^* \$ \ U \$ ((w_2 \dots w_n) \ Y_1) \vdash^* _U ((\dots w_n) \ Y_2) \vdash^* _U (\dots) \ Y_n),$  where  $Y_i \in F'$
- Therefore,  $w \in L(U)$
- ( $\Leftarrow$ ) We need to show that  $w \in L(U) \Rightarrow w \in L$
- Assume  $w \in L(U)$
- This means that  $w = w_1 w_2 \dots w_n$  such that:  
 $((w_1 \dots w_n) \ S') \vdash^* _U ((w_2 \dots w_n) \ Y_1) \vdash^* _U ((w_3 \dots w_n) \ Y_2) \dots ((w_n) \ Y_{n-1}) \vdash^* _U (\dots) \ Y_n),$  where  $Y_i \in F'$
- By construction of  $U$ ,  $F'$  contains  $S'$  and  $F$
- This means that  $w$  is the concatenation of zero or more words in  $L(M)$
- Thus,  $w \in L$

# FSA and Regular Expressions

## Lemma

$$w \notin L \Leftrightarrow w \notin L(U)$$

- ( $\Rightarrow$ )
- We need to show that  $w \notin L \Rightarrow w \notin L(U)$

# FSA and Regular Expressions

## Lemma

$$w \notin L \Leftrightarrow w \notin L(U)$$

- ( $\Rightarrow$ )
- We need to show that  $w \notin L \Rightarrow w \notin L(U)$
- Assume  $w \notin L$
- This means that  $w \neq w_1 w_2 \dots w_n$ , where  $w_i \in L(M)$
- By construction of  $U$ , the processing of  $w$  can occur in two ways
- The first,  $w$  is consumed and  $U$  does not end in a final state
- The second,  $w$  cannot be completely consumed
- In both cases,  $w$  is rejected
- Thus,  $w \notin L(U)$ .

# FSA and Regular Expressions

## Lemma

$$w \notin L \Leftrightarrow w \notin L(U)$$

- ( $\Rightarrow$ )
- We need to show that  $w \notin L \Rightarrow w \notin L(U)$
- Assume  $w \notin L$
- This means that  $w \neq w_1 w_2 \dots w_n$ , where  $w_i \in L(M)$
- By construction of  $U$ , the processing of  $w$  can occur in two ways
- The first,  $w$  is consumed and  $U$  does not end in a final state
- The second,  $w$  cannot be completely consumed
- In both cases,  $w$  is rejected
- Thus,  $w \notin L(U)$ .
- ( $\Leftarrow$ ) We need to show that  $w \notin L(U) \Rightarrow w \notin L$

# FSA and Regular Expressions

## Lemma

$$w \notin L \Leftrightarrow w \notin L(U)$$

- ( $\Rightarrow$ )
  - We need to show that  $w \notin L \Rightarrow w \notin L(U)$
  - Assume  $w \in L$
  - This means that  $w = w_1 w_2 \dots w_n$ , where  $w_i \in L(M)$
  - By construction of  $U$ , the processing of  $w$  can occur in two ways
    - The first,  $w$  is consumed and  $U$  does not end in a final state
    - The second,  $w$  cannot be completely consumed
  - In both cases,  $w$  is rejected
  - Thus,  $w \notin L(U)$ .
- ( $\Leftarrow$ ) We need to show that  $w \notin L(U) \Rightarrow w \notin L$ 
  - Assume  $w \notin L(U)$
  - This means that  $w \neq w_1 w_2 \dots w_n$  such that:  
 $((w_1 \dots w_n) \ S') \vdash^*_{\cal U} ((w_2 \dots w_n) \ Y_1) \vdash^*_{\cal U} ((w_3 \dots w_{n-1}) \ Y_2) \dots ((w_n) \ Y_{n-1}) \vdash^*_{\cal U} (\ ) \ Y_n$ , where  $Y_i \in F'$
  - By construction of  $U$ ,  $F'$  contains  $S'$  and  $F$
  - This means that  $w$  is not the concatenation of zero or more words in  $L(M)$
  - Thus,  $w \notin L$

# FSA and Regular Expressions

Regular  
Expressions

Deterministic  
Finite  
Automata

Nondeterministic  
Finite  
Automata

Finite-State  
Automata and  
Regular  
Expressions

Regular  
Grammars

Pumping  
Theorem for  
Regular  
Languages

## Theorem

*The languages accepted by finite-state machines are closed under Kleene star.*

- The proof follows from the previous two lemmas

# FSA and Regular Expressions

- HOMEWORK: 10, 11
- QUIZ: 9

# FSA and Regular Expressions

## Theorem

*The languages accepted by finite-state machines are closed under complement.*

- Let  $M$  be a dfa. The complement of  $L(M)$  is defined as follows:

$$\bar{L}(M) = \{w \mid w \notin L(M)\}$$

# FSA and Regular Expressions

## Theorem

*The languages accepted by finite-state machines are closed under complement.*

- Let  $M$  be a dfa. The complement of  $L(M)$  is defined as follows:

$$\bar{L}(M) = \{w \mid w \notin L(M)\}$$

- The complement of  $L(M)$  is the language that contains all words not in  $L(M)$

# FSA and Regular Expressions

## Theorem

*The languages accepted by finite-state machines are closed under complement.*

- Let  $M$  be a dfa. The complement of  $L(M)$  is defined as follows:  
$$\bar{L}(M) = \{w \mid w \notin L(M)\}$$
- The complement of  $L(M)$  is the language that contains all words not in  $L(M)$
- Given that  $M$  is a dfa, this suggests inverting the roles of  $M$ 's states

# FSA and Regular Expressions

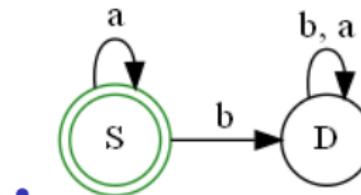
## Theorem

*The languages accepted by finite-state machines are closed under complement.*

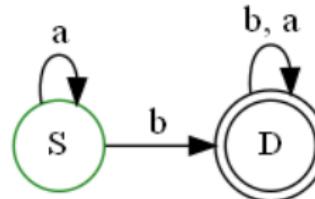
- Let  $M$  be a dfa. The complement of  $L(M)$  is defined as follows:

$$\bar{L}(M) = \{w \mid w \notin L(M)\}$$

- The complement of  $L(M)$  is the language that contains all words not in  $L(M)$
- Given that  $M$  is a dfa, this suggests inverting the roles of  $M$ 's states



- Reversing the roles of the states yields:



# FSA and Regular Expressions

- ;; dfa → dfa

;; Purpose: Construct a dfa for the complement of given dfa's language  
`(define (complement-fsa M)`

# FSA and Regular Expressions

- ;; dfa → dfa

;; Purpose: Construct a dfa for the complement of given dfa's language  
(define (complement-fsa M)

- ;; Tests for complement-fsa

```
(define not-a* (complement-fsa a*))  
(define not-EVEN-A-ODD-B (complement-fsa EVEN-A-ODD-B))  
  
(check-equal? (sm-apply not-a* '()) 'reject)  
(check-equal? (sm-apply not-a* '(a a a)) 'reject)  
(check-equal? (sm-apply not-a* '(a a b)) 'accept)  
(check-equal? (sm-apply not-a* '(b b a a b)) 'accept)  
(check-equal? (sm-testequiv? not-a* (sm-complement a*)) #t)  
  
(check-equal? (sm-apply not-EVEN-A-ODD-B '(b)) 'reject)  
(check-equal? (sm-apply not-EVEN-A-ODD-B '(a a b)) 'reject)  
(check-equal? (sm-apply not-EVEN-A-ODD-B '(b b a b a)) 'reject)  
(check-equal? (sm-apply not-EVEN-A-ODD-B '()) 'accept) ▶ ⏪ ⏴ ⏵ ⏵ ⏵
```

# FSA and Regular Expressions

- `;; dfa → dfa`  
`;; Purpose: Construct a dfa for the complement of given dfa's language`  
`(define (complement-fsa M)`
  - `(let* [(new-finals (filter (λ (s) (not (member s (sm-finals M))))))  
 (sm-states M))])`
- 
- `;; Tests for complement-fsa`  
`(define not-a* (complement-fsa a*))`  
`(define not-EVEN-A-ODD-B (complement-fsa EVEN-A-ODD-B))`  
  
`(check-equal? (sm-apply not-a* '()) 'reject)`  
`(check-equal? (sm-apply not-a* '(a a a)) 'reject)`  
`(check-equal? (sm-apply not-a* '(a a b)) 'accept)`  
`(check-equal? (sm-apply not-a* '(b b a a b)) 'accept)`  
`(check-equal? (sm-testequiv? not-a* (sm-complement a*)) #t)`  
  
`(check-equal? (sm-apply not-EVEN-A-ODD-B '(b)) 'reject)`  
`(check-equal? (sm-apply not-EVEN-A-ODD-B '(a a b)) 'reject)`  
`(check-equal? (sm-apply not-EVEN-A-ODD-B '(b b a b a)) 'reject)`  
`(check-equal? (sm-apply not-EVEN-A-ODD-B '()) 'accept)`

# FSA and Regular Expressions

- `;; dfa → dfa`  
`;; Purpose: Construct a dfa for the complement of given dfa's language`  
`(define (complement-fsa M)`
- `(let* [(new-finals (filter (λ (s) (not (member s (sm-finals M))))))  
 (sm-states M)))]`
- `(make-dfa (sm-states M)  
 (sm-sigma M)  
 (sm-start M)  
 new-finals  
 (sm-rules M)  
 'no-dead)))`
- `;; Tests for complement-fsa`  
`(define not-a* (complement-fsa a*))`  
`(define not-EVEN-A-ODD-B (complement-fsa EVEN-A-ODD-B))`  
  
`(check-equal? (sm-apply not-a* '()) 'reject)`  
`(check-equal? (sm-apply not-a* '(a a a)) 'reject)`  
`(check-equal? (sm-apply not-a* '(a a b)) 'accept)`  
`(check-equal? (sm-apply not-a* '(b b a a b)) 'accept)`  
`(check-equal? (sm-testequiv? not-a* (sm-complement a*)) #t)`  
  
`(check-equal? (sm-apply not-EVEN-A-ODD-B '(b)) 'reject)`  
`(check-equal? (sm-apply not-EVEN-A-ODD-B '(a a b)) 'reject)`  
`(check-equal? (sm-apply not-EVEN-A-ODD-B '(b b a b a)) 'reject)`  
`(check-equal? (sm-apply not-EVEN-A-ODD-B '()) 'accept)`

# FSA and Regular Expressions

- Define two machines as follows:

$M = (\text{make-dfa } S \Sigma Z F \delta)$

$U = (\text{complement-fsa } M) = (\text{make-ndfa } S \Sigma Z F' \delta)$

- Let  $L = \bar{L}(M)$
- We proceed to prove that  $L = L(U)$

# FSA and Regular Expressions

## Lemma

$$w \in L \Leftrightarrow w \notin L(U)$$

- ( $\Rightarrow$ )
- We need to show that  $w \in L \Rightarrow w \notin L(U)$

# FSA and Regular Expressions

## Lemma

$$w \in L \Leftrightarrow w \notin L(U)$$

- ( $\Rightarrow$ )
- We need to show that  $w \in L \Rightarrow w \notin L(U)$
- Assume  $w \in L$
- Given that  $M$  is a dfa, the following is the computation performed on  $w$ :  
 $((w) S) \vdash^*_M ((\cdot) Q)$ , where  $Q \in F$
- By construction of  $U$ ,  $Q \notin F'$  and  $M$  performs the same computation moving from  $S$  to  $Q$  by consuming  $w$
- Therefore,  $w \notin L(U)$

# FSA and Regular Expressions

## Lemma

$$w \in L \Leftrightarrow w \notin L(U)$$

- ( $\Rightarrow$ )
- We need to show that  $w \in L \Rightarrow w \notin L(U)$
- Assume  $w \in L$
- Given that  $M$  is a dfa, the following is the computation performed on  $w$ :  
 $((w) S) \vdash^*_M ((\cdot) Q)$ , where  $Q \in F$
- By construction of  $U$ ,  $Q \notin F'$  and  $M$  performs the same computation moving from  $S$  to  $Q$  by consuming  $w$
- Therefore,  $w \notin L(U)$
- ( $\Leftarrow$ ) We need to show that  $w \notin L(U) \Rightarrow w \in L$

# FSA and Regular Expressions

## Lemma

$$w \in L \Leftrightarrow w \notin L(U)$$

- ( $\Rightarrow$ )
  - We need to show that  $w \in L \Rightarrow w \notin L(U)$
  - Assume  $w \in L$
  - Given that  $M$  is a dfa, the following is the computation performed on  $w$ :  
 $((w) S) \vdash_M^* ((Q), \text{ where } Q \in F)$
  - By construction of  $U$ ,  $Q \notin F'$  and  $M$  performs the same computation moving from  $S$  to  $Q$  by consuming  $w$
  - Therefore,  $w \notin L(U)$
- ( $\Leftarrow$ ) We need to show that  $w \notin L(U) \Rightarrow w \in L$ 
  - Assume  $w \notin L(U)$
  - This means that  $U$  performs the following computation on  $w$ :  
 $((w) S) \vdash_U^* ((Q), \text{ where } Q \notin F')$
  - By construction of  $U$ ,  $Q \in F$  and  $M$  performs the same computation moving from  $S$  to  $Q$  by consuming  $w$
  - Thus,  $w \in L$

# FSA and Regular Expressions

## Lemma

$$w \notin L \Leftrightarrow w \in L(U)$$

- ( $\Rightarrow$ ) We need to show that  $w \notin L \Rightarrow w \in L(U)$

# FSA and Regular Expressions

## Lemma

$$w \notin L \Leftrightarrow w \in L(U)$$

- ( $\Rightarrow$ ) We need to show that  $w \notin L \Rightarrow w \in L(U)$
- Assume  $w \notin L$
- Given that  $M$  is a dfa, the following is the computation it performs on  $w$ :  
 $((w) S) \vdash^*_M ((Q) Q)$ , where  $Q \notin F$
- By construction of  $U$ ,  $Q \in F'$  and  $U$  performs the same computation moving from  $S$  to  $Q$  by consuming  $w$
- Therefore,  $w \in L(U)$ .

# FSA and Regular Expressions

## Lemma

$$w \notin L \Leftrightarrow w \in L(U)$$

- ( $\Rightarrow$ ) We need to show that  $w \notin L \Rightarrow w \in L(U)$
- Assume  $w \notin L$
- Given that  $M$  is a dfa, the following is the computation it performs on  $w$ :  
 $((w) S) \vdash^*_M ((Q) Q)$ , where  $Q \notin F$
- By construction of  $U$ ,  $Q \in F'$  and  $U$  performs the same computation moving from  $S$  to  $Q$  by consuming  $w$
- Therefore,  $w \in L(U)$ .
- ( $\Leftarrow$ ) We need to show that  $w \in L(U) \Rightarrow w \notin L$

# FSA and Regular Expressions

## Lemma

$$w \notin L \Leftrightarrow w \in L(U)$$

- ( $\Rightarrow$ ) We need to show that  $w \notin L \Rightarrow w \in L(U)$
- Assume  $w \notin L$
- Given that  $M$  is a dfa, the following is the computation it performs on  $w$ :  
 $((w) S) \vdash^*_M ((Q), \text{ where } Q \notin F)$
- By construction of  $U$ ,  $Q \in F'$  and  $U$  performs the same computation moving from  $S$  to  $Q$  by consuming  $w$
- Therefore,  $w \in L(U)$ .
- ( $\Leftarrow$ ) We need to show that  $w \in L(U) \Rightarrow w \notin L$
- Assume  $w \in L(U)$
- This means that  $U$  performs the following computation on  $w$ :  
 $((w) S) \vdash^*_U ((Q), \text{ where } Q \in F')$
- By construction of  $U$ ,  $Q \notin F$  and  $M$  performs the same computation moving from  $S$  to  $Q$  by consuming  $w$
- Thus,  $w \notin L$

# FSA and Regular Expressions

Regular  
Expressions

Deterministic  
Finite  
Automata

Nondeterministic  
Finite  
Automata

Finite-State  
Automata and  
Regular  
Expressions

Regular  
Grammars

Pumping  
Theorem for  
Regular  
Languages

## Theorem

*The languages accepted by finite-state machines are closed under complement.*

- The proof follows from the previous two lemmas

# FSA and Regular Expressions

- HOMEWORK: 12, 14

# FSA and Regular Expressions

## Theorem

*The languages decided by finite-state automata are closed under intersection.*

- $M = (\text{make-ndfa } S_M \Sigma_M A F_M \delta_M)$   
 $N = (\text{make-ndfa } S_N \Sigma_N R F_N \delta_N)$
- We need to construct an ndfa that accepts and only accepts the words in  $L(M) \cap L(N)$

# FSA and Regular Expressions

## Theorem

*The languages decided by finite-state automata are closed under intersection.*

- $M = (\text{make-ndfa } S_M \Sigma_M A F_M \delta_M)$   
 $N = (\text{make-ndfa } S_N \Sigma_N R F_N \delta_N)$
- We need to construct an ndfa that accepts and only accepts the words in  $L(M) \cap L(N)$
- Consider the following facts from set theory:  
 $\Sigma^* - B = \{w \mid w \in \Sigma^* \wedge w \notin B\}$   
 $\Sigma^* - A = \{w \mid w \in \Sigma^* \wedge w \notin A\}$
- All possible words in  $\Sigma^*$  that are not in B and all possible words in  $\Sigma^*$  that are not in A

# FSA and Regular Expressions

## Theorem

*The languages decided by finite-state automata are closed under intersection.*

- $M = (\text{make-ndfa } S_M \Sigma_M A F_M \delta_M)$   
 $N = (\text{make-ndfa } S_N \Sigma_N R F_N \delta_N)$
- We need to construct an ndfa that accepts and only accepts the words in  $L(M) \cap L(N)$
- Consider the following facts from set theory:

$$\begin{aligned}\Sigma^* - B &= \{w \mid w \in \Sigma^* \wedge w \notin B\} \\ \Sigma^* - A &= \{w \mid w \in \Sigma^* \wedge w \notin A\}\end{aligned}$$

- All possible words in  $\Sigma^*$  that are not in B and all possible words in  $\Sigma^*$  that are not in A
- Consider the union of these two sets:

$$\{\Sigma^* - B\} \cup \{\Sigma^* - A\} = \{w \mid w \notin A \wedge w \notin B\}$$

- What words are not contained in this union?

# FSA and Regular Expressions

## Theorem

*The languages decided by finite-state automata are closed under intersection.*

- $M = (\text{make-ndfa } S_M \Sigma_M A F_M \delta_M)$   
 $N = (\text{make-ndfa } S_N \Sigma_N R F_N \delta_N)$
- We need to construct an ndfa that accepts and only accepts the words in  $L(M) \cap L(N)$
- Consider the following facts from set theory:

$$\begin{aligned}\Sigma^* - B &= \{w \mid w \in \Sigma^* \wedge w \notin B\} \\ \Sigma^* - A &= \{w \mid w \in \Sigma^* \wedge w \notin A\}\end{aligned}$$

- All possible words in  $\Sigma^*$  that are not in B and all possible words in  $\Sigma^*$  that are not in A
- Consider the union of these two sets:

$$\{\Sigma^* - B\} \cup \{\Sigma^* - A\} = \{w \mid w \notin A \wedge w \notin B\}$$

- What words are not contained in this union?
- It is exactly the elements that are in both A and B

# FSA and Regular Expressions

## Theorem

*The languages decided by finite-state automata are closed under intersection.*

- $M = (\text{make-ndfa } S_M \Sigma_M A F_M \delta_M)$   
 $N = (\text{make-ndfa } S_N \Sigma_N R F_N \delta_N)$
- We need to construct an ndfa that accepts and only accepts the words in  $L(M) \cap L(N)$
- Consider the following facts from set theory:

$$\begin{aligned}\Sigma^* - B &= \{w \mid w \in \Sigma^* \wedge w \notin B\} \\ \Sigma^* - A &= \{w \mid w \in \Sigma^* \wedge w \notin A\}\end{aligned}$$

- All possible words in  $\Sigma^*$  that are not in B and all possible words in  $\Sigma^*$  that are not in A
- Consider the union of these two sets:

$$\{\Sigma^* - B\} \cup \{\Sigma^* - A\} = \{w \mid w \notin A \wedge w \notin B\}$$

- What words are not contained in this union?
- It is exactly the elements that are in both A and B
- We may define the language for the machine we wish to implement as follows:

$$\begin{aligned}L(M) \cap L(N) &= \Sigma^* - \{\{\Sigma^* - L(M)\} \cup \{\Sigma^* - L(N)\}\} \\ &= \Sigma^* - \{\bar{L}(M) \cup \bar{L}(N)\}\end{aligned}$$

# FSA and Regular Expressions

- ;; ndfa ndfa → ndfa

;; Purpose: Construct an ndfa for the intersection of the languages  
;; given ndfas  
`(define (intersect-fsa M N)`

# FSA and Regular Expressions

- ```
;; ndfa ndfa → ndfa
;; Purpose: Construct an ndfa for the intersection of the languages
;;           given ndfas
(define (intersect-fsa M N)
```
- ```
;; Tests for intersect-fsa
(define ab*-intersect-a-aUb-b* (intersect-fsa ab* a-aUb-b*))
(define a-aUb-b*-intersect-EVEN-A-ODD-B (intersect-fsa a-aUb-b*
  EVEN-A-ODD-B)

(check-equal? (sm-apply ab*-intersect-a-aUb-b* '()) 'reject)
(check-equal? (sm-apply ab*-intersect-a-aUb-b* '(a b b a)) 'reject)
(check-equal? (sm-apply ab*-intersect-a-aUb-b* '(a b)) 'reject)
(check-equal? (sm-testequiv? ab*-intersect-a-aUb-b*
                           (sm-intersection ab* a-aUb-b*))
               #t)

(check-equal? (sm-apply a-aUb-b*-intersect-EVEN-A-ODD-B '()) 'reject)
(check-equal? (sm-apply a-aUb-b*-intersect-EVEN-A-ODD-B '(b b)) 'reject)
```

# FSA and Regular Expressions

- ```
;; ndfa ndfa → ndfa
;; Purpose: Construct an ndfa for the intersection of the languages
;;           given ndfas
(define (intersect-fsa M N)
```
- ```
(let* [(notM (sm-rename-states
                  (list DEAD)
                  (sm-complement (ndfa->dfa M))))
        (notN (sm-rename-states
                  (list DEAD)
                  (sm-complement (ndfa->dfa N)))]
```
- ```
; Tests for intersect-fsa
(define ab*-intersect-a-aUb-b* (intersect-fsa ab* a-aUb-b*))
(define a-aUb-b*-intersect-EVEN-A-ODD-B (intersect-fsa a-aUb-b*
                                              EVEN-A-ODD-  
  
(check-equal? (sm-apply ab*-intersect-a-aUb-b* '()) 'reject)
(check-equal? (sm-apply ab*-intersect-a-aUb-b* '(a b b a)) 'reject)
(check-equal? (sm-apply ab*-intersect-a-aUb-b* '(a b)) 'reject)
(check-equal? (sm-testequiv? ab*-intersect-a-aUb-b*
                           (sm-intersection ab* a-aUb-b*))
               #t)  
  
(check-equal? (sm-apply a-aUb-b*-intersect-EVEN-A-ODD-B '()) 'reject)
(check-equal? (sm-apply a-aUb-b*-intersect-EVEN-A-ODD-B '(b b)) 'reject)
```

## FSA and Regular Expressions

- ```

• ;; ndfa ndfa → ndfa
;; Purpose: Construct an ndfa for the intersection of the languages
;; given ndfas
(define (intersect-fsa M N)
• (let* [(notM (sm-rename-states
                  (list DEAD)
                  (sm-complement (ndfa->dfa M))))
        (notN (sm-rename-states
                  (list DEAD)
                  (sm-complement (ndfa->dfa N))))]
•   (complement-fsa (ndfa->dfa (sm-union notM notN)))))

• ;; Tests for intersect-fsa
(define ab*-intersect-a-aUb-b* (intersect-fsa ab* a-aUb-b*))
(define a-aUb-b*-intersect-EVEN-A-ODD-B (intersect-fsa a-aUb-b*
   EVEN-A-ODD-)

(check-equal? (sm-apply ab*-intersect-a-aUb-b* '()) 'reject)
(check-equal? (sm-apply ab*-intersect-a-aUb-b* '(a b b a)) 'reject)
(check-equal? (sm-apply ab*-intersect-a-aUb-b* '(a b)) 'reject)
(check-equal? (sm-testequiv? ab*-intersect-a-aUb-b*
                               (sm-intersection ab* a-aUb-b*))
              #t)

(check-equal? (sm-apply a-aUb-b*-intersect-EVEN-A-ODD-B '()) 'reject)
(check-equal? (sm-apply a-aUb-b*-intersect-EVEN-A-ODD-B '(b b)) 'reject)

```

# FSA and Regular Expressions

Regular  
Expressions

Deterministic  
Finite  
Automata

Nondeterministic  
Finite  
Automata

Finite-State  
Automata and  
Regular  
Expressions

Regular  
Grammars

Pumping  
Theorem for  
Regular  
Languages

- Define three machines as follows:

$$M = (\text{make-ndfa } S \ \Sigma \ Z \ F \ \delta)$$

$$N = (\text{make-ndfa } S' \ \Sigma' \ Z' \ F' \ \delta')$$

$$U = (\text{intersect-fsa } M \ N) = (\text{make-ndfa } S'' \ \Sigma'' \ Z'' \ F'' \ \delta'')$$

- Let  $L = L(M) \cap L(N)$
- We proceed to prove that  $L = L(U)$

# FSA and Regular Expressions

## Lemma

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

- ( $\Rightarrow$ ) We need to show that  $w \in L \Rightarrow w \in L(U)$

# FSA and Regular Expressions

Regular  
Expressions

Deterministic  
Finite  
Automata

Nondeterministic  
Finite  
Automata

Finite-State  
Automata and  
Regular  
Expressions

Regular  
Grammars

Pumping  
Theorem for  
Regular  
Languages

## Lemma

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

- ( $\Rightarrow$ ) We need to show that  $w \in L \Rightarrow w \in L(U)$
- Assume  $w \in L$
- This means that  $w \in L(M)$  and  $w \in L(N)$
- Therefore,  $w \in (\Sigma^* - \{\bar{L}(M) \cup \bar{L}(N)\})$
- By construction of  $U$ ,  $w \in L(U)$

# FSA and Regular Expressions

Regular  
Expressions

Deterministic  
Finite  
Automata

Nondeterministic  
Finite  
Automata

Finite-State  
Automata and  
Regular  
Expressions

Regular  
Grammars

Pumping  
Theorem for  
Regular  
Languages

## Lemma

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

- ( $\Rightarrow$ ) We need to show that  $w \in L \Rightarrow w \in L(U)$
- Assume  $w \in L$
- This means that  $w \in L(M)$  and  $w \in L(N)$
- Therefore,  $w \in (\Sigma^* - \{\bar{L}(M) \cup \bar{L}(N)\})$
- By construction of  $U$ ,  $w \in L(U)$
- ( $\Leftarrow$ ) We need to show that  $w \in L(U) \Rightarrow w \in L$

# FSA and Regular Expressions

Regular  
Expressions

Deterministic  
Finite  
Automata

Nondeterministic  
Finite  
Automata

Finite-State  
Automata and  
Regular  
Expressions

Regular  
Grammars

Pumping  
Theorem for  
Regular  
Languages

## Lemma

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

- ( $\Rightarrow$ ) We need to show that  $w \in L \Rightarrow w \in L(U)$
- Assume  $w \in L$
- This means that  $w \in L(M)$  and  $w \in L(N)$
- Therefore,  $w \in (\Sigma^* - \{\bar{L}(M) \cup \bar{L}(N)\})$
- By construction of  $U$ ,  $w \in L(U)$
- ( $\Leftarrow$ ) We need to show that  $w \in L(U) \Rightarrow w \in L$
- Assume  $w \in L(U)$
- By  $U$ 's construction,  $w \in (\Sigma^* - \{\bar{L}(M) \cup \bar{L}(N)\})$
- Therefore,  $w \in L(M)$  and  $w \in L(N)$
- Thus,  $w \in L$

# FSA and Regular Expressions

## Lemma

$$w \notin L \Leftrightarrow w \notin L(U)$$

- ( $\Rightarrow$ ) We need to show that  $w \notin L \Rightarrow w \notin L(U)$

# FSA and Regular Expressions

Regular  
Expressions

Deterministic  
Finite  
Automata

Nondeterministic  
Finite  
Automata

Finite-State  
Automata and  
Regular  
Expressions

Regular  
Grammars

Pumping  
Theorem for  
Regular  
Languages

## Lemma

$$w \notin L \Leftrightarrow w \notin L(U)$$

- ( $\Rightarrow$ ) We need to show that  $w \notin L \Rightarrow w \notin L(U)$
- Assume  $w \notin L$
- This means that  $w \notin L(M)$  or  $w \notin L(N)$
- Therefore,  $w \notin (\Sigma^* - \{\bar{L}(M) \cup \bar{L}(N)\})$
- By construction of  $U$ ,  $w \notin L(U)$

# FSA and Regular Expressions

Regular  
Expressions

Deterministic  
Finite  
Automata

Nondeterministic  
Finite  
Automata

Finite-State  
Automata and  
Regular  
Expressions

Regular  
Grammars

Pumping  
Theorem for  
Regular  
Languages

## Lemma

$$w \notin L \Leftrightarrow w \notin L(U)$$

- ( $\Rightarrow$ ) We need to show that  $w \notin L \Rightarrow w \notin L(U)$
- Assume  $w \notin L$
- This means that  $w \notin L(M)$  or  $w \notin L(N)$
- Therefore,  $w \notin (\Sigma^* - \{\bar{L}(M) \cup \bar{L}(N)\})$
- By construction of  $U$ ,  $w \notin L(U)$
- ( $\Leftarrow$ ) We need to show that  $w \notin L(U) \Rightarrow w \notin L$

# FSA and Regular Expressions

Regular  
Expressions

Deterministic  
Finite  
Automata

Nondeterministic  
Finite  
Automata

Finite-State  
Automata and  
Regular  
Expressions

Regular  
Grammars

Pumping  
Theorem for  
Regular  
Languages

## Lemma

$$w \notin L \Leftrightarrow w \notin L(U)$$

- ( $\Rightarrow$ ) We need to show that  $w \notin L \Rightarrow w \notin L(U)$
- Assume  $w \notin L$
- This means that  $w \notin L(M)$  or  $w \notin L(N)$
- Therefore,  $w \notin (\Sigma^* - \{\bar{L}(M) \cup \bar{L}(N)\})$
- By construction of  $U$ ,  $w \notin L(U)$
- ( $\Leftarrow$ ) We need to show that  $w \notin L(U) \Rightarrow w \notin L$
- Assume  $w \notin L(U)$
- By  $U$ 's construction,  $w \notin (\Sigma^* - \{\bar{L}(M) \cup \bar{L}(N)\})$
- Therefore,  $w \notin L(M)$  or  $w \notin L(N)$
- Thus,  $w \notin L$

# FSA and Regular Expressions

Regular  
Expressions

Deterministic  
Finite  
Automata

Nondeterministic  
Finite  
Automata

Finite-State  
Automata and  
Regular  
Expressions

Regular  
Grammars

Pumping  
Theorem for  
Regular  
Languages

## Theorem

*The languages decided by finite-state automata are closed under intersection.*

- The proof follows from the previous two lemmas

# FSA and Regular Expressions

- HOMEWORK: 15
- Quiz: 16 (due in 1 week)

# FSA and Regular Expressions

- Is there a finite-state machine for the language of any regular expression?

# FSA and Regular Expressions

- Is there a finite-state machine for the language of any regular expression?
- Is there is a regular expression for any language decided by a finite-state machine?

# FSA and Regular Expressions

- Is there a finite-state machine for the language of any regular expression?
- Is there a regular expression for any language decided by a finite-state machine?
- Our goal: establish the equivalence of finite-state machines and regular expressions

# FSA and Regular Expressions

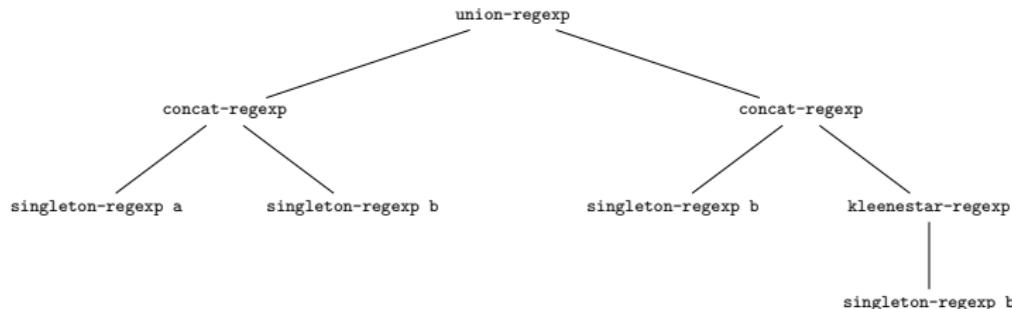
- We start by proving that there is a finite-state machine that decides the language of a regular expression

# FSA and Regular Expressions

- We start by proving that there is a finite-state machine that decides the language of a regular expression
- Think about the structure of a regular expression
- A regular expression may be thought of as a tree
- The empty regular expression and the singleton regular expressions are leaves
- The concatenation, union, and Kleene star regular expressions are interior nodes

# FSA and Regular Expressions

- We start by proving that there is a finite-state machine that decides the language of a regular expression
- Think about the structure of a regular expression
- A regular expression may be thought of as a tree
- The empty regular expression and the singleton regular expressions are leaves
- The concatenation, union, and Kleene star regular expressions are interior nodes
- $ab \cup bb^*$



- A regular expression may be processed using structural recursion

# FSA and Regular Expressions

- To build an `ndfa`, a regular expression,  $e$ , and the language's alphabet,  $\Sigma$ , are needed

# FSA and Regular Expressions

Regular  
Expressions

Deterministic  
Finite  
Automata

Nondeterministic  
Finite  
Automata

Finite-State  
Automata and  
Regular  
Expressions

Regular  
Grammars

Pumping  
Theorem for  
Regular  
Languages

- To build an `ndfa`, a regular expression,  $e$ , and the language's alphabet,  $\Sigma$ , are needed
- The subtype of the given regular expression is determined and the appropriate `ndfa` is constructed

# FSA and Regular Expressions

Regular  
Expressions

Deterministic  
Finite  
Automata

Nondeterministic  
Finite  
Automata

Finite-State  
Automata and  
Regular  
Expressions

Regular  
Grammars

Pumping  
Theorem for  
Regular  
Languages

- To build an `ndfa`, a regular expression,  $e$ , and the language's alphabet,  $\Sigma$ , are needed
- The subtype of the given regular expression is determined and the appropriate `ndfa` is constructed
- If the subtype is an empty regular expression then an `ndfa` that only accepts empty is returned.

# FSA and Regular Expressions

Regular  
Expressions

Deterministic  
Finite  
Automata

Nondeterministic  
Finite  
Automata

Finite-State  
Automata and  
Regular  
Expressions

Regular  
Grammars

Pumping  
Theorem for  
Regular  
Languages

- To build an `ndfa`, a regular expression,  $e$ , and the language's alphabet,  $\Sigma$ , are needed
- The subtype of the given regular expression is determined and the appropriate `ndfa` is constructed
- If the subtype is an empty regular expression then an `ndfa` that only accepts empty is returned.
- If the subtype is a singleton regular expression for  $x \in \Sigma$  then an `ndfa` that only accepts  $x$  is returned

# FSA and Regular Expressions

Regular  
Expressions

Deterministic  
Finite  
Automata

Nondeterministic  
Finite  
Automata

Finite-State  
Automata and  
Regular  
Expressions

Regular  
Grammars

Pumping  
Theorem for  
Regular  
Languages

- To build an `ndfa`, a regular expression,  $e$ , and the language's alphabet,  $\Sigma$ , are needed
- The subtype of the given regular expression is determined and the appropriate `ndfa` is constructed
- If the subtype is an empty regular expression then an `ndfa` that only accepts empty is returned.
- If the subtype is a singleton regular expression for  $x \in \Sigma$  then an `ndfa` that only accepts  $x$  is returned
- If the subtype is a union regular expression then closure under union is used

# FSA and Regular Expressions

- To build an `ndfa`, a regular expression,  $e$ , and the language's alphabet,  $\Sigma$ , are needed
- The subtype of the given regular expression is determined and the appropriate `ndfa` is constructed
- If the subtype is an empty regular expression then an `ndfa` that only accepts empty is returned.
- If the subtype is a singleton regular expression for  $x \in \Sigma$  then an `ndfa` that only accepts  $x$  is returned
- If the subtype is a union regular expression then closure under union is used
- If the subtype is a concatenation regular expression then closure under concatenation is used

# FSA and Regular Expressions

Regular  
Expressions

Deterministic  
Finite  
Automata

Nondeterministic  
Finite  
Automata

Finite-State  
Automata and  
Regular  
Expressions

Regular  
Grammars

Pumping  
Theorem for  
Regular  
Languages

- To build an `ndfa`, a regular expression,  $e$ , and the language's alphabet,  $\Sigma$ , are needed
- The subtype of the given regular expression is determined and the appropriate `ndfa` is constructed
- If the subtype is an empty regular expression then an `ndfa` that only accepts empty is returned.
- If the subtype is a singleton regular expression for  $x \in \Sigma$  then an `ndfa` that only accepts  $x$  is returned
- If the subtype is a union regular expression then closure under union is used
- If the subtype is a concatenation regular expression then closure under concatenation is used
- If the subtype is a Kleene star regular expression then closure under Kleene star is used

# FSA and Regular Expressions

- `; ; regexp alphabet → ndfa   Purpose: Build ndfa for given regexp  
(define (regexp->ndfa e sigma)`

# FSA and Regular Expressions

- ;; Tests for reg-exp->ndfa

```
(define e (empty-regexp))
(define a (singleton-regexp "a"))
(define b (singleton-regexp "b"))
(define ab (concat-regexp a b))
(define aa (concat-regexp a a))
(define abUe (union-regexp ab e))
(define abUaa (union-regexp ab aa))
(define aa-* (kleenestar-regexp aa))
(define abUaa-* (kleenestar-regexp abUaa))
```

# FSA and Regular Expressions

- ;; Tests for reg-exp->ndfa  
(define e (empty-regexp))  
(define a (singleton-regexp "a"))  
(define b (singleton-regexp "b"))  
(define ab (concat-regexp a b))  
(define aa (concat-regexp a a))  
(define abUe (union-regexp ab e))  
(define abUaa (union-regexp ab aa))  
(define aa-\* (kleenestar-regexp aa))  
(define abUaa-\* (kleenestar-regexp abUaa))
- (define Me (regexp->ndfa e '(a b)))  
(define Ma (regexp->ndfa a '(a b)))  
(define Mb (regexp->ndfa b '(a b)))  
(define Mab (regexp->ndfa ab '(a b)))  
(define Maa (regexp->ndfa aa '(a b)))  
(define MabUMe (regexp->ndfa abUe '(a b)))  
(define MabUaa (regexp->ndfa abUaa '(a b)))  
(define Maa-\* (regexp->ndfa aa-\* '(a b)))  
(define MabUaa-\* (regexp->ndfa abUaa-\* '(a b)))

# FSA and Regular Expressions

- ;; Tests for reg-exp->ndfa  
(define e (empty-regexp))  
(define a (singleton-regexp "a"))  
(define b (singleton-regexp "b"))  
(define ab (concat-regexp a b))  
(define aa (concat-regexp a a))  
(define abUe (union-regexp ab e))  
(define abUaa (union-regexp ab aa))  
(define aa-\* (kleenestar-regexp aa))  
(define abUaa-\* (kleenestar-regexp abUaa))
- (define Me (regexp->ndfa e '(a b)))  
(define Ma (regexp->ndfa a '(a b)))  
(define Mb (regexp->ndfa b '(a b)))  
(define Mab (regexp->ndfa ab '(a b)))  
(define Maa (regexp->ndfa aa '(a b)))  
(define MabUMe (regexp->ndfa abUe '(a b)))  
(define MabUaa (regexp->ndfa abUaa '(a b)))  
(define Maa-\* (regexp->ndfa aa-\* '(a b)))  
(define MabUaa-\* (regexp->ndfa abUaa-\* '(a b)))
- (check-equal? (sm-apply Me '(a)) 'reject)  
(check-equal? (sm-apply Me '()) 'accept)  
(check-equal? (sm-apply Ma '(b)) 'reject)  
(check-equal? (sm-apply Ma '(a)) 'accept)  
(check-equal? (sm-apply Mab '()) 'reject)  
(check-equal? (sm-apply Mab '(a b)) 'accept)

# FSA and Regular Expressions

- ;; regexp alphabet → ndfa Purpose: Build ndfa for given regexp  

```
(define (regexp->ndfa e sigma)
  (let* [(simple-tbl (map (λ (a)
                               (let [(S (generate-symbol 'S '(S)))
                                     (A (generate-symbol 'A '(A)))]
                                 (list a (make-ndfa
   (list S A)
   sigma
   S
   (list A)
   (list (list S a A))))))
                           (cons EMP sigma)))]
```

# FSA and Regular Expressions

- `;; regexp alphabet → ndfa Purpose: Build ndfa for given regexp`  
`(define (regexp->ndfa e sigma)`  
 `(let* [(simple-tbl (map (λ (a)`  
 `(let [([S (generate-symbol 'S '(S)))`  
 `(A (generate-symbol 'A '(A)))]`  
 `(list a (make-ndfa`  
 `(list S A)`  
 `sigma`  
 `S`  
 `(list A)`  
 `(list (list S a A))))])`  
 `(cons EMP sigma)))]`  
`)`  
 `(cond [(empty-regexp? e) (second (assoc EMP simple-tbl))]`

# FSA and Regular Expressions

- `;; regexp alphabet → ndfa Purpose: Build ndfa for given regexp`  
`(define (regexp->ndfa e sigma)`  
 `(let* [(simple-tbl (map (λ (a)`  
 `(let [(S (generate-symbol 'S '(S)))`  
 `(A (generate-symbol 'A '(A))))]`  
 `(list a (make-ndfa`  
 `(list S A)`  
 `sigma`  
 `S`  
 `(list A)`  
 `(list (list S a A))))))])`  
 `(cons EMP sigma)))]`
- `(cond [(empty-regexp? e) (second (assoc EMP simple-tbl))]`
- `[(singleton-regexp? e)`  
 `(second (assoc (string->symbol (singleton-regexp-a e))`  
 `simple-tbl))]`

# FSA and Regular Expressions

- ;; regexp alphabet → ndfa Purpose: Build ndfa for given regexp  

```
(define (regexp->ndfa e sigma)
  (let* [(simple-tbl (map (λ (a)
                               (let [(S (generate-symbol 'S '(S)))
                                     (A (generate-symbol 'A '(A)))]
                                 (list a (make-ndfa
   (list S A)
   sigma
   S
   (list A)
   (list (list S a A))))))
                               (cons EMP sigma)))]
```
- (cond [(empty-regexp? e) (second (assoc EMP simple-tbl))])
- [(singleton-regexp? e)
 (second (assoc (string->symbol (singleton-regexp-a e))
 simple-tbl)))]
- [(concat-regexp? e)
 (concat-fsa (regexp->ndfa (concat-regexp-r1 e) sigma)
 (regexp->ndfa (concat-regexp-r2 e) sigma)))]

# FSA and Regular Expressions

- `; ; regexp alphabet → ndfa Purpose: Build ndfa for given regexp`  
`(define (regexp->ndfa e sigma)`  
 `(let* [(simple-tbl (map (λ (a)`  
 `(let [([S (generate-symbol 'S '(S)))`  
 `(A (generate-symbol 'A '(A)))]`  
 `(list a (make-ndfa`  
 `(list S A)`  
 `sigma`  
 `S`  
 `(list A)`  
 `(list (list S a A))))]))`  
 `(cons EMP sigma))))]`
- `(cond [(empty-regexp? e) (second (assoc EMP simple-tbl))]`
- `[(singleton-regexp? e)`  
 `(second (assoc (string->symbol (singleton-regexp-a e)`  
 `simple-tbl)))]`
- `[(concat-regexp? e)`  
 `(concat-fsa (regexp->ndfa (concat-regexp-r1 e) sigma)`  
 `(regexp->ndfa (concat-regexp-r2 e) sigma)))]`
- `[(union-regexp? e)`  
 `(union-fsa (regexp->ndfa (union-regexp-r1 e) sigma)`  
 `(regexp->ndfa (union-regexp-r2 e) sigma)))]`

# FSA and Regular Expressions

- `; ; regexp alphabet → ndfa Purpose: Build ndfa for given regexp`  
`(define (regexp->ndfa e sigma)`  
 `(let* [(simple-tbl (map (λ (a)`  
 `(let [([S (generate-symbol 'S '(S)))`  
 `(A (generate-symbol 'A '(A)))]`  
 `(list a (make-ndfa`  
 `(list S A)`  
 `sigma`  
 `S`  
 `(list A)`  
 `(list (list S a A))))))`  
 `(cons EMP sigma)))]`
- `(cond [(empty-regexp? e) (second (assoc EMP simple-tbl))]`
- `[(singleton-regexp? e)`  
 `(second (assoc (string->symbol (singleton-regexp-a e)`  
 `simple-tbl)))]`
- `[(concat-regexp? e)`  
 `(concat-fsa (regexp->ndfa (concat-regexp-r1 e) sigma)`  
 `(regexp->ndfa (concat-regexp-r2 e) sigma)))]`
- `[(union-regexp? e)`  
 `(union-fsa (regexp->ndfa (union-regexp-r1 e) sigma)`  
 `(regexp->ndfa (union-regexp-r2 e) sigma)))]`
- `[else (kstar-fsa (regexp->ndfa (kleenestar-regexp-r1 e)`  
 `sigma))))])])`

# FSA and Regular Expressions

- we shall prove that `regexp->ndfa` is correct
- Given that `regexp->ndfa` uses structural recursion on a binary tree, the proof is by induction on the height of the binary tree

# FSA and Regular Expressions

## Theorem

$L$  is a regular language  $\Rightarrow L$  is decided by a finite-state machine.

- Assume  $L$  is regular

# FSA and Regular Expressions

## Theorem

$L$  is a regular language  $\Rightarrow L$  is decided by a finite-state machine.

- Assume  $L$  is regular
- This means that there is a regular expression,  $R$ , that defines  $L$
- Let  $\Sigma$  be the alphabet for the language of  $R$
- We prove by induction on the height of  $R$  that (`regexp->ndfa R Σ`) builds an `ndfa` for  $L$

# FSA and Regular Expressions

Regular  
Expressions

Deterministic  
Finite  
Automata

Nondeterministic  
Finite  
Automata

Finite-State  
Automata and  
Regular  
Expressions

Regular  
Grammars

Pumping  
Theorem for  
Regular  
Languages

## Theorem

$L$  is a regular language  $\Rightarrow L$  is decided by a finite-state machine.

- Assume  $L$  is regular
- This means that there is a regular expression,  $R$ , that defines  $L$
- Let  $\Sigma$  be the alphabet for the language of  $R$
- We prove by induction on the height of  $R$  that (`regexp->ndfa R Σ`) builds an `ndfa` for  $L$
- Base Case:  $h = 0$
- If  $h$  is zero then  $R$  must be an empty or a singleton regular expression

# FSA and Regular Expressions

Regular  
Expressions

Deterministic  
Finite  
Automata

Nondeterministic  
Finite  
Automata

Finite-State  
Automata and  
Regular  
Expressions

Regular  
Grammars

Pumping  
Theorem for  
Regular  
Languages

## Theorem

$L$  is a regular language  $\Rightarrow L$  is decided by a finite-state machine.

- Assume  $L$  is regular
- This means that there is a regular expression,  $R$ , that defines  $L$
- Let  $\Sigma$  be the alphabet for the language of  $R$
- We prove by induction on the height of  $R$  that `(regexp->ndfa R Σ)` builds an ndfa for  $L$
- Base Case:  $h = 0$
- If  $h$  is zero then  $R$  must be an empty or a singleton regular expression
- If  $R$  is the empty regular expression the `(regexp->ndfa R Σ)` returns an ndfa that only accepts EMP

# FSA and Regular Expressions

Regular  
Expressions

Deterministic  
Finite  
Automata

Nondeterministic  
Finite  
Automata

Finite-State  
Automata and  
Regular  
Expressions

Regular  
Grammars

Pumping  
Theorem for  
Regular  
Languages

## Theorem

$L$  is a regular language  $\Rightarrow L$  is decided by a finite-state machine.

- Assume  $L$  is regular
- This means that there is a regular expression,  $R$ , that defines  $L$
- Let  $\Sigma$  be the alphabet for the language of  $R$
- We prove by induction on the height of  $R$  that  $(\text{regexp-} \rightarrow \text{ndfa } R \ \Sigma)$  builds an ndfa for  $L$
- Base Case:  $h = 0$
- If  $h$  is zero then  $R$  must be an empty or a singleton regular expression
- If  $R$  is the empty regular expression the  $(\text{regexp-} \rightarrow \text{ndfa } R \ \Sigma)$  returns an ndfa that only accepts  $\text{EMP}$
- If  $R$  is  $(\text{singleton-regexp } a)$  then  $(\text{regexp-} \rightarrow \text{ndfa } R \ \Sigma)$  returns an ndfa that only accepts  $a$ , where  $a \in \Sigma$ .
- This establishes the base case

# FSA and Regular Expressions

## Theorem

$L$  is a regular language  $\Rightarrow L$  is decided by a finite-state machine.

- Inductive Step:
- Assume:  $(\text{regexp-} \rightarrow \text{ndfa } R \Sigma)$  returns an ndfa that decides  $L$  for  $h = k$
- Show:  $(\text{regexp-} \rightarrow \text{ndfa } R \Sigma)$  returns an ndfa that decides  $L$  for  $h = k+1$

# FSA and Regular Expressions

## Theorem

$L$  is a regular language  $\Rightarrow L$  is decided by a finite-state machine.

- Inductive Step:
- Assume:  $(\text{regexp-} \rightarrow \text{ndfa } R \Sigma)$  returns an ndfa that decides  $L$  for  $h = k$
- Show:  $(\text{regexp-} \rightarrow \text{ndfa } R \Sigma)$  returns an ndfa that decides  $L$  for  $h = k+1$
- $h \geq 0 \Rightarrow h+1 \geq 1 \Rightarrow$  union, a concatenation, or a Kleene star regexp

# FSA and Regular Expressions

## Theorem

$L$  is a regular language  $\Rightarrow L$  is decided by a finite-state machine.

- Inductive Step:
- Assume:  $(\text{regexp-} \rightarrow \text{ndfa } R \Sigma)$  returns an ndfa that decides  $L$  for  $h = k$
- Show:  $(\text{regexp-} \rightarrow \text{ndfa } R \Sigma)$  returns an ndfa that decides  $L$  for  $h = k+1$
- $h \geq 0 \Rightarrow h+1 \geq 1 \Rightarrow$  union, a concatenation, or a Kleene star regexp
- We analyze each regular expression subtype independently:

# FSA and Regular Expressions

## Theorem

$L$  is a regular language  $\Rightarrow L$  is decided by a finite-state machine.

- Inductive Step:
- Assume:  $(\text{regexp-} \rightarrow \text{ndfa } R \Sigma)$  returns an ndfa that decides  $L$  for  $h = k$
- Show:  $(\text{regexp-} \rightarrow \text{ndfa } R \Sigma)$  returns an ndfa that decides  $L$  for  $h = k+1$
- $h \geq 0 \Rightarrow h+1 \geq 1 \Rightarrow$  union, a concatenation, or a Kleene star regexp
- We analyze each regular expression subtype independently:
- $(\text{union-regexp } S T) (\text{regexp-} \rightarrow \text{ndfa } R \Sigma)$  returns  
$$(\text{union-fsa } (\text{regexp-} \rightarrow \text{ndfa } (\text{union-regexp-r1 } e) \Sigma))$$
$$(\text{regexp-} \rightarrow \text{ndfa } (\text{union-regexp-r2 } e) \Sigma))$$
- $(\text{union-regexp-r1 } e)$ 's and  $(\text{union-regexp-r2 } e)$ 's height at most  $k$
- By IH, recursive calls return ndfas for the language of each
- Closure under union,  $\text{union-fsa}$  returns ndfa for their union

# FSA and Regular Expressions

## Theorem

$L$  is a regular language  $\Rightarrow L$  is decided by a finite-state machine.

- Inductive Step:
- Assume:  $(\text{regexp-} \rightarrow \text{ndfa } R \Sigma)$  returns an ndfa that decides  $L$  for  $h = k$
- Show:  $(\text{regexp-} \rightarrow \text{ndfa } R \Sigma)$  returns an ndfa that decides  $L$  for  $h = k+1$
- $h \geq 0 \Rightarrow h+1 \geq 1 \Rightarrow$  union, a concatenation, or a Kleene star regexp
- We analyze each regular expression subtype independently:
- $(\text{union-regexp } S T) (\text{regexp-} \rightarrow \text{ndfa } R \Sigma)$  returns  
$$(\text{union-fsa } (\text{regexp-} \rightarrow \text{ndfa } (\text{union-regexp-r1 } e) \Sigma))$$
$$(\text{regexp-} \rightarrow \text{ndfa } (\text{union-regexp-r2 } e) \Sigma))$$
- $(\text{union-regexp-r1 } e)$ 's and  $(\text{union-regexp-r2 } e)$ 's height at most  $k$
- By IH, recursive calls return ndfas for the language of each
- Closure under union, union-fsa returns ndfa for their union
- $(\text{concat-regexp } S T) (\text{regexp-} \rightarrow \text{ndfa } R \Sigma)$  returns  
$$(\text{concat-fsa } (\text{regexp-} \rightarrow \text{ndfa } (\text{concat-regexp-r1 } e) \Sigma))$$
$$(\text{regexp-} \rightarrow \text{ndfa } (\text{concat-regexp-r2 } e) \Sigma))$$
- By IH, the recursive calls return ndfas for the language of each
- Closure under concat, concat-fsa returns an ndfa for their concat

# FSA and Regular Expressions

## Theorem

$L$  is a regular language  $\Rightarrow L$  is decided by a finite-state machine.

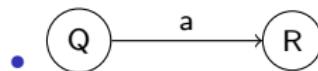
- Inductive Step:
- Assume:  $(\text{regexp-} \rightarrow \text{ndfa } R \Sigma)$  returns an ndfa that decides  $L$  for  $h = k$
- Show:  $(\text{regexp-} \rightarrow \text{ndfa } R \Sigma)$  returns an ndfa that decides  $L$  for  $h = k+1$
- $h \geq 0 \Rightarrow h+1 \geq 1 \Rightarrow$  union, a concatenation, or a Kleene star regexp
- We analyze each regular expression subtype independently:
  - (union-regexp S T)  $(\text{regexp-} \rightarrow \text{ndfa } R \Sigma)$  returns
$$(\text{union-fsa } (\text{regexp-} \rightarrow \text{ndfa } (\text{union-regexp-r1 } e) \Sigma) \\ (\text{regexp-} \rightarrow \text{ndfa } (\text{union-regexp-r2 } e) \Sigma))$$
  - $(\text{union-regexp-r1 } e)$ 's and  $(\text{union-regexp-r2 } e)$ 's height at most  $k$
  - By IH, recursive calls return ndfas for the language of each
  - Closure under union, union-fsa returns ndfa for their union
- (concat-regexp S T)  $(\text{regexp-} \rightarrow \text{ndfa } R \Sigma)$  returns
$$(\text{concat-fsa } (\text{regexp-} \rightarrow \text{ndfa } (\text{concat-regexp-r1 } e) \Sigma) \\ (\text{regexp-} \rightarrow \text{ndfa } (\text{concat-regexp-r2 } e) \Sigma))$$
- By IH, the recursive calls return ndfas for the language of each
- Closure under concat, concat-fsa returns an ndfa for their concat
- $(\text{regexp-} \rightarrow \text{ndfa } R \Sigma)$  returns
$$(\text{kstar-fsa } (\text{regexp-} \rightarrow \text{ndfa } (\text{kleenestar-regexp-r1 } e) \Sigma))$$
- By IH, the recursive call returns an ndfa,  $N$ , for its language
- Closure under Kleene star, kstar-fsa returns an ndfa for  $*$
-

# FSA and Regular Expressions

- To create a regular expression for the language of an ndfa, a regular expression is needed that generates all words that take the given machine from its start state to any its final states
- This means that a regular expression is needed from any state, Q, to any state, R, that is reachable from Q

# FSA and Regular Expressions

- To create a regular expression for the language of an ndfa, a regular expression is needed that generates all words that take the given machine from its start state to any its final states
- This means that a regular expression is needed from any state, Q, to any state, R, that is reachable from Q

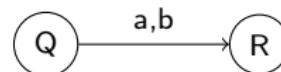


- Regular expression is needed for the part of the word that takes the machine from Q to R
- A singleton regular expression is needed for the rule (Q a R):

`(singleton-regexp "a")`

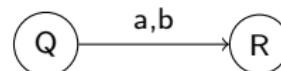
# FSA and Regular Expressions

- There can be more than one transition from Q to R:



# FSA and Regular Expressions

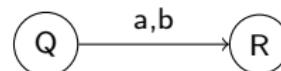
- There can be more than one transition from Q to R:



- The machine can move from Q to R on a a or on a b

# FSA and Regular Expressions

- There can be more than one transition from Q to R:

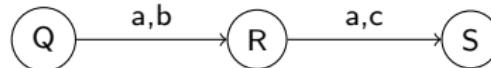


- The machine can move from Q to R on a a or on a b
- A union-regexp is needed:

```
(union-regexp (singleton-regexp "a")  
             (singleton-regexp "b"))
```

# FSA and Regular Expressions

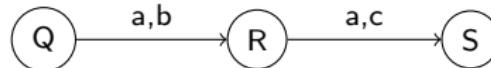
- More than two connected nodes:



- S is reachable from Q
- A regular expression is needed to generate the part of the word that is consumed when the machine moves from Q to S

# FSA and Regular Expressions

- More than two connected nodes:

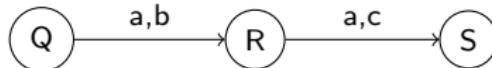


- S is reachable from Q
- A regular expression is needed to generate the part of the word that is consumed when the machine moves from Q to S



# FSA and Regular Expressions

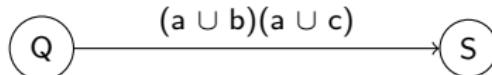
- More than two connected nodes:



- S is reachable from Q
- A regular expression is needed to generate the part of the word that is consumed when the machine moves from Q to S

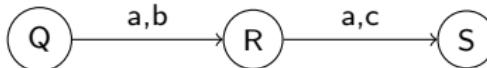


- R may be ripped out, along with the transitions into and out of it
- Substitute with a transition from Q to S that concatenates the regular expressions:



# FSA and Regular Expressions

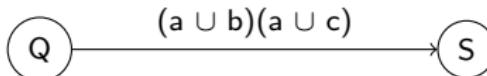
- More than two connected nodes:



- S is reachable from Q
- A regular expression is needed to generate the part of the word that is consumed when the machine moves from Q to S



- R may be ripped out, along with the transitions into and out of it
- Substitute with a transition from Q to S that concatenates the regular expressions:

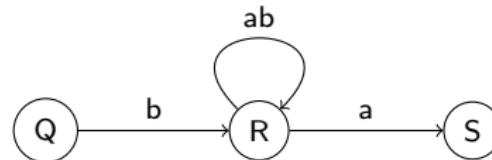


- The needed regular expression:

```
(concat-regexp (union-regexp (singleton-regexp a)
                               (singleton-regexp b))
               (union-regexp (singleton-regexp a)
                           (singleton-regexp c)))
```

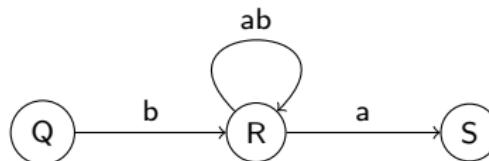
# FSA and Regular Expressions

- The intermediate node R may have a loop on it:

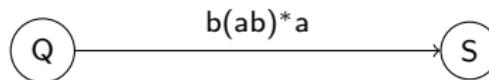


# FSA and Regular Expressions

- The intermediate node R may have a loop on it:

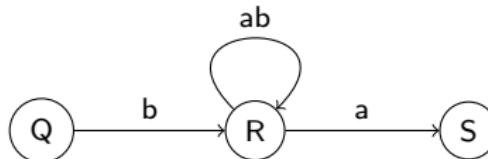


- To rip out the intermediate state the regular expression must generate the part of the word that takes the machine from Q to R, then generates zero or more times the part of the word that takes the machine from R to R, and finally generates the part of the word that takes the machine from R to S
- A Kleene star regular expression is needed:

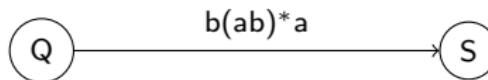


# FSA and Regular Expressions

- The intermediate node R may have a loop on it:



- To rip out the intermediate state the regular expression must generate the part of the word that takes the machine from Q to R, then generates zero or more times the part of the word that takes the machine from R to R, and finally generates the part of the word that takes the machine from R to S
- A Kleene star regular expression is needed:



- The needed regular expression is:

```
(concat-regexp
  (singleton-regexp "b")
  (concat-regexp
    (kleenestar-regexp
      (concat-regexp (singleton-regexp "a")
                    (singleton-regexp "b")))
    (singleton-regexp "a")))
```

# FSA and Regular Expressions

- Given an NFA the goal is to construct a regular expression for all transition diagram paths from the start state to all reachable final states

# FSA and Regular Expressions

- Given an `ndfa` the goal is to construct a regular expression for all transition diagram paths from the start state to all reachable final states
- Create a directed graph and rip out all the states
- Initial digraph has all the machine's states as nodes and the edges are labeled with a regular expression for what is consumed by a transition between two states

# FSA and Regular Expressions

- Given an ndfa the goal is to construct a regular expression for all transition diagram paths from the start state to all reachable final states
- Create a directed graph and rip out all the states
- Initial digraph has all the machine's states as nodes and the edges are labeled with a regular expression for what is consumed by a transition between two states
- In addition, the initial directed graph has two extra nodes representing a new start state and a new and only final state
- There is an  $\epsilon$ -transition from the new start state to the machine's start state and there are  $\epsilon$ -transitions from every machine final state to the new final state

# FSA and Regular Expressions

- Given an `ndfa` the goal is to construct a regular expression for all transition diagram paths from the start state to all reachable final states
- Create a directed graph and rip out all the states
- Initial digraph has all the machine's states as nodes and the edges are labeled with a regular expression for what is consumed by a transition between two states
- In addition, the initial directed graph has two extra nodes representing a new start state and a new and only final state
- There is an  $\epsilon$ -transition from the new start state to the machine's start state and there are  $\epsilon$ -transitions from every machine final state to the new final state
- The process starts by collapsing multiple edges between two nodes into one labeled with a union regular expression
- At each step, this graph is collapsed by ripping out a node

# FSA and Regular Expressions

- Given an `ndfa` the goal is to construct a regular expression for all transition diagram paths from the start state to all reachable final states
- Create a directed graph and rip out all the states
- Initial digraph has all the machine's states as nodes and the edges are labeled with a regular expression for what is consumed by a transition between two states
- In addition, the initial directed graph has two extra nodes representing a new start state and a new and only final state
- There is an  $\epsilon$ -transition from the new start state to the machine's start state and there are  $\epsilon$ -transitions from every machine final state to the new final state
- The process starts by collapsing multiple edges between two nodes into one labeled with a union regular expression
- At each step, this graph is collapsed by ripping out a node
- Ripping out a state may result in multiple edges between nodes and these are collapsed before moving on

# FSA and Regular Expressions

- Given an NFA the goal is to construct a regular expression for all transition diagram paths from the start state to all reachable final states
- Create a directed graph and rip out all the states
- Initial digraph has all the machine's states as nodes and the edges are labeled with a regular expression for what is consumed by a transition between two states
- In addition, the initial directed graph has two extra nodes representing a new start state and a new and only final state
- There is an  $\epsilon$ -transition from the new start state to the machine's start state and there are  $\epsilon$ -transitions from every machine final state to the new final state
- The process starts by collapsing multiple edges between two nodes into one labeled with a union regular expression
- At each step, this graph is collapsed by ripping out a node
- Ripping out a state may result in multiple edges between nodes and these are collapsed before moving on
- After all machine states are ripped out the graph has been collapsed to two states (the new start state and the new final state) with a single edge between them
- The label on that edge is the regular expression for the machine's language

# FSA and Regular Expressions

- To rip out a state,  $S$ , the graph's edges are partitioned into 4 subsets:

*not- $s$ -edges*   *The list of edges that are not into nor out of  $S$*   
*into- $s$ -edges*   *The list of non-loop edges that are into  $S$*   
*outof- $s$ -edges*   *The list of non-loop edges that are out of  $S$*   
*self-edges*   *The list of self-loop edges on  $S$*

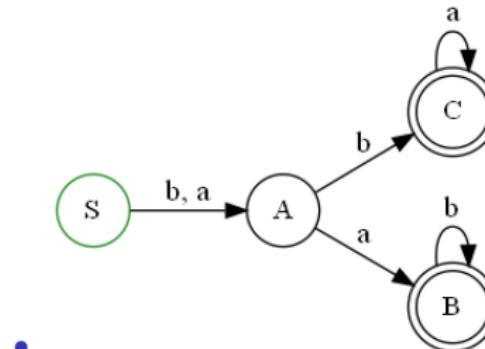
# FSA and Regular Expressions

- To rip out a state,  $S$ , the graph's edges are partitioned into 4 subsets:
  - not-s-edges* *The list of edges that are not into nor out of  $S$*
  - into-s-edges* *The list of non-loop edges that are into  $S$*
  - outof-s-edges* *The list of non-loop edges that are out of  $S$*
  - self-edges* *The list of self-loop edges on  $S$*
- A new graph is constructed using not-s-edges and the new edges created using the other 3 sets of edges
- If  $S$  has a self-loop new edges are created for each incoming edge using the outgoing edges
- Each new edge is from the start node of the incoming node to the destination node of an outgoing edge
- The edge's label is a concatenation regular expression for the label of the incoming edge, the Kleene star of the self-loop label, and the label of an outgoing edge

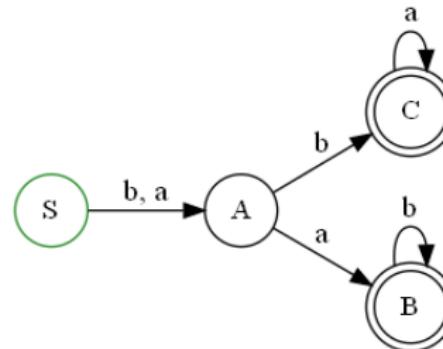
# FSA and Regular Expressions

- To rip out a state,  $S$ , the graph's edges are partitioned into 4 subsets:
  - not-s-edges* *The list of edges that are not into nor out of  $S$*
  - into-s-edges* *The list of non-loop edges that are into  $S$*
  - outof-s-edges* *The list of non-loop edges that are out of  $S$*
  - self-edges* *The list of self-loop edges on  $S$*
- A new graph is constructed using not-s-edges and the new edges created using the other 3 sets of edges
- If  $S$  has a self-loop new edges are created for each incoming edge using the outgoing edges
- Each new edge is from the start node of the incoming node to the destination node of an outgoing edge
- The edge's label is a concatenation regular expression for the label of the incoming edge, the Kleene star of the self-loop label, and the label of an outgoing edge
- If  $S$  does not have a self-loop new edges are also created for each incoming edge using the outgoing edges
- Each new edge is from the start node of the incoming node to the destination node of an outgoing edge
- The edge's label is a concatenation regular expression for the label of the incoming edge and the label of an outgoing edge

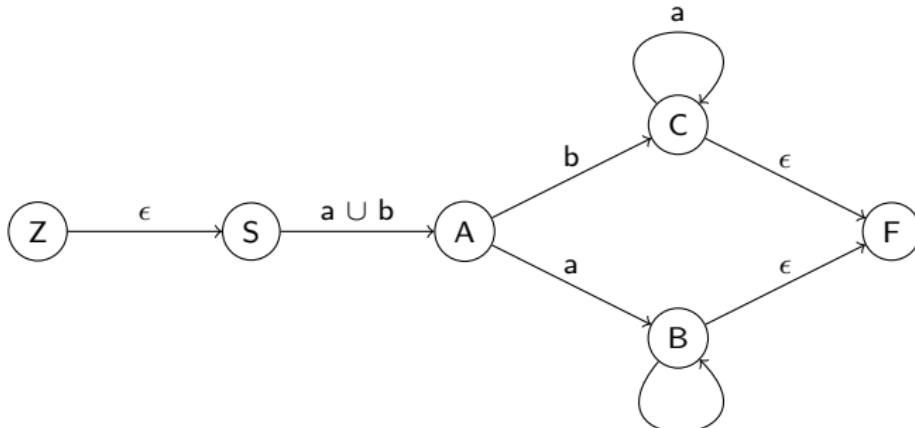
# FSA and Regular Expressions



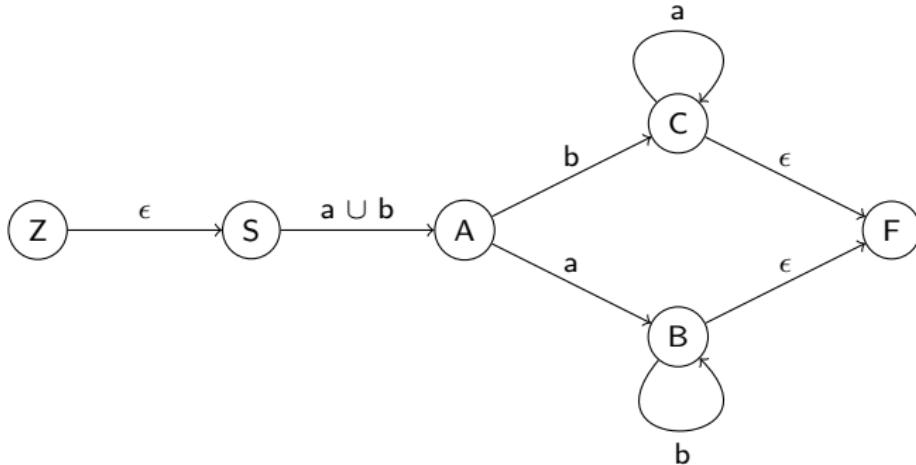
# FSA and Regular Expressions



- Initial Graph collapse multiple edges between nodes:

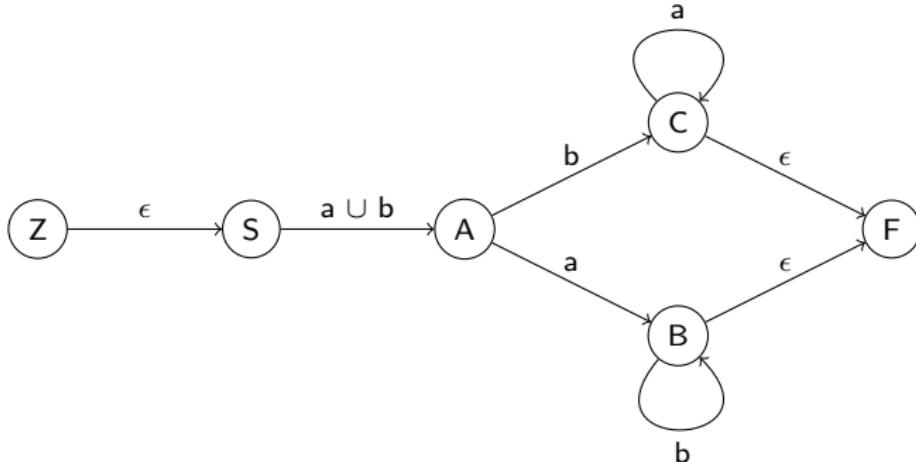


# FSA and Regular Expressions



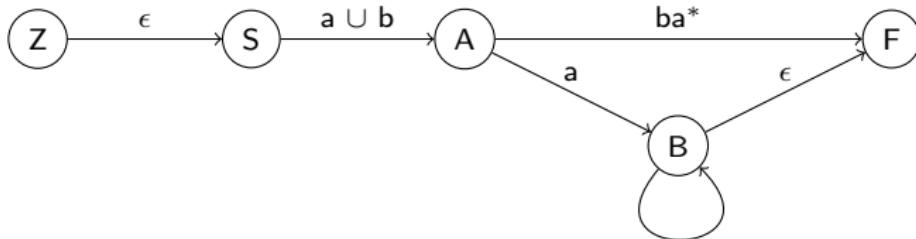
- To collapse the graph, at each step a node is ripped out
  - The order in which they are ripped out does not matter

# FSA and Regular Expressions

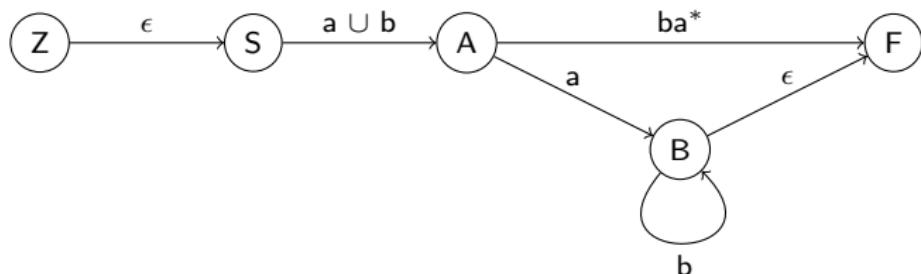


•

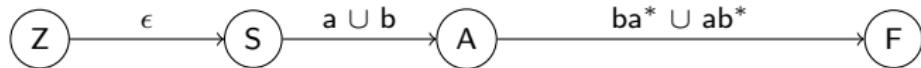
- To collapse the graph, at each step a node is ripped out
- The order in which they are ripped out does not matter
- Let us start by ripping out C:



# FSA and Regular Expressions



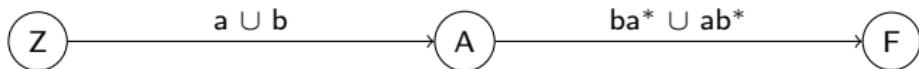
- Let us now rip out B:



# FSA and Regular Expressions



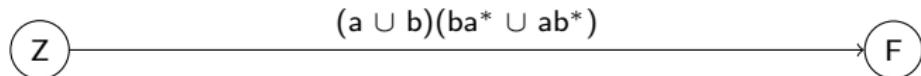
- Let's rip out S:



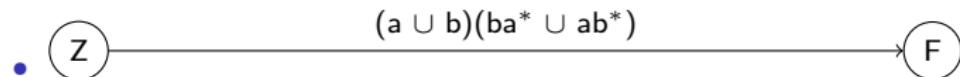
# FSA and Regular Expressions



- Let's rip out A:

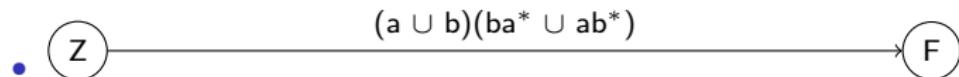


# FSA and Regular Expressions



- All nodes for machine states ripped out

# FSA and Regular Expressions



- All nodes for machine states ripped out
- ```
(concat-regexp
  (union-regexp (singleton-regexp "a")
                (singleton-regexp "b")))
  (union-regexp
    (concat-regexp
      (singleton-regexp "b")
      (kleenestar-regexp (singleton-regexp "a"))))
    (concat-regexp
      (singleton-regexp "a")
      (kleenestar-regexp (singleton-regexp "b"))))))
```

# FSA and Regular Expressions

- The discussion so far has assumed that  $L(M) \neq \emptyset$

# FSA and Regular Expressions

- The discussion so far has assumed that  $L(M) \neq \emptyset$
- If the language of the given machine is empty then the collapsed graph will have no edges
- This is a problem because there is no regular expression for generating no words

# FSA and Regular Expressions

- The discussion so far has assumed that  $L(M) \neq \emptyset$
- If the language of the given machine is empty then the collapsed graph will have no edges
- This is a problem because there is no regular expression for generating no words
- To address this problem FSM introduces a new regular expression constructor, **null-regexp**, to represent a language with no words

# FSA and Regular Expressions

- ```
;; Data Definitions
;;
;; A node is a symbol
;;
;; An edge, (list node regexp node), has a beginning
;; node, a regular expression for its label, and
;; destination node.
;;
;; A directed graph, dgraph, is a (listof edge)
```

# FSA and Regular Expressions

- To test the constructor EVEN-A-ODD-B and the following machines are used:

```
; ; L =  
(define EMPTY (make-ndfa '(S) '(a b) 'S '() '()))  
  
; ; L = ab* U ba*  
(define aUb-ba*Uab*  
  (make-ndfa  
    '(S A B C)  
    '(a b)  
    'S  
    '(B C)  
    '(((S a A) (S b A) (A a B) (A b C) (B b B) (C a C))))  
  
; ; L = b*  
(define b* (make-ndfa `,(DEAD S A)  
                      '(a b)  
                      'S  
                      '(A)  
                      `((S ,EMP A) (S a ,DEAD) (A b A))))
```

# FSA and Regular Expressions

- ;; ndfa → regexp   Purpose: Create a regexp from the given ndfa  
;; Assume: Transition diagram of machine is connected digraph  
`(define (ndfa2regexp m)`

# FSA and Regular Expressions

- ;; ndfa → regexp Purpose: Create a regexp from the given ndfa  
;; Assume: Transition diagram of machine is connected digraph  
`(define (ndfa2regexp m)`

- ;; Tests for ndfa2regexp
  - `(check-equal? (printable-regexp (ndfa2regexp EMPTY)) "()")`
  - `(check-equal? (printable-regexp (ndfa2regexp b*)) "b*")`
  - `(check-equal? (printable-regexp (ndfa2regexp ab*Uaa*)) "(b U a)(ab*)")`
  - `(check-equal? (printable-regexp (ndfa2regexp EVEN-A-ODD-B)) "((ba U ab)(aa U bb)*(ab_U ba) U (aa_U bb))*((ba_U ab)*((aa_U bb)*(ab_U ba) U (aa_U bb)))")`

# FSA and Regular Expressions

- `;; ndfa → regexp` Purpose: Create a regexp from the given ndfa  
;; Assume: Transition diagram of machine is connected digraph  
`(define (ndfa2regexp m)`
- `(let* [(new-start (generate-symbol 'S (sm-states m)))`

- `;; Tests for ndfa2regexp`  
`(check-equal? (printable-regexp (ndfa2regexp EMPTY)) "()")`  
`(check-equal? (printable-regexp (ndfa2regexp b*)) "b*")`  
`(check-equal? (printable-regexp (ndfa2regexp ab*Uaa*)) "(b U a)(ab*`  
`(check-equal? (printable-regexp (ndfa2regexp EVEN-A-ODD-B))`  
`"((ba U ab)(aa U bb)*(ab_U ba) U (aa U bb))*((ba_U ab)`

# FSA and Regular Expressions

- ;; ndfa → regexp Purpose: Create a regexp from the given ndfa  
;; Assume: Transition diagram of machine is connected digraph
  - (define (ndfa2regexp m)
  - (let\* [(new-start (generate-symbol 'S (sm-states m)))
  - (new-final (generate-symbol 'F (sm-states m)))

Regular  
Expressions

Deterministic  
Finite  
Automata

Nondeterministic  
Finite  
Automata

Finite-State  
Automata and  
Regular  
Expressions

Regular  
Grammars

Pumping  
Theorem for  
Regular  
Languages

- ;; Tests for ndfa2regexp
  - (check-equal? (printable-regexp (ndfa2regexp EMPTY)) "()")
  - (check-equal? (printable-regexp (ndfa2regexp b\*)) "b\*")
  - (check-equal? (printable-regexp (ndfa2regexp ab\*Uaa\*)) "(b U a)(ab\*)")
  - (check-equal? (printable-regexp (ndfa2regexp EVEN-A-ODD-B)) "((ba U ab)(aa U bb)\*(ab\_U ba) U (aa U bb))\*((ba\_U ab)\*((aa\_U bb)\*(ab\_U ba) U (aa\_U bb))\*((ba\_U ab)\*(aa\_U bb)))")

# FSA and Regular Expressions

- `;; ndfa → regexp` Purpose: Create a regexp from the given ndfa  
;; Assume: Transition diagram of machine is connected digraph
  - `(define (ndfa2regexp m)`
  - `(let* [(new-start (generate-symbol 'S (sm-states m)))`
  - `(new-final (generate-symbol 'F (sm-states m)))`
  - `(init-dgraph (make-dgraph`
    - `(cons (list new-start EMP (sm-start m))`
    - `(append`
    - `(map (λ (f) (list f EMP new-final))`
    - `(sm-finals m))`
    - `(sm-rules m))))`
- `; ; Tests for ndfa2regexp`
  - `(check-equal? (printable-regexp (ndfa2regexp EMPTY)) "()")`
  - `(check-equal? (printable-regexp (ndfa2regexp b*)) "b*)")`
  - `(check-equal? (printable-regexp (ndfa2regexp ab*Uaa*)) "(b U a)(ab*`
  - `(check-equal? (printable-regexp (ndfa2regexp EVEN-A-ODD-B))`
    - `"((ba U ab)(aa U bb)*(ab_U ba) U (aa U bb))*((ba_U ab)*((aa_U bb)*(bb_U aa))*((aa_U bb)*(bb_U aa)))")`

# FSA and Regular Expressions

- `;; ndfa → regexp` Purpose: Create a regexp from the given ndfa  
;; Assume: Transition diagram of machine is connected digraph
  - (define (ndfa2regexp m)
  - (let\* [(new-start (generate-symbol 'S (sm-states m)))
  - (new-final (generate-symbol 'F (sm-states m)))
  - (init-dgraph (make-dgraph
    - (cons (list new-start EMP (sm-start m))
    - (append
    - (map (λ (f) (list f EMP new-final))
    - (sm-finals m))
    - (sm-rules m))))
  - (collapsed-dgraph
    - (rip-out-nodes (sm-states m)
    - (remove-multiple-edges init-dgraph))))]
  - (check-equal? (printable-regexp (ndfa2regexp EMPTY)) "()")  
(check-equal? (printable-regexp (ndfa2regexp b\*)) "b\*")  
(check-equal? (printable-regexp (ndfa2regexp ab\*Uaa\*)) "(b U a)(ab\*Uaa\*)")  
(check-equal? (printable-regexp (ndfa2regexp EVEN-A-ODD-B))  
"((ba U ab)(aa U bb)\*(ab\_U ba) U (aa U bb))\*((ba\_U ab)\*((aa\_U bb)\*(ab\_U ba)))")

# FSA and Regular Expressions

- `;; ndfa → regexp` Purpose: Create a regexp from the given ndfa  
;; Assume: Transition diagram of machine is connected digraph
  - (define (ndfa2regexp m)
  - (let\* [(new-start (generate-symbol 'S (sm-states m)))
  - (new-final (generate-symbol 'F (sm-states m)))
  - (init-dgraph (make-dgraph
    - (cons (list new-start EMP (sm-start m))
    - (append
    - (map (λ (f) (list f EMP new-final))
    - (sm-finals m))
    - (sm-rules m))))
  - (collapsed-dgraph
    - (rip-out-nodes (sm-states m)
    - (remove-multiple-edges init-dgraph)))]
  - (if (empty? collapsed-dgraph)
  - (null-regexp)
  - (simplify-regexp (second (first collapsed-dgraph)))))
  - ; Tests for ndfa2regexp
    - (check-equal? (printable-regexp (ndfa2regexp EMPTY)) "()")
    - (check-equal? (printable-regexp (ndfa2regexp b\*)) "b\*")
    - (check-equal? (printable-regexp (ndfa2regexp ab\*Uaa\*)) "(b U a)(ab\*)")
    - (check-equal? (printable-regexp (ndfa2regexp EVEN-A-ODD-B)) "((ba U ab)(aa U bb)\*(ab\_U ba) U (aa U bb))\*((ba\_U ab)\*((aa\_U bb)\*(bb\_U aa)))")

# FSA and Regular Expressions

- `;; (listof ndfa-rule) → dgraph`  
;; Purpose: Create a dgraph from the given ndfa  
`(define (make-dgraph lor))`

# FSA and Regular Expressions

- ```
;; (listof ndfa-rule) → dgraph
;; Purpose: Create a dgraph from the given ndfa
(define (make-dgraph lor)
```
- ```
;; Tests for make-dgraph
(check-equal? (make-dgraph '()) '())
(check-equal?
 (make-dgraph `((S ,EMP A) (S a ,DEAD) (A b A)))
 (list (list 'S (empty-regexp) 'A)
       (list 'S (singleton-regexp "a") 'ds)
       (list 'A (singleton-regexp "b") 'A)))
)
(check-equal?
 (make-dgraph `((S a A) (S b A) (A a B) (A b C) (B b B) (C a C))
 (list (list 'S (singleton-regexp "a") 'A)
       (list 'S (singleton-regexp "b") 'A)
       (list 'A (singleton-regexp "a") 'B)
       (list 'A (singleton-regexp "b") 'C)
       (list 'B (singleton-regexp "b") 'B))
 )
)
```

# FSA and Regular Expressions

# FSA and Regular Expressions

- ```
;; dgraph → dgraph
;; Purpose: Collapse multiple edges between nodes
;; Accumulator Invariant: g = the unprocessed graph
(define (remove-multiple-edges g))
```

# FSA and Regular Expressions

- ```
;; dgraph → dgraph
;; Purpose: Collapse multiple edges between nodes
;; Accumulator Invariant: g = the unprocessed graph
(define (remove-multiple-edges g))
```
- ```
;; Tests for remove-multiple-edges
(check-equal? '() '())
(check-equal?
(remove-multiple-edges `((S ,(singleton-regexp "a")) A)
                         (S ,(singleton-regexp "b")) A)
                         (A ,(singleton-regexp "a")) A)))
`((S
      ,(union-regexp (singleton-regexp "a") (singleton-regexp "b"))
      A)
      (A ,(singleton-regexp "a")) A)))
```

# FSA and Regular Expressions

- ```
;; dgraph → dgraph
;; Purpose: Collapse multiple edges between nodes
;; Accumulator Invariant: g = the unprocessed graph
(define (remove-multiple-edges g))
```
  - ```
(if (empty? g)
      '())
```
- 
- ```
;; Tests for remove-multiple-edges
(check-equal? '() '())
(check-equal?
(remove-multiple-edges `((S ,(singleton-regexp "a")) A)
                         (S ,(singleton-regexp "b")) A)
                         (A ,(singleton-regexp "a")) A)))
`((S
      ,(union-regexp (singleton-regexp "a") (singleton-regexp "b"))
      A)
      (A ,(singleton-regexp "a")) A)))
```

# FSA and Regular Expressions

- `;; dgraph → dgraph`  
`;; Purpose: Collapse multiple edges between nodes`  
`;; Accumulator Invariant: g = the unprocessed graph`  
`(define (remove-multiple-edges g)`
- `(if (empty? g)`  
 `'()`
- `(let* [(curr-edge (first g))`  
 `(from-state (first curr-edge))`  
 `(to-state (third curr-edge))`  
 `(to-collapse (filter (λ (e) (and (eq? (first e) from-  
 (eq? (third e) to-sta  
 g))  
 (remaining-g (filter (λ (e) (not (member e to-collapse))  
 (cons (list from-state (collapse-edges to-collapse) to-stat  
 (remove-multiple-edges remaining-g)))))))`
- `;; Tests for remove-multiple-edges`  
`(check-equal? '() '())`  
`(check-equal?`  
 `(remove-multiple-edges `((S ,(singleton-regexp "a") A)  
 (S ,(singleton-regexp "b") A)  
 (A ,(singleton-regexp "a") A)))`  
 ``((S`  
 `,(union-regexp (singleton-regexp "a") (singleton-regexp "b"))  
 A)  
 (A ,(singleton-regexp "a") A)))`

# FSA and Regular Expressions

- `;; (listof edge) → regexp`  
`;; Purpose: Collapse the given edges into a regexp`  
`(define (collapse-edges loe)`  
 `(cond [(empty? loe) '()]`  
 `[(!empty? (rest loe)) (second (first loe))])`  
 `[else (union-regexp (second (first loe))`  
 `(collapse-edges (rest loe))))]))`  
  
`;; Tests for collapse-edges`  
`(check-equal? (collapse-edges '()) '())`  
`(check-equal? (collapse-edges `((S ,(singleton-regexp "a") S)))`  
 `(singleton-regexp "a")))`  
`(check-equal?`  
 `(collapse-edges `((A ,(singleton-regexp "a") A)`  
 `(A ,(singleton-regexp "b") A)`  
 `(A ,(empty-regexp) A))))`  
`(union-regexp (singleton-regexp "a")`  
 `(union-regexp (singleton-regexp "b")`  
 `(empty-regexp))))`

# FSA and Regular Expressions

- ```
;; (listof node) dgraph → dgraph
;; Purpose: Rip out the given nodes from the given graph
;; Assume: Given nodes in given graph and g has no multiple edges
;;          between nodes
(define (rip-out-nodes lon g)
  (foldr (λ (s g) (rip-out-node s g)) g lon))

;; Tests for rip-out-nodes
(check-equal? (rip-out-nodes '() `((S ,(singleton-regexp "a") A)
                                     (A ,(singleton-regexp "b") B)))
                `((S ,(singleton-regexp "a") A)
                  (A ,(singleton-regexp "b") B)))

(check-equal?
  (rip-out-nodes '(A B) `((S ,(singleton-regexp "a") A)
                           (A ,(singleton-regexp "b") B)
                           (B ,(singleton-regexp "b") C)))
  `((S
      ,(concat-regexp (singleton-regexp "a"))
      ,(concat-regexp (singleton-regexp "b"))
      ,(singleton-regexp "b")))

C)))
```

# FSA and Regular Expressions

- `;; state dgraph → dgraph`   Purpose: Rip out given node from given graph  
;; Assume: Given node in given graph and g has no multiple edges between nodes  
`(define (rip-out-node n g))`

# FSA and Regular Expressions

- ;; Tests for rip-out-node  
(check-equal?  
 (rip-out-node  
 'A  
 `((S ,(singleton-regexp "a") A) (A ,(singleton-regexp "b") B)))  
 `((S  
 ,(concat-regexp (singleton-regexp "a") (singleton-regexp "b"))  
 B)))  
 (check-equal?  
 (rip-out-node  
 'C  
 `((S ,(singleton-regexp "a") A) (S ,(singleton-regexp "b") B)  
 (A ,(singleton-regexp "a") C) (B ,(singleton-regexp "b") C)  
 (C ,(singleton-regexp "a") D) (C ,(singleton-regexp "b") E)))  
 `((S ,(singleton-regexp "a") A)  
 (S ,(singleton-regexp "b") B)  
 (A  
 ,(concat-regexp (singleton-regexp "a") (singleton-regexp "a"))  
 D)  
 (A  
 ,(concat-regexp (singleton-regexp "a") (singleton-regexp "b"))  
 E)  
 (B ,(concat-regexp (singleton-regexp "b") (singleton-regexp "a"))  
 (B ,(concat-regexp (singleton-regexp "b") (singleton-regexp "b"))))))

# FSA and Regular Expressions

- ```
;; state dgraph → dgraph Purpose: Rip out given node from given graph
;; Assume: Given node in given graph and g has no multiple edges between nodes
(define (rip-out-node n g)
  (let*
    [(non (filter (λ (r) (and (not (eq? (third r) n))(not (eq? (first r) n)))) g))
```

# FSA and Regular Expressions

- ```
;; state dgraph → dgraph Purpose: Rip out given node from given graph
;; Assume: Given node in given graph and g has no multiple edges between nodes
(define (rip-out-node n g)
  (let*
    [(non (filter (λ (r) (and (not (eq? (third r) n))(not (eq? (first r) n)))) g))
```
- ```
(into-n (filter (λ (r) (and (eq? (third r) n) (not (eq? (first r) n)))) g))
```

# FSA and Regular Expressions

- ```
;; state dgraph → dgraph Purpose: Rip out given node from given graph
;; Assume: Given node in given graph and g has no multiple edges between nodes
(define (rip-out-node n g)
  (let*
    [(non (filter (λ (r) (and (not (eq? (third r) n)) (not (eq? (first r) n)))) g))
     (into-n (filter (λ (r) (and (eq? (third r) n) (not (eq? (first r) n)))) g))
     (outof-n (filter (λ (r) (and (eq? (first r) n) (not (eq? (third r) n)))) g))
```
- ```
(into-n (filter (λ (r) (and (eq? (third r) n) (not (eq? (first r) n)))) g))
```
- ```
(outof-n (filter (λ (r) (and (eq? (first r) n) (not (eq? (third r) n)))) g))
```

# FSA and Regular Expressions

- `;; state dgraph → dgraph` Purpose: Rip out given node from given graph  
`;; Assume: Given node in given graph and g has no multiple edges between nodes`  
`(define (rip-out-node n g)`  
 `(let*`  
 `[(non (filter (λ (r) (and (not (eq? (third r) n))(not (eq? (first r) n)))) g))`
  - `(into-n (filter (λ (r) (and (eq? (third r) n) (not (eq? (first r) n)))) g))`
  - `(outof-n (filter (λ (r) (and (eq? (first r) n) (not (eq? (third r) n)))) g))`
  - `(self-edges (filter (λ (r) (and (eq? (first r) n) (eq? (third r) n)))) g))]`
- `(remove-multiple-edges`  
 `(append`  
 `non`

# FSA and Regular Expressions

- `;; state dgraph → dgraph Purpose: Rip out given node from given graph  
;; Assume: Given node in given graph and g has no multiple edges between nodes  
(define (rip-out-node n g)  
 (let*  
 [(non (filter (λ (r) (and (not (eq? (third r) n)) (not (eq? (first r) n)))) g))`
- `(into-n (filter (λ (r) (and (eq? (third r) n) (not (eq? (first r) n)))) g))`
- `(outof-n (filter (λ (r) (and (eq? (first r) n) (not (eq? (third r) n)))) g))`
- `(self-edges (filter (λ (r) (and (eq? (first r) n) (eq? (third r) n)))) g))]`
- `(remove-multiple-edges  
 (append  
 non  
 (if (not (empty? self-edges))  
 (let [(se (first self-edges))]  
 (append-map  
 (λ (into-edge)  
 (map (λ (outof-edge)  
 (list (first into-edge)  
 (concat-regexp  
 (second into-edge)  
 (concat-regexp (kleenestar-regexp (second se))  
 (second outof-edge)))  
 (third outof-edge)))  
 outof-s-edges))  
 into-s-edges))`

# FSA and Regular Expressions

- `;; state dgraph → dgraph` Purpose: Rip out given node from given graph  
`;; Assume: Given node in given graph and g has no multiple edges between nodes`  
`(define (rip-out-node n g)`  
 `(let*`  
 `[(non (filter (λ (r) (and (not (eq? (third r) n)) (not (eq? (first r) n)))) g))`  
  
 `(into-n (filter (λ (r) (and (eq? (third r) n) (not (eq? (first r) n)))) g))`  
 `(outof-n (filter (λ (r) (and (eq? (first r) n) (not (eq? (third r) n)))) g))`  
 `(self-edges (filter (λ (r) (and (eq? (first r) n) (eq? (third r) n)))) g))]`  
  
 `(remove-multiple-edges`  
 `(append`  
 `non`  
 `(if (not (empty? self-edges))`  
 `(let [(se (first self-edges))]`  
 `(append-map`  
 `(λ (into-edge)`  
 `(map (λ (outof-edge)`  
 `(list (first into-edge)`  
 `(concat-regexp`  
 `(second into-edge)`  
 `(concat-regexp (kleenestar-regexp (second se))`  
 `(second outof-edge)))`  
 `(third outof-edge)))`  
 `outof-s-edges))`  
 `into-s-edges))`  
 `(append-map`  
 `(λ (into-edge)`  
 `(map (λ (outof-edge) (list (first into-edge)`  
 `(concat-regexp (second into-edge)`  
 `(second outof-edge)))`  
 `(third outof-edge)))`  
 `outof-s-edges))`  
 `into-s-edges))))))`

# FSA and Regular Expressions

Regular  
Expressions

Deterministic  
Finite  
Automata

Nondeterministic  
Finite  
Automata

Finite-State  
Automata and  
Regular  
Expressions

Regular  
Grammars

Pumping  
Theorem for  
Regular  
Languages

## Theorem

$(\text{ndfa2regexp } m)$  returns a regular expression for  $L(m)$ .

- The proof of correctness requires proving that all functions return the expected value
- To prove that each function is correct assume that the auxiliary functions are correct
- We start with the main function

# FSA and Regular Expressions

## Theorem

$(ndfa2regexp m)$  returns a regular expression for  $L(m)$ .

- New start and final states are generated
- To  $m$ 's rules  $\epsilon$ -transitions are added from the new start state to  $m$ 's start state and from each of  $m$ 's final states to the new final state

# FSA and Regular Expressions

## Theorem

$(ndfa2regexp m)$  returns a regular expression for  $L(m)$ .

- New start and final states are generated
- To  $m$ 's rules  $\epsilon$ -transitions are added from the new start state to  $m$ 's start state and from each of  $m$ 's final states to the new final state
- By assumption, `make-dgraph` creates the correct initial directed graph
- Multiple edges between any pair of nodes are removed by `remove-multiple-edges`
- All the nodes that represent a state in  $m$  are ripped out by `rip-out-nodes` to create the collapsed graph

# FSA and Regular Expressions

## Theorem

$(ndfa2regexp m)$  returns a regular expression for  $L(m)$ .

- New start and final states are generated
- To  $m$ 's rules  $\epsilon$ -transitions are added from the new start state to  $m$ 's start state and from each of  $m$ 's final states to the new final state
- By assumption, `make-dgraph` creates the correct initial directed graph
- Multiple edges between any pair of nodes are removed by `remove-multiple-edges`
- All the nodes that represent a state in  $m$  are ripped out by `rip-out-nodes` to create the collapsed graph
- Observe that the graph meets the assumptions made by `rip-out-nodes`
- These auxiliary functions, by assuming their correctness, return the correct graph for their given input

# FSA and Regular Expressions

## Theorem

*(ndfa2regexp m) returns a regular expression for  $L(m)$ .*

- New start and final states are generated
- To  $m$ 's rules  $\epsilon$ -transitions are added from the new start state to  $m$ 's start state and from each of  $m$ 's final states to the new final state
- By assumption, `make-dgraph` creates the correct initial directed graph
- Multiple edges between any pair of nodes are removed by `remove-multiple-edges`
- All the nodes that represent a state in  $m$  are ripped out by `rip-out-nodes` to create the collapsed graph
- Observe that the graph meets the assumptions made by `rip-out-nodes`
- These auxiliary functions, by assuming their correctness, return the correct graph for their given input
- The collapsed graph is examined to test if it is empty
- If so, `(null-regexp)` is returned because  $L(m)$  is empty

# FSA and Regular Expressions

## Theorem

*(ndfa2regexp m) returns a regular expression for  $L(m)$ .*

- New start and final states are generated
- To  $m$ 's rules  $\epsilon$ -transitions are added from the new start state to  $m$ 's start state and from each of  $m$ 's final states to the new final state
- By assumption, `make-dgraph` creates the correct initial directed graph
- Multiple edges between any pair of nodes are removed by `remove-multiple-edges`
- All the nodes that represent a state in  $m$  are ripped out by `rip-out-nodes` to create the collapsed graph
- Observe that the graph meets the assumptions made by `rip-out-nodes`
- These auxiliary functions, by assuming their correctness, return the correct graph for their given input
- The collapsed graph is examined to test if it is empty
- If so, `(null-regexp)` is returned because  $L(m)$  is empty
- Otherwise, the only edge's regular expression is returned
- This is correct because this regular expression can generate words on all paths from the new start state to the new final state in the initial directed graph

# FSA and Regular Expressions

## Theorem

*(rip-out-nodes l o n g) returns a dgraph resulting from removing the given nodes from the given graph.*

- By assumption, all the given nodes are in the given graph and the given graph does not have multiple edges between nodes

# FSA and Regular Expressions

## Theorem

*(rip-out-nodes lon g)* returns a dgraph resulting from removing the given nodes from the given graph.

- By assumption, all the given nodes are in the given graph and the given graph does not have multiple edges between nodes
- The given list of nodes is traversed using `foldr` to rip out one node at a time
- Initially, `foldr`'s accumulator is the given graph
- For each node, `foldr` creates a new graph by ripping out the next node using `rip-out-node`

# FSA and Regular Expressions

Regular  
Expressions

Deterministic  
Finite  
Automata

Nondeterministic  
Finite  
Automata

Finite-State  
Automata and  
Regular  
Expressions

Regular  
Grammars

Pumping  
Theorem for  
Regular  
Languages

## Theorem

*(rip-out-nodes long) returns a dgraph resulting from removing the given nodes from the given graph.*

- By assumption, all the given nodes are in the given graph and the given graph does not have multiple edges between nodes
- The given list of nodes is traversed using `foldr` to rip out one node at a time
- Initially, `foldr`'s accumulator is the given graph
- For each node, `foldr` creates a new graph by ripping out the next node using `rip-out-node`
- Observe that the graph returned by `rip-out-node` satisfies the assumptions made by this function
- Therefore, the returned graph may be input to `rip-out-node`
- This auxiliary function, by assumption, is correct
- Thus, `rip-out-nodes` returns the correct directed graph after ripping out all the given nodes

# FSA and Regular Expressions

## Theorem

*(rip-out-node n g) returns a dgraph resulting from removing the given node from the given dgraph.*

- By assumption, the given node is in the given graph

# FSA and Regular Expressions

## Theorem

*(rip-out-node n g)* returns a dgraph resulting from removing the given node from the given dgraph.

- By assumption, the given node is in the given graph
- Extracts four mutually exclusive sets of rules:

<b>non</b>	The set of edges that are not into nor out of n
<b>into-n</b>	The set of edges into n
<b>outof-n</b>	The set of edges out of n
<b>self-edges</b>	The set of edges that are self-loops on n ← at most one

# FSA and Regular Expressions

## Theorem

*(rip-out-node n g)* returns a dgraph resulting from removing the given node from the given dgraph.

- By assumption, the given node is in the given graph
- Extracts four mutually exclusive sets of rules:

**non** The set of edges that are not into nor out of n  
**into-n** The set of edges into n  
**outof-n** The set of edges out of n  
**self-edges** The set of edges that are self-loops on n ← at most one

- If there is a self-loop on n, for each edge, i, in into-n a new edge is created using each edge, o, in outof-n that has the following form:

*(starting state of i*  
*(concat-regexp*  
*regular expression in i*  
*(concat-regexp*  
*(kleenestar-regexp regular expression in only self-edge)*  
*regular expression in o))*  
*destination state of o)*

- This is correct because the regular expression can generate all words that take the machine from the state represented by the starting node of i to the destination state of o

# FSA and Regular Expressions

## Theorem

(rip-out-node n g) returns a dgraph resulting from removing the given node from the given dgraph.

- By assumption, the given node is in the given graph
- Extracts four mutually exclusive sets of rules:

**non** The set of edges that are not into nor out of n

**into-n** The set of edges into n

**outof-n** The set of edges out of n

**self-edges** The set of edges that are self-loops on n ← at most one

- If there is a self-loop on n, for each edge, i, in into-n a new edge is created using each edge, o, in outof-n that has the following form:

(starting state of i

(concat-regexp

regular expression in i

(concat-regexp

(kleenestar-regexp regular expression in only self-edge)

regular expression in o))

destination state of o)

- This is correct because the regular expression can generate all words that take the machine from the state represented by the starting node of i to the destination state of o
- If there is no self-loop on n then for each edge, i, in into-n a new edge is created using each edge, o, in outof-n:

(starting state of i

(concat-regexp

regular expression in i

regular expression in o)

destination state of o)

- This is correct because the regular expression can generate all words that take the machine from the state represented by the starting node of i to the state represented by the destination node of o

# FSA and Regular Expressions

## Theorem

(rip-out-node n g) returns a dgraph resulting from removing the given node from the given dgraph.

- By assumption, the given node is in the given graph
- Extracts four mutually exclusive sets of rules:

**non** The set of edges that are not into nor out of n

**into-n** The set of edges into n

**outof-n** The set of edges out of n

**self-edges** The set of edges that are self-loops on n ← at most one

- If there is a self-loop on n, for each edge, i, in into-n a new edge is created using each edge, o, in outof-n that has the following form:

*(starting state of i*

*(concat-regexp*

*regular expression in i*

*(concat-regexp*

*(kleenestar-regexp regular expression in only self-edge)*

*regular expression in o))*

*destination state of o)*

- This is correct because the regular expression can generate all words that take the machine from the state represented by the starting node of i to the destination state of o
- If there is no self-loop on n then for each edge, i, in into-n a new edge is created using each edge, o, in outof-n:

*(starting state of i*

*(concat-regexp*

*regular expression in i*

*regular expression in o)*

*destination state of o)*

- This is correct because the regular expression can generate all words that take the machine from the state represented by the starting node of i to the state represented by the destination node of o
- The remaining proofs for auxiliary functions are left as exercises

# FSA and Regular Expressions

- HOMEWORK: 17–19
- QUIZ: 20 (due in a week)

# Regular Grammars

- We know that a dfa decides a regular language by reading one symbol at a time
- This suggests that words in the language may be generated one symbol at a time starting with a symbol S

# Regular Grammars

- We know that a dfa decides a regular language by reading one symbol at a time
- This suggests that words in the language may be generated one symbol at a time starting with a symbol S
- We have to be able to generate the empty word because it may be part of a regular language

# Regular Grammars

- We know that a dfa decides a regular language by reading one symbol at a time
- This suggests that words in the language may be generated one symbol at a time starting with a symbol  $S$
- We have to be able to generate the empty word because it may be part of a regular language
- There are three types of rules in a regular grammar:
  - ①  $S$  generates the empty word (i.e.,  $EMP$ )
  - ② Rules that generate an alphabet member
  - ③ Rules that generate a symbol representing the concatenation of a terminal symbol and a symbol representing a syntactic category.
- The members of the alphabet are the terminal symbols
- The symbols representing syntactic categories are the nonterminals

# Regular Grammars

- Formally:

A regular grammar is an instance of  $(\text{make-rg } N \Sigma R S)$

- $N$  is the set of capital letters in the Roman alphabet representing the nonterminal symbols
- $\Sigma$  is the set of lowercase symbols called the alphabet
- $S$  is the starting nonterminal symbol
- $R$  is the set of generating (or production) rules

# Regular Grammars

- Formally:

A regular grammar is an instance of  $(\text{make-rg } N \Sigma R S)$

- $N$  is the set of capital letters in the Roman alphabet representing the nonterminal symbols
- $\Sigma$  is the set of lowercase symbols called the alphabet
- $S$  is the starting nonterminal symbol
- $R$  is the set of generating (or production) rules
- Each production rule contains a nonterminal followed by an arrow and a symbol
- There are only 3 types of production rules:

$S \rightarrow \epsilon$ , where  $S$  is the starting nonterminal and  $S \in N$

$A \rightarrow a$ , where  $A \in N$  and  $a \in \Sigma$

$A \rightarrow aB$ , where  $A, B \in N$  and  $a \in \Sigma$

- Observe that each rule generates one terminal symbol at a time

# Regular Grammars

- Formally:

A regular grammar is an instance of  $(\text{make-rg } N \Sigma R S)$

- $N$  is the set of capital letters in the Roman alphabet representing the nonterminal symbols
- $\Sigma$  is the set of lowercase symbols called the alphabet
- $S$  is the starting nonterminal symbol
- $R$  is the set of generating (or production) rules
- Each production rule contains a nonterminal followed by an arrow and a symbol
- There are only 3 types of production rules:

$S \rightarrow \epsilon$ , where  $S$  is the starting nonterminal and  $S \in N$

$A \rightarrow a$ , where  $A \in N$  and  $a \in \Sigma$

$A \rightarrow aB$ , where  $A, B \in N$  and  $a \in \Sigma$

- Observe that each rule generates one terminal symbol at a time
- The language of a grammar  $G$  is denoted as  $L(G)$
- It contains all the words that can be generated using  $G$

# Regular Grammars

- Grammar observers:

(grammar-nts g): Returns a list of g's nonterminal symbols.

(grammar-sigma g): Returns a list of g's terminal symbols.

(grammar-rules g): Returns a list of g's production rules

(grammar-start g): Returns g's starting nonterminal.

(grammar-type g): Returns a symbol for g's grammar type.

(grammar-derive g w): If the given word, w, is in the language of the given grammar then a derivation for w is returned.  
Otherwise, a string indicating that w is not in the language of the given grammar is returned.

- A derivation consists of 1 or more derivation steps
- A derivation step is denoted by  $\rightarrow$
- One or more derivation steps is denoted by  $\rightarrow^+$ .

# Regular Grammars

- Grammar observers:

(grammar-nts g): Returns a list of g's nonterminal symbols.

(grammar-sigma g): Returns a list of g's terminal symbols.

(grammar-rules g): Returns a list of g's production rules

(grammar-start g): Returns g's starting nonterminal.

(grammar-type g): Returns a symbol for g's grammar type.

(grammar-derive g w): If the given word, w, is in the language of the given grammar then a derivation for w is returned.  
Otherwise, a string indicating that w is not in the language of the given grammar is returned.

- A derivation consists of 1 or more derivation steps
- A derivation step is denoted by  $\rightarrow$
- One or more derivation steps is denoted by  $\rightarrow^+$ .
- Testing functions:

(grammar-both-derive g1 g2 w): Tests if both of the given grammars derive the given word.

(grammar-testequiv g1 g2 [natnum]): Tests if the given grammars derive 100 (or the optional number of) randomly generated words. If all tests give the same result true is returned. Otherwise, a list of words that produce different results is returned.

(grammar-test g1 [natnum]): Tests the given grammar with 100 (or the optional number of) randomly generated words. A list of pairs containing a word and the result of attempting to derive the word are returned.

# Regular Grammars

- The Design Recipe for 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

# Regular Grammars

- Design a regular grammar for:

$$L = \{w \mid \text{the number of } a\text{s in } w \text{ is a multiple of 3}\}$$

# Regular Grammars

- Design a regular grammar for:  
 $L = \{w \mid \text{the number of } a\text{s in } w \text{ is a multiple of 3}\}$
- Name: MULT3-as
- $\Sigma = \{a, b\}$

# Regular Grammars

- Syntactic categories
- $S = \text{words where the number of } a \text{ is } 3n, \text{ starting nonterminal}$

# Regular Grammars

- Syntactic categories
  - $S = \text{words where the number of } a \text{ is } 3n, \text{ starting nonterminal}$
  - If  $a$  is generated by  $S$  then  $3n+2$  as needed
- 
- $B = \text{words where the number of } a \text{ is } 3n + 2$

# Regular Grammars

- Syntactic categories
- $S = \text{words where the number of } a \text{ is } 3n, \text{ starting nonterminal}$
- If  $a$  is generated by  $S$  then  $3n+2$  as needed
- $B = \text{words where the number of } a \text{ is } 3n + 2$
- If  $a$  is generated by  $B$  then  $3n+1$  as needed
- $C = \text{words where the number of } a \text{ is } 3n + 1$
- If  $C$  generates an  $a$  then a word with  $3n$  as must be generated

# Regular Grammars

- Syntactic categories
- $S = \text{words where the number of } a \text{ is } 3n, \text{ starting nonterminal}$
- If  $a$  is generated by  $S$  then  $3n+2$  as needed
- $(\text{list } 'S \text{ ARROW } \text{EMP})$   
 $(\text{list } 'S \text{ ARROW } 'aB)$   
 $(\text{list } 'S \text{ ARROW } 'bS)$
- $B = \text{words where the number of } a \text{ is } 3n + 2$
- If  $a$  is generated by  $B$  then  $3n+1$  as needed
  
- $C = \text{words where the number of } a \text{ is } 3n + 1$
- If  $C$  generates an  $a$  then a word with  $3n$  as must be generated

# Regular Grammars

- Syntactic categories
- $S = \text{words where the number of } a \text{ is } 3n, \text{ starting nonterminal}$
- If  $a$  is generated by  $S$  then  $3n+2$  as needed
- $(\text{list } 'S \text{ ARROW } \text{EMP})$   
 $(\text{list } 'S \text{ ARROW } 'aB)$   
 $(\text{list } 'S \text{ ARROW } 'bS)$
- $B = \text{words where the number of } a \text{ is } 3n + 2$
- If  $a$  is generated by  $B$  then  $3n+1$  as needed
- $(\text{list } 'B \text{ ARROW } 'aC)$   
 $(\text{list } 'B \text{ ARROW } 'bB)$
- $C = \text{words where the number of } a \text{ is } 3n + 1$
- If  $C$  generates an  $a$  then a word with  $3n$  as must be generated

# Regular Grammars

- Syntactic categories
- $S = \text{words where the number of } a \text{ is } 3n, \text{ starting nonterminal}$
- If  $a$  is generated by  $S$  then  $3n+2$  as needed
- $(\text{list } 'S \text{ ARROW } \text{EMP})$   
 $(\text{list } 'S \text{ ARROW } 'aB)$   
 $(\text{list } 'S \text{ ARROW } 'bS)$
- $B = \text{words where the number of } a \text{ is } 3n + 2$
- If  $a$  is generated by  $B$  then  $3n+1$  as needed
- $(\text{list } 'B \text{ ARROW } 'aC)$   
 $(\text{list } 'B \text{ ARROW } 'bB)$
- $C = \text{words where the number of } a \text{ is } 3n + 1$
- If  $C$  generates an  $a$  then a word with  $3n$  as must be generated
- **Typo in the book.**  
 $(C \text{ ,ARROW } aS)$   
 $(C \text{ ,ARROW } bC)$

# Regular Grammars

- Unit Tests

```
;; Tests for MULT3-as
(check-equal? (grammar-derive MULT3-as '(b b a b b))
               "(b b a b b) is not in L(G).")
(check-equal? (grammar-derive MULT3-as '(b b a b b a))
               "(b b a b b a) is not in L(G).")
(check-equal?
  (grammar-derive MULT3-as '(b b a b a b a a b))
  "(b b a b a b a a b) is not in L(G).")
(check-equal? (grammar-derive MULT3-as '())
               '(S -> ε))
(check-equal?
  (grammar-derive MULT3-as '(a a a))
  '(S -> aB -> aaC -> aaaS -> aaa))
(check-equal?
  (grammar-derive MULT3-as '(b b a a b a b b))
  '(S -> bS -> bbS -> bbaB -> bbaaC -> bbaabC -> bbaabaS
    -> bbaababS -> bbaababbS -> bbaababb))
```

# Regular Grammars

- Implementation

```
(define MULT3-as (make-rg '(S B C)
                           '(a b)
                           `((S ,ARROW ,EMP)
                             (S ,ARROW aB)
                             (S ,ARROW bS)
                             (B ,ARROW ac)
                             (B ,ARROW bB)
                             (C ,ARROW aS)
                             (C ,ARROW bC)))
                           'S))
```

# Regular Grammars

- Implementation

```
(define MULT3-as (make-rg '(S B C)
                           '(a b)
                           `((S ,ARROW ,EMP)
                             (S ,ARROW aB)
                             (S ,ARROW bS)
                             (B ,ARROW ac)
                             (B ,ARROW bB)
                             (C ,ARROW aS)
                             (C ,ARROW bC)))
                           'S))
```

- Run the tests

# Regular Grammars

- > (grammar-test MULT3-as 10)  
'(((b a a a) (S -> bS -> baB -> baAC -> baaaS -> baaa))  
((b a a a a) "(b a a a a) is not in L(G).")  
((a a) "(a a) is not in L(G).")  
((a b a b) "(a b a b) is not in L(G).")  
((a b) "(a b) is not in L(G).")  
(() (S -> ε))  
((b) (S -> bS -> b))  
((b b) (S -> bS -> bbS -> bb))  
((b a b a) "(b a b a) is not in L(G).")  
((b a a b a b)  
    (S  
        ->  
        bS  
        ->  
        baB  
        ->  
        baAC  
        ->  
        baabC  
        ->  
        baababS  
        ->  
        baabababS  
        ->  
        baabab)))

# Regular Grammars

- HOMEWORK: 1–5

# Regular Grammars

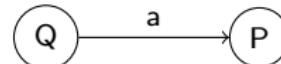
## Theorem

$L \text{ is regular} \Leftrightarrow L \text{ is generated by a regular grammar.}$

- Must be able to build a rg from a dfa
- Must be able to build a dfa from an rg

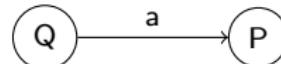
# Regular Grammars

- Building a regular grammar,  $R$ , from,  $D$ , a dfa such that  $L(R) = L(D)$
- D's transition rules always consume an element of the alphabet:



# Regular Grammars

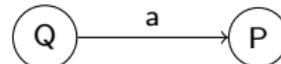
- Building a regular grammar,  $R$ , from,  $D$ , a dfa such that  $L(R) = L(D)$
- D's transition rules always consume an element of the alphabet:



- $R$  must have a production rule to produce such an  $a$
- $a$  must be part of the right hand side of the production rule

# Regular Grammars

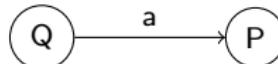
- Building a regular grammar,  $R$ , from,  $D$ , a dfa such that  $L(R) = L(D)$
- $D$ 's transition rules always consume an element of the alphabet:



- $R$  must have a production rule to produce such an  $a$
- $a$  must be part of the right hand side of the production rule
- We do not know what  $D$  may consume after reaching  $P$ , but whatever it is it must be generated by production rules obtained from transition rules starting at  $P$

# Regular Grammars

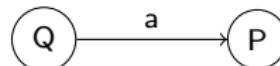
- Building a regular grammar,  $R$ , from,  $D$ , a dfa such that  $L(R) = L(D)$
- $D$ 's transition rules always consume an element of the alphabet:



- $R$  must have a production rule to produce such an  $a$
- $a$  must be part of the right hand side of the production rule
- We do not know what  $D$  may consume after reaching  $P$ , but whatever it is it must be generated by production rules obtained from transition rules starting at  $P$
- $a$  and anything read after reaching  $P$  must be generated

# Regular Grammars

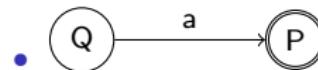
- Building a regular grammar, R, from, D, a dfa such that  $L(R) = L(D)$
- D's transition rules always consume an element of the alphabet:



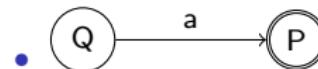
- R must have a production rule to produce such an a
- a must be part of the right hand side of the production rule
- We do not know what D may consume after reaching P, but whatever it is it must be generated by production rules obtained from transition rules starting at P
- a and anything read after reaching P must be generated
- The states of D are the nonterminals of the regular grammar
- From Q an a and whatever is produced by P must be produced:

(list 'Q ARROW 'aP)

# Regular Grammars



# Regular Grammars

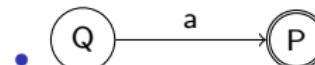


- Two rules when P is a final state

(list 'Q ARROW 'a)

(list 'Q ARROW 'aP)

# Regular Grammars



- Two rules when P is a final state
  - (list 'Q ARROW 'a)
  - (list 'Q ARROW 'aP)
- D's starting state, S, is a final state:  
(list 'S ARROW EMP)

# Regular Grammars

- `;; (listof dfa-rule) (listof state) → (listof rg-rule)`  
;; Purpose: Generate production rules for the given  
;; dfa-rules and the given final states  
`(define (mk-prod-rules mrules mfinals)`

# Regular Grammars

- ```
;; (listof dfa-rule) (listof state) → (listof rg-rule)
;; Purpose: Generate production rules for the given
;;           dfa-rules and the given final states
(define (mk-prod-rules mrules mfinals)
```
- ```
;; Tests for mk-prod-rules
(check-equal? (mk-prod-rules '() '(F G)) '())
(check-equal? (mk-prod-rules '((S a F) (S b R)
                                (R a G) (R b R)
                                (G a G) (G b G))
                                '(F G))
               '((S -> a) (S -> aF) (S -> bR)
                 (R -> a) (R -> aG) (R -> bR)
                 (G -> a) (G -> aG) (G -> b) (G -> bG)))
```

## Regular Grammars

- ```
;; (listof dfa-rule) (listof state) → (listof rg-rule)
;; Purpose: Generate production rules for the given
;;           dfa-rules and the given final states
(define (mk-prod-rules mrules mfinals)

• (append-map
  (λ (r)
    (if (not (member (third r) mfinals))
        (list (list (first r) ARROW (los->symbol (rest r))))))

• ;; Tests for mk-prod-rules
(check-equal? (mk-prod-rules '() '(F G)) '())
(check-equal? (mk-prod-rules '((S a F) (S b R)
                                (R a G) (R b R)
                                (G a G) (G b G))
                                '(F G))
              '((S -> a) (S -> aF) (S -> bR)
                (R -> a) (R -> aG) (R -> bR)
                (G -> a) (G -> aG) (G -> b) (G -> bG)))
```

# Regular Grammars

- `;; (listof dfa-rule) (listof state) → (listof rg-rule)`  
;; Purpose: Generate production rules for the given  
;; dfa-rules and the given final states  
`(define (mk-prod-rules mrules mfinals))`
- `(append-map`  
 `(λ (r)`  
 `(if (not (member (third r) mfinals))`  
 `(list (list (first r) ARROW (los->symbol (rest r))))`
- `(list (list (first r) ARROW (second r))`  
 `(list (first r) ARROW (los->symbol (rest r))))))`  
 `mrules))`
- `;; Tests for mk-prod-rules`  
`(check-equal? (mk-prod-rules '() '(F G)) '())`  
`(check-equal? (mk-prod-rules '((S a F) (S b R)`  
 `(R a G) (R b R)`  
 `(G a G) (G b G))`  
 `'(F G))`  
`' ((S -> a) (S -> aF) (S -> bR)`  
 `(R -> a) (R -> aG) (R -> bR)`  
 `(G -> a) (G -> aG) (G -> b) (G -> bG)))`

# Regular Grammars

- `;; dfa → rg`  
`;; Purpose: Build a rg for the language of the given dfa`  
`;; Assume: dfa states are represented by a single capital letter`  
`(define (dfa2rg m)`

# Regular Grammars

- `;; dfa → rg`  
`;; Purpose: Build a rg for the language of the given dfa`  
`;; Assume: dfa states are represented by a single capital letter`  
`(define (dfa2rg m)`

- `;; Tests for dfa2rg`  
`(define SIGMA*-rg (dfa2rg SIGMA*))`  
`(define EA-OB-rg (dfa2rg EVEN-A-ODD-B))`  
  
`(check-equal? (eq? (last (grammar-derive SIGMA*-rg '()) ) EMP)`  
`(eq? (sm-apply SIGMA* '()) 'accept))`  
`(check-equal? (eq? (last (grammar-derive SIGMA*-rg '(a b c)))`  
`(los->symbol '(a b c)))`  
`(eq? (sm-apply SIGMA* '(a b c)) 'accept))`  
`(check-equal? (eq? (last (grammar-derive SIGMA*-rg '(c c a b a c)))`  
`(los->symbol '(c c a b a c)))`  
`(eq? (sm-apply SIGMA* '(c c a b a c)) 'accept))`  
  
`(check-equal? (string? (grammar-derive EA-OB-rg '(a b)))`  
`(eq? (sm-apply EVEN-A-ODD-B '(a b)) 'reject))`  
`(check-equal? (string? (grammar-derive EA-OB-rg '(a a b a)))`  
`(eq? (sm-apply EVEN-A-ODD-B '(a a b a)) 'reject))`  
`(check-equal? (eq? (last (grammar-derive EA-OB-rg '(b)))`  
`(los->symbol '(b)))`  
`(eq? (sm-apply EVEN-A-ODD-B '(b)) 'accept))`  
`(check-equal? (eq? (last (grammar-derive EA-OB-rg '(b a a b b)))`  
`(los->symbol '(b a a b b))))`

# Regular Grammars

- ```
;; dfa → rg
;; Purpose: Build a rg for the language of the given dfa
;; Assume: dfa states are represented by a single capital letter
(define (dfa2rg m)
  (let* [(nts (sm-states m))
         (sigma (sm-sigma m))
         (startnt (sm-start m))
```
- ```
;; Tests for dfa2rg
(define SIGMA*-rg (dfa2rg SIGMA*))
(define EA-OB-rg (dfa2rg EVEN-A-ODD-B))

(check-equal? (eq? (last (grammar-derive SIGMA*-rg '())) 'EMP)
               (eq? (sm-apply SIGMA* '()) 'accept))
(check-equal? (eq? (last (grammar-derive SIGMA*-rg '(a b c)))
                   (los->symbol '(a b c)))
               (eq? (sm-apply SIGMA* '(a b c)) 'accept))
(check-equal? (eq? (last (grammar-derive SIGMA*-rg '(c c a b a c)))
                   (los->symbol '(c c a b a c)))
               (eq? (sm-apply SIGMA* '(c c a b a c)) 'accept))

(check-equal? (string? (grammar-derive EA-OB-rg '(a b)))
               (eq? (sm-apply EVEN-A-ODD-B '(a b)) 'reject))
(check-equal? (string? (grammar-derive EA-OB-rg '(a a b a)))
               (eq? (sm-apply EVEN-A-ODD-B '(a a b a)) 'reject))
(check-equal? (eq? (last (grammar-derive EA-OB-rg '(b)))
                   (los->symbol '(b)))
               (eq? (sm-apply EVEN-A-ODD-B '(b)) 'accept))
(check-equal? (eq? (last (grammar-derive EA-OB-rg '(b a a b b)))
                   (los->symbol '(b a a b b))))
```

# Regular Grammars

- ```
;; dfa → rg
;; Purpose: Build a rg for the language of the given dfa
;; Assume: dfa states are represented by a single capital letter
(define (dfa2rg m)
```
- ```
(let* [(nts (sm-states m))
         (sigma (sm-sigma m))
         (startnt (sm-start m))]
```
- ```
(prules (if (member (sm-start m) (sm-finals m))
                  (cons (list (sm-start m) ARROW EMP)
                        (mk-prod-rules (sm-rules m) (sm-finals m)))
                  (mk-prod-rules (sm-rules m) (sm-finals m))))]
```
- ```
;; Tests for dfa2rg
(define SIGMA*-rg (dfa2rg SIGMA*))
(define EA-OB-rg (dfa2rg EVEN-A-ODD-B))

(check-equal? (eq? (last (grammar-derive SIGMA*-rg '())) EMP)
              (eq? (sm-apply SIGMA* '()) 'accept))
(check-equal? (eq? (last (grammar-derive SIGMA*-rg '(a b c)))
                  (los->symbol '(a b c)))
              (eq? (sm-apply SIGMA* '(a b c)) 'accept))
(check-equal? (eq? (last (grammar-derive SIGMA*-rg '(c c a b a c)))
                  (los->symbol '(c c a b a c)))
              (eq? (sm-apply SIGMA* '(c c a b a c)) 'accept))

(check-equal? (string? (grammar-derive EA-OB-rg '(a b)))
              (eq? (sm-apply EVEN-A-ODD-B '(a b)) 'reject))
(check-equal? (string? (grammar-derive EA-OB-rg '(a a b a)))
              (eq? (sm-apply EVEN-A-ODD-B '(a a b a)) 'reject))
(check-equal? (eq? (last (grammar-derive EA-OB-rg '(b)))
                  (los->symbol '(b)))
              (eq? (sm-apply EVEN-A-ODD-B '(b)) 'accept))
(check-equal? (eq? (last (grammar-derive EA-OB-rg '(b a a b b)))
                  (los->symbol '(b a a b b)))
```

# Regular Grammars

- `;; dfa → rg`  
`;; Purpose: Build a rg for the language of the given dfa`  
`;; Assume: dfa states are represented by a single capital letter`  
`(define (dfa2rg m)`
- `(let* [(nts (sm-states m))`  
`(sigma (sm-sigma m))`  
`(startnt (sm-start m))`
- `(prules (if (member (sm-start m) (sm-finals m))`  
`(cons (list (sm-start m) ARROW EMP)`  
`(mk-prod-rules (sm-rules m) (sm-finals m)))`  
`(mk-prod-rules (sm-rules m) (sm-finals m))))])`
- `(make-rg nts sigma prules startnt)))`
- `;; Tests for dfa2rg`  
`(define SIGMA*-rg (dfa2rg SIGMA*))`  
`(define EA-OB-rg (dfa2rg EVEN-A-ODD-B))`  
  
`(check-equal? (eq? (last (grammar-derive SIGMA*-rg '()) ) EMP)`  
`(eq? (sm-apply SIGMA* '()) 'accept))`  
`(check-equal? (eq? (last (grammar-derive SIGMA*-rg '(a b c)))`  
`(los->symbol '(a b c)))`  
`(eq? (sm-apply SIGMA* '(a b c)) 'accept))`  
`(check-equal? (eq? (last (grammar-derive SIGMA*-rg '(c c a b a c)))`  
`(los->symbol '(c c a b a c)))`  
`(eq? (sm-apply SIGMA* '(c c a b a c)) 'accept))`  
  
`(check-equal? (string? (grammar-derive EA-OB-rg '(a b)))`  
`(eq? (sm-apply EVEN-A-ODD-B '(a b)) 'reject))`  
`(check-equal? (string? (grammar-derive EA-OB-rg '(a a b a)))`  
`(eq? (sm-apply EVEN-A-ODD-B '(a a b a)) 'reject))`  
`(check-equal? (eq? (last (grammar-derive EA-OB-rg '(b)))`  
`(los->symbol '(b)))`  
`(eq? (sm-apply EVEN-A-ODD-B '(b)) 'accept))`  
`(check-equal? (eq? (last (grammar-derive EA-OB-rg '(b a a b b)))`  
`(los->symbol '(b a a b b))))`

# Regular Grammars

## Theorem

$L$  is regular  $\Rightarrow L$  is generated by a regular grammar.

- $M = (\text{make-dfa } S \Sigma Z F \delta)$  decides  $L$
- $G = (\text{dfa2rg } M)$
- Show that  $w \in L(M) \Leftrightarrow w \in L(G) \wedge w \notin L(M) \Leftrightarrow w \notin L(G)$

# Regular Grammars

## Theorem

$L$  is regular  $\Rightarrow L$  is generated by a regular grammar.

- $M = (\text{make-dfa } S \Sigma Z F \delta)$  decides  $L$
- $G = (\text{dfa2rg } M)$
- Show that  $w \in L(M) \Leftrightarrow w \in L(G) \wedge w \notin L(M) \Leftrightarrow w \notin L(G)$
- $w \in L(M) \Leftrightarrow w \in L(G)$

# Regular Grammars

## Theorem

$L$  is regular  $\Rightarrow L$  is generated by a regular grammar.

- $M = (\text{make-dfa } S \Sigma Z F \delta)$  decides  $L$
- $G = (\text{dfa2rg } M)$
- Show that  $w \in L(M) \Leftrightarrow w \in L(G) \wedge w \notin L(M) \Leftrightarrow w \notin L(G)$
- $w \in L(M) \Leftrightarrow w \in L(G)$
- ( $\Rightarrow$ ) Assume  $w \in L(M)$
- This means that  $M$  performs the following computation:  $(w Z) \vdash^* M (() K)$ , where  $K \in F$

# Regular Grammars

## Theorem

$L \text{ is regular} \Rightarrow L \text{ is generated by a regular grammar.}$

- $M = (\text{make-dfa } S \Sigma Z F \delta)$  decides  $L$
- $G = (\text{dfa2rg } M)$
- Show that  $w \in L(M) \Leftrightarrow w \in L(G) \wedge w \notin L(M) \Leftrightarrow w \notin L(G)$
- $w \in L(M) \Leftrightarrow w \in L(G)$
- ( $\Rightarrow$ ) Assume  $w \in L(M)$
- This means that  $M$  performs the following computation:  $(w Z) \vdash^* M (() K)$ , where  $K \in F$
- By construction of  $G$ , for every rule,  $(C \ a\ D)$ , in  $M$  there is a corresponding production rule  $(C \rightarrow aD)$
- In addition, if  $D$  is a final state:  $(C \rightarrow a)$

# Regular Grammars

## Theorem

$L \text{ is regular} \Rightarrow L \text{ is generated by a regular grammar.}$

- $M = (\text{make-dfa } S \Sigma Z F \delta)$  decides  $L$
- $G = (\text{dfa2rg } M)$
- Show that  $w \in L(M) \Leftrightarrow w \in L(G) \wedge w \notin L(M) \Leftrightarrow w \notin L(G)$
- $w \in L(M) \Leftrightarrow w \in L(G)$
- ( $\Rightarrow$ ) Assume  $w \in L(M)$
- This means that  $M$  performs the following computation:  $(w Z) \vdash^* M (( ) K)$ , where  $K \in F$
- By construction of  $G$ , for every rule,  $(C \ a\ D)$ , in  $M$  there is a corresponding production rule  $(C \rightarrow aD)$
- In addition, if  $D$  is a final state:  $(C \rightarrow a)$
- This means that every element consumed by  $M$ 's computation is generated by a derivation using  $G$  simulating  $M$ 's execution
- If  $M$  moves to a final state and accepts then the derivation only generates a terminal symbol that is the same symbol consumed by  $M$

# Regular Grammars

## Theorem

$L \text{ is regular} \Rightarrow L \text{ is generated by a regular grammar.}$

- $M = (\text{make-dfa } S \Sigma Z F \delta)$  decides  $L$
- $G = (\text{dfa2rg } M)$
- Show that  $w \in L(M) \Leftrightarrow w \in L(G) \wedge w \notin L(M) \Leftrightarrow w \notin L(G)$
- $w \in L(M) \Leftrightarrow w \in L(G)$
- ( $\Rightarrow$ ) Assume  $w \in L(M)$
- This means that  $M$  performs the following computation:  $(w Z) \vdash^* M (( ) K)$ , where  $K \in F$
- By construction of  $G$ , for every rule,  $(C \ a\ D)$ , in  $M$  there is a corresponding production rule  $(C \rightarrow aD)$
- In addition, if  $D$  is a final state:  $(C \rightarrow a)$
- This means that every element consumed by  $M$ 's computation is generated by a derivation using  $G$  simulating  $M$ 's execution
- If  $M$  moves to a final state and accepts then the derivation only generates a terminal symbol that is the same symbol consumed by  $M$
- Finally, if nothing is consumed and  $M$  accepts then the derivation using  $G$  generates  $\text{EMP}$
- This means there is a derivation using  $G$  that generates a word accepted by  $M$
- Thus,  $w \in L(G)$

# Regular Grammars

## Theorem

$L \text{ is regular} \Rightarrow L \text{ is generated by a regular grammar.}$

- $M = (\text{make-dfa } S \Sigma Z F \delta)$  decides  $L$
- $G = (\text{dfa2rg } M)$
- Show that  $w \in L(M) \Leftrightarrow w \in L(G) \wedge w \notin L(M) \Leftrightarrow w \notin L(G)$
- $w \in L(M) \Leftrightarrow w \in L(G)$
- ( $\Rightarrow$ ) Assume  $w \in L(M)$
- This means that  $M$  performs the following computation:  $(w Z) \vdash^* M (( ) K)$ , where  $K \in F$
- By construction of  $G$ , for every rule,  $(C \ a\ D)$ , in  $M$  there is a corresponding production rule  $(C \rightarrow aD)$
- In addition, if  $D$  is a final state:  $(C \rightarrow a)$
- This means that every element consumed by  $M$ 's computation is generated by a derivation using  $G$  simulating  $M$ 's execution
- If  $M$  moves to a final state and accepts then the derivation only generates a terminal symbol that is the same symbol consumed by  $M$
- Finally, if nothing is consumed and  $M$  accepts then the derivation using  $G$  generates  $\text{EMP}$
- This means there is a derivation using  $G$  that generates a word accepted by  $M$
- Thus,  $w \in L(G)$
- ( $\Leftarrow$ ) Assume  $w \in L(G)$

# Regular Grammars

## Theorem

$L \text{ is regular} \Rightarrow L \text{ is generated by a regular grammar.}$

- $M = (\text{make-dfa } S \Sigma Z F \delta)$  decides  $L$
- $G = (\text{dfa2rg } M)$
- Show that  $w \in L(M) \Leftrightarrow w \in L(G) \wedge w \notin L(M) \Leftrightarrow w \notin L(G)$
- $w \in L(M) \Leftrightarrow w \in L(G)$
- ( $\Rightarrow$ ) Assume  $w \in L(M)$
- This means that  $M$  performs the following computation:  $(w Z) \vdash^* M (( ) K)$ , where  $K \in F$
- By construction of  $G$ , for every rule,  $(C \ a\ D)$ , in  $M$  there is a corresponding production rule  $(C \rightarrow aD)$
- In addition, if  $D$  is a final state:  $(C \rightarrow a)$
- This means that every element consumed by  $M$ 's computation is generated by a derivation using  $G$  simulating  $M$ 's execution
- If  $M$  moves to a final state and accepts then the derivation only generates a terminal symbol that is the same symbol consumed by  $M$
- Finally, if nothing is consumed and  $M$  accepts then the derivation using  $G$  generates  $\text{EMP}$
- This means there is a derivation using  $G$  that generates a word accepted by  $M$
- Thus,  $w \in L(G)$
- ( $\Leftarrow$ ) Assume  $w \in L(G)$
- This means that  $G$  can derive  $w$

# Regular Grammars

## Theorem

$L$  is regular  $\Rightarrow L$  is generated by a regular grammar.

- $M = (\text{make-dfa } S \Sigma Z F \delta)$  decides  $L$
- $G = (\text{dfa2rg } M)$
- Show that  $w \in L(M) \Leftrightarrow w \in L(G) \wedge w \notin L(M) \Leftrightarrow w \notin L(G)$
- $w \in L(M) \Leftrightarrow w \in L(G)$
- ( $\Rightarrow$ ) Assume  $w \in L(M)$
- This means that  $M$  performs the following computation:  $(w Z) \vdash^* M (( ) K)$ , where  $K \in F$
- By construction of  $G$ , for every rule,  $(C \ a\ D)$ , in  $M$  there is a corresponding production rule  $(C \rightarrow aD)$
- In addition, if  $D$  is a final state:  $(C \rightarrow a)$
- This means that every element consumed by  $M$ 's computation is generated by a derivation using  $G$  simulating  $M$ 's execution
- If  $M$  moves to a final state and accepts then the derivation only generates a terminal symbol that is the same symbol consumed by  $M$
- Finally, if nothing is consumed and  $M$  accepts then the derivation using  $G$  generates  $\text{EMP}$
- This means there is a derivation using  $G$  that generates a word accepted by  $M$
- Thus,  $w \in L(G)$
- ( $\Leftarrow$ ) Assume  $w \in L(G)$
- This means that  $G$  can derive  $w$
- By construction of  $G$ , every derivation step of  $G$  simulates a step taken by  $M$  in a computation that consumes  $w$  and reaches a final state

# Regular Grammars

## Theorem

$L$  is regular  $\Rightarrow L$  is generated by a regular grammar.

- $M = (\text{make-dfa } S \Sigma Z F \delta)$  decides  $L$
- $G = (\text{dfa2rg } M)$
- Show that  $w \in L(M) \Leftrightarrow w \in L(G) \wedge w \notin L(M) \Leftrightarrow w \notin L(G)$
- $w \in L(M) \Leftrightarrow w \in L(G)$
- ( $\Rightarrow$ ) Assume  $w \in L(M)$
- This means that  $M$  performs the following computation:  $(w Z) \vdash^* M (( ) K)$ , where  $K \in F$
- By construction of  $G$ , for every rule,  $(C \ a\ D)$ , in  $M$  there is a corresponding production rule  $(C \rightarrow aD)$
- In addition, if  $D$  is a final state:  $(C \rightarrow a)$
- This means that every element consumed by  $M$ 's computation is generated by a derivation using  $G$  simulating  $M$ 's execution
- If  $M$  moves to a final state and accepts then the derivation only generates a terminal symbol that is the same symbol consumed by  $M$
- Finally, if nothing is consumed and  $M$  accepts then the derivation using  $G$  generates  $\text{EMP}$
- This means there is a derivation using  $G$  that generates a word accepted by  $M$
- Thus,  $w \in L(G)$
- ( $\Leftarrow$ ) Assume  $w \in L(G)$
- This means that  $G$  can derive  $w$
- By construction of  $G$ , every derivation step of  $G$  simulates a step taken by  $M$  in a computation that consumes  $w$  and reaches a final state
- Therefore,  $w \in L(M)$

# Regular Grammars

## Theorem

$L \text{ is regular} \Rightarrow L \text{ is generated by a regular grammar.}$

- $M = (\text{make-dfa } S \Sigma Z F \delta)$  decides  $L$
- $G = (\text{dfa2rg } M)$
- $w \notin L(M) \Leftrightarrow w \notin L(G)$

# Regular Grammars

## Theorem

$L$  is regular  $\Rightarrow L$  is generated by a regular grammar.

- $M = (\text{make-dfa } S \Sigma Z F \delta)$  decides  $L$
- $G = (\text{dfa2rg } M)$
- $w \notin L(M) \Leftrightarrow w \notin L(G)$
- ( $\Rightarrow$ ) Assume  $w \notin L(M)$ .

# Regular Grammars

## Theorem

$L$  is regular  $\Rightarrow L$  is generated by a regular grammar.

- $M = (\text{make-dfa } S \Sigma Z F \delta)$  decides  $L$
- $G = (\text{dfa2rg } M)$
- $w \notin L(M) \Leftrightarrow w \notin L(G)$
- ( $\Rightarrow$ ) Assume  $w \notin L(M)$ .
- This means that  $M$ 's computation on  $w$  is  $(w Z) \vdash^* M (( ) K)$ , where  $K \notin F$

# Regular Grammars

## Theorem

$L$  is regular  $\Rightarrow L$  is generated by a regular grammar.

- $M = (\text{make-dfa } S \Sigma Z F \delta)$  decides  $L$
- $G = (\text{dfa2rg } M)$
- $w \notin L(M) \Leftrightarrow w \notin L(G)$
- ( $\Rightarrow$ ) Assume  $w \notin L(M)$ .
- This means that  $M$ 's computation on  $w$  is  $(w Z) \vdash^* M (() K)$ , where  $K \notin F$
- By construction of  $G$ , for every rule,  $(C \ a\ D)$ , in  $M$  there is a corresponding production rule  $(C \rightarrow aD)$

# Regular Grammars

## Theorem

$L \text{ is regular} \Rightarrow L \text{ is generated by a regular grammar.}$

- $M = (\text{make-dfa } S \Sigma Z F \delta)$  decides  $L$
- $G = (\text{dfa2rg } M)$
- $w \notin L(M) \Leftrightarrow w \notin L(G)$
- ( $\Rightarrow$ ) Assume  $w \notin L(M)$ .
- This means that  $M$ 's computation on  $w$  is  $(w Z) \vdash^* M (() K)$ , where  $K \notin F$
- By construction of  $G$ , for every rule,  $(C \ a D)$ , in  $M$  there is a corresponding production rule  $(C \rightarrow aD)$
- That is, a derivation using  $G$  simulates the computation done by  $M$

# Regular Grammars

## Theorem

$L \text{ is regular} \Rightarrow L \text{ is generated by a regular grammar.}$

- $M = (\text{make-dfa } S \Sigma Z F \delta)$  decides  $L$
- $G = (\text{dfa2rg } M)$
- $w \notin L(M) \Leftrightarrow w \notin L(G)$
- ( $\Rightarrow$ ) Assume  $w \notin L(M)$ .
- This means that  $M$ 's computation on  $w$  is  $(w \mid Z) \vdash^* M ((\mid K), \text{ where } K \notin F)$
- By construction of  $G$ , for every rule,  $(C \mid a D)$ , in  $M$  there is a corresponding production rule  $(C \rightarrow aD)$
- That is, a derivation using  $G$  simulates the computation done by  $M$
- Let  $(L \mid K)$  be the last transition rule  $M$  uses during its computation on  $w$

# Regular Grammars

## Theorem

$L \text{ is regular} \Rightarrow L \text{ is generated by a regular grammar.}$

- $M = (\text{make-dfa } S \Sigma Z F \delta)$  decides  $L$
- $G = (\text{dfa2rg } M)$
- $w \notin L(M) \Leftrightarrow w \notin L(G)$
- ( $\Rightarrow$ ) Assume  $w \notin L(M)$ .
- This means that  $M$ 's computation on  $w$  is  $(w \mid Z) \vdash^* M ((\mid) K)$ , where  $K \notin F$
- By construction of  $G$ , for every rule,  $(C \mid a D)$ , in  $M$  there is a corresponding production rule  $(C \rightarrow aD)$
- That is, a derivation using  $G$  simulates the computation done by  $M$
- Let  $(L \mid a K)$  be the last transition rule  $M$  uses during its computation on  $w$
- This means that the last production rule  $G$  uses is  $(L \rightarrow aK)$

# Regular Grammars

## Theorem

$L \text{ is regular} \Rightarrow L \text{ is generated by a regular grammar.}$

- $M = (\text{make-dfa } S \Sigma Z F \delta)$  decides  $L$
- $G = (\text{dfa2rg } M)$
- $w \notin L(M) \Leftrightarrow w \notin L(G)$
- ( $\Rightarrow$ ) Assume  $w \notin L(M)$ .
- This means that  $M$ 's computation on  $w$  is  $(w Z) \vdash^* M (( ) K)$ , where  $K \notin F$
- By construction of  $G$ , for every rule,  $(C \ a D)$ , in  $M$  there is a corresponding production rule  $(C \rightarrow aD)$
- That is, a derivation using  $G$  simulates the computation done by  $M$
- Let  $(L \ a K)$  be the last transition rule  $M$  uses during its computation on  $w$
- This means that the last production rule  $G$  uses is  $(L \rightarrow aK)$
- Observe that the symbol produced still represents the existence of a nonterminal and, thus, cannot equal  $w$

# Regular Grammars

## Theorem

$L$  is regular  $\Rightarrow L$  is generated by a regular grammar.

- $M = (\text{make-dfa } S \Sigma Z F \delta)$  decides  $L$
- $G = (\text{dfa2rg } M)$
- $w \notin L(M) \Leftrightarrow w \notin L(G)$
- ( $\Rightarrow$ ) Assume  $w \notin L(M)$ .
- This means that  $M$ 's computation on  $w$  is  $(w \mid Z) \vdash^* M ((\mid) K)$ , where  $K \notin F$
- By construction of  $G$ , for every rule,  $(C \mid aD)$ , in  $M$  there is a corresponding production rule  $(C \rightarrow aD)$
- That is, a derivation using  $G$  simulates the computation done by  $M$
- Let  $(L \mid aK)$  be the last transition rule  $M$  uses during its computation on  $w$
- This means that the last production rule  $G$  uses is  $(L \rightarrow aK)$
- Observe that the symbol produced still represents the existence of a nonterminal and, thus, cannot equal  $w$
- Given that  $G$  is simulating a dfa and  $K$  is not a final state of  $M$ , there is no other possible set of derivation steps on  $w$ .
- Thus,  $w \notin L(G)$

# Regular Grammars

## Theorem

$L$  is regular  $\Rightarrow L$  is generated by a regular grammar.

- $M = (\text{make-dfa } S \Sigma Z F \delta)$  decides  $L$
- $G = (\text{dfa2rg } M)$
- $w \notin L(M) \Leftrightarrow w \notin L(G)$
- ( $\Rightarrow$ ) Assume  $w \notin L(M)$ .
- This means that  $M$ 's computation on  $w$  is  $(w \mid Z) \vdash^* M ((\mid) K)$ , where  $K \notin F$
- By construction of  $G$ , for every rule,  $(C \mid aD)$ , in  $M$  there is a corresponding production rule  $(C \rightarrow aD)$
- That is, a derivation using  $G$  simulates the computation done by  $M$
- Let  $(L \mid aK)$  be the last transition rule  $M$  uses during its computation on  $w$
- This means that the last production rule  $G$  uses is  $(L \rightarrow aK)$
- Observe that the symbol produced still represents the existence of a nonterminal and, thus, cannot equal  $w$
- Given that  $G$  is simulating a dfa and  $K$  is not a final state of  $M$ , there is no other possible set of derivation steps on  $w$ .
- Thus,  $w \notin L(G)$
- ( $\Leftarrow$ ) Assume  $w \notin L(G)$

# Regular Grammars

## Theorem

$L$  is regular  $\Rightarrow L$  is generated by a regular grammar.

- $M = (\text{make-dfa } S \Sigma Z F \delta)$  decides  $L$
- $G = (\text{dfa2rg } M)$
- $w \notin L(M) \Leftrightarrow w \notin L(G)$
- ( $\Rightarrow$ ) Assume  $w \notin L(M)$ .
- This means that  $M$ 's computation on  $w$  is  $(w \mid Z) \vdash^* M ((\mid K), \text{ where } K \notin F)$
- By construction of  $G$ , for every rule,  $(C \mid aD)$ , in  $M$  there is a corresponding production rule  $(C \rightarrow aD)$
- That is, a derivation using  $G$  simulates the computation done by  $M$
- Let  $(L \mid aK)$  be the last transition rule  $M$  uses during its computation on  $w$
- This means that the last production rule  $G$  uses is  $(L \rightarrow aK)$
- Observe that the symbol produced still represents the existence of a nonterminal and, thus, cannot equal  $w$
- Given that  $G$  is simulating a dfa and  $K$  is not a final state of  $M$ , there is no other possible set of derivation steps on  $w$ .
- Thus,  $w \notin L(G)$
- ( $\Leftarrow$ ) Assume  $w \notin L(G)$
- This means that there is no derivation of  $w$  using  $G$

# Regular Grammars

## Theorem

$L$  is regular  $\Rightarrow L$  is generated by a regular grammar.

- $M = (\text{make-dfa } S \Sigma Z F \delta)$  decides  $L$
- $G = (\text{dfa2rg } M)$
- $w \notin L(M) \Leftrightarrow w \notin L(G)$
- ( $\Rightarrow$ ) Assume  $w \notin L(M)$ .
- This means that  $M$ 's computation on  $w$  is  $(w Z) \vdash^* M (() K)$ , where  $K \notin F$
- By construction of  $G$ , for every rule,  $(C \ a D)$ , in  $M$  there is a corresponding production rule  $(C \rightarrow aD)$
- That is, a derivation using  $G$  simulates the computation done by  $M$
- Let  $(L \ a K)$  be the last transition rule  $M$  uses during its computation on  $w$
- This means that the last production rule  $G$  uses is  $(L \rightarrow aK)$
- Observe that the symbol produced still represents the existence of a nonterminal and, thus, cannot equal  $w$
- Given that  $G$  is simulating a dfa and  $K$  is not a final state of  $M$ , there is no other possible set of derivation steps on  $w$ .
- Thus,  $w \notin L(G)$
- ( $\Leftarrow$ ) Assume  $w \notin L(G)$
- This means that there is no derivation of  $w$  using  $G$
- By construction, a derivation using  $G$  is simulating the computation of  $M$  on  $w$

# Regular Grammars

## Theorem

$L$  is regular  $\Rightarrow L$  is generated by a regular grammar.

- $M = (\text{make-dfa } S \Sigma Z F \delta)$  decides  $L$
- $G = (\text{dfa2rg } M)$
- $w \notin L(M) \Leftrightarrow w \notin L(G)$
- ( $\Rightarrow$ ) Assume  $w \notin L(M)$ .
- This means that  $M$ 's computation on  $w$  is  $(w \mid Z) \vdash^* M ((\mid K), \text{ where } K \notin F)$
- By construction of  $G$ , for every rule,  $(C \mid aD)$ , in  $M$  there is a corresponding production rule  $(C \rightarrow aD)$
- That is, a derivation using  $G$  simulates the computation done by  $M$
- Let  $(L \mid aK)$  be the last transition rule  $M$  uses during its computation on  $w$
- This means that the last production rule  $G$  uses is  $(L \rightarrow aK)$
- Observe that the symbol produced still represents the existence of a nonterminal and, thus, cannot equal  $w$
- Given that  $G$  is simulating a dfa and  $K$  is not a final state of  $M$ , there is no other possible set of derivation steps on  $w$ .
- Thus,  $w \notin L(G)$
- ( $\Leftarrow$ ) Assume  $w \notin L(G)$
- This means that there is no derivation of  $w$  using  $G$
- By construction, a derivation using  $G$  is simulating the computation of  $M$  on  $w$
- Given that  $M$  is deterministic there is only one computation possible

# Regular Grammars

## Theorem

$L$  is regular  $\Rightarrow L$  is generated by a regular grammar.

- $M = (\text{make-dfa } S \Sigma Z F \delta)$  decides  $L$
- $G = (\text{dfa2rg } M)$
- $w \notin L(M) \Leftrightarrow w \notin L(G)$
- ( $\Rightarrow$ ) Assume  $w \notin L(M)$ .
- This means that  $M$ 's computation on  $w$  is  $(w \mid Z) \vdash^* M ((\mid K), \text{ where } K \notin F)$
- By construction of  $G$ , for every rule,  $(C \mid aD)$ , in  $M$  there is a corresponding production rule  $(C \rightarrow aD)$
- That is, a derivation using  $G$  simulates the computation done by  $M$
- Let  $(L \mid aK)$  be the last transition rule  $M$  uses during its computation on  $w$
- This means that the last production rule  $G$  uses is  $(L \rightarrow aK)$
- Observe that the symbol produced still represents the existence of a nonterminal and, thus, cannot equal  $w$
- Given that  $G$  is simulating a dfa and  $K$  is not a final state of  $M$ , there is no other possible set of derivation steps on  $w$ .
- Thus,  $w \notin L(G)$
- ( $\Leftarrow$ ) Assume  $w \notin L(G)$
- This means that there is no derivation of  $w$  using  $G$
- By construction, a derivation using  $G$  is simulating the computation of  $M$  on  $w$
- Given that  $M$  is deterministic there is only one computation possible
- Thus,  $w \notin L(M)$

# Regular Grammars

- Convert  $G=(\text{make-rg } N \ \Sigma P \ S)$  into,  $M$ , a finite-state machine that decides  $L(G)$
- The finite-state machine shall simulate a derivation

# Regular Grammars

- $G$ 's nonterminals are states in  $M$
- $\Sigma$  is  $M$ 's alphabet
- $S$  is  $M$ 's starting state

# Regular Grammars

- $G$ 's nonterminals are states in  $M$
- $\Sigma$  is  $M$ 's alphabet
- $S$  is  $M$ 's starting state
- If  $M$  is to simulate a derivation under  $G$  then  $M$  must accept when a simple production rule is used
- A simple production rule:

$$I \rightarrow i, \text{ where } i \in \{\epsilon\} \cup \Sigma \quad I \in \mathbb{N}$$

# Regular Grammars

- $G$ 's nonterminals are states in  $M$
- $\Sigma$  is  $M$ 's alphabet
- $S$  is  $M$ 's starting state
- If  $M$  is to simulate a derivation under  $G$  then  $M$  must accept when a simple production rule is used
- A simple production rule:

$$I \rightarrow i, \text{ where } i \in \{\epsilon\} \cup \Sigma \wedge I \in \mathbb{N}$$

- $M$  shall have,  $Z$ , a unique state that is also the single final state:

$$I \rightarrow i \dashrightarrow (I \ i \ Z)$$

# Regular Grammars

- $G$ 's nonterminals are states in  $M$
- $\Sigma$  is  $M$ 's alphabet
- $S$  is  $M$ 's starting state
- If  $M$  is to simulate a derivation under  $G$  then  $M$  must accept when a simple production rule is used
- A simple production rule:

$$I \rightarrow i, \text{ where } i \in \{\epsilon\} \cup \Sigma \wedge I \in \mathbb{N}$$

- $M$  shall have,  $Z$ , a unique state that is also the single final state:  
 $I \rightarrow i \dashrightarrow (I \ i \ Z)$
- $M$  must also have a transition rule for every compound production rule in  $G$
- A compound production rule is defined as follows:

$$I \rightarrow iJ, \text{ where } i \in \Sigma \wedge I, J \in \mathbb{N}$$

# Regular Grammars

- G's nonterminals are states in M
- $\Sigma$  is M's alphabet
- S is M's starting state
- If M is to simulate a derivation under G then M must accept when a simple production rule is used
- A simple production rule:

$$I \rightarrow i, \text{ where } i \in \{\epsilon\} \cup \Sigma \wedge I \in \mathbb{N}$$

- M shall have, Z, a unique state that is also the single final state:  
$$I \rightarrow i \dashrightarrow (I \ i \ Z)$$
- M must also have a transition rule for every compound production rule in G
- A compound production rule is defined as follows:

$$I \rightarrow iJ, \text{ where } i \in \Sigma \wedge I, J \in \mathbb{N}$$

- To simulate a compound production rule:  
$$I \rightarrow iJ \dashrightarrow (I \ i \ J)$$

# Regular Grammars

- Consider:

```
(define a*Ub*-rg    ;; L = a* U b*
  (make-rg '(S A B) '(a b)
            `((S ,ARROW ,EMP) (S ,ARROW aA) (S ,ARROW bB)
              (S ,ARROW a) (S ,ARROW b) (A ,ARROW aA)
              (A ,ARROW a) (B ,ARROW bB) (B ,ARROW b))
            'S))
```

# Regular Grammars

- Consider:

```
(define a*Ub*-rg    ;; L = a* U b*
  (make-rg '(S A B) '(a b)
            `((S ,ARROW ,EMP) (S ,ARROW aA) (S ,ARROW bB)
              (S ,ARROW a) (S ,ARROW b) (A ,ARROW aA)
              (A ,ARROW a) (B ,ARROW bB) (B ,ARROW b))
            'S))
```

- $S = (\text{cons } 'Z \ (S \ A \ B))$   
 $\Sigma = '(a \ b)$   
 $A = 'S$   
 $F = (\text{list } 'Z)$

# Regular Grammars

- Consider:

```
(define a*Ub*-rg    ;; L = a* U b*
  (make-rg '(S A B) '(a b)
            `((S ,ARROW ,EMP) (S ,ARROW aA) (S ,ARROW bB)
              (S ,ARROW a) (S ,ARROW b) (A ,ARROW aA)
              (A ,ARROW a) (B ,ARROW bB) (B ,ARROW b))
            'S))
```

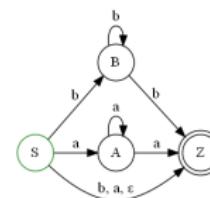
- $S = (\text{cons } 'Z \ (S \ A \ B))$   
 $\Sigma = '(a \ b)$   
 $A = 'S$   
 $F = (\text{list } 'Z)$
- $I \rightarrow i \text{ rules}$        $I \rightarrow iJ \text{ rules}$   
 $(S \rightarrow \epsilon)$                    $(S \rightarrow aA)$   
 $(S \rightarrow a)$                    $(S \rightarrow bB)$   
 $(S \rightarrow b)$                    $(A \rightarrow aA)$   
 $(A \rightarrow a)$                    $(B \rightarrow bB)$   
 $(B \rightarrow b)$

# Regular Grammars

- Consider:

```
(define a*Ub*-rg    ;; L = a* U b*
  (make-rg '(S A B) '(a b)
            `((S ,ARROW ,EMP) (S ,ARROW aA) (S ,ARROW bB)
              (S ,ARROW a) (S ,ARROW b) (A ,ARROW aA)
              (A ,ARROW a) (B ,ARROW bB) (B ,ARROW b))
            'S))
```

- $S = (\text{cons } 'Z \ (S \ A \ B))$
- $\Sigma = '(a \ b)$
- $A = 'S$
- $F = (\text{list } 'Z)$
- $I \rightarrow i \text{ rules}$ 
  - $(S \rightarrow \epsilon)$
  - $(S \rightarrow a)$
  - $(S \rightarrow b)$
  - $(A \rightarrow a)$
  - $(B \rightarrow b)$
- $I \rightarrow iJ \text{ rules}$ 
  - $(S \rightarrow aA)$
  - $(S \rightarrow bB)$
  - $(A \rightarrow aA)$
  - $(B \rightarrow bB)$



- It is not difficult to see that it decides  $L = a^* \cup b^*$

# Regular Grammars

- To test the constructor:

```
;; Sample rg
;; L = (a U b U c)*
(define SIGMA*-rg (dfa2rg SIGMA*))
```

```
;; L = w | w has an even number of a and an odd number of b
(define EA-OB-rg (dfa2rg EVEN-A-ODD-B))
```

```
;; L = a* U b*
(define a*Ub*-rg
  (make-rg
    '(S A B)
    '(a b)
    `((S ,ARROW ,EMP) (S ,ARROW aA) (S ,ARROW bB) (S ,ARROW a)
      (S ,ARROW b) (A ,ARROW aA) (A ,ARROW a) (B ,ARROW bB)
      (B ,ARROW b))
    'S))
```

```
;; L = a aba
(define a-aba-rg
  (make-rg
    '(S A B)
    '(a b)
    `((S ,ARROW a) (S ,ARROW aA) (A ,ARROW bB) (B ,ARROW a))
    'S))
```

# Regular Grammars

- `;; rg → ndfa`  
`;; Purpose: Build a ndfa for the language of the given rg`  
`(define (rg2ndfa rg)`

# Regular Grammars

- ;; Tests for rg2ndfa  
(define SIGMA\*2 (rg2ndfa SIGMA\*-rg)) (define EA-OB (rg2ndfa EA-OB-rg))  
(define a\*Ub\* (rg2ndfa a\*Ub\*-rg)) (define a-aba (rg2ndfa a-aba-rg))  
  
(check-equal? (eq? (sm-apply SIGMA\*2 '()) 'accept)  
                 (eq? (last (grammar-derive SIGMA\*-rg '())) EMP))  
  
      :  
  
(check-equal? (sm-testequiv? SIGMA\* SIGMA\*2) #t)  
(check-equal? (eq? (sm-apply EA-OB '(a b)) 'reject)  
                 (string=? (grammar-derive EA-OB-rg '(a b))  
                          "(a b) is not in L(G)."))  
(check-equal? (eq? (sm-apply EA-OB '(b)) 'accept)  
                 (eq? (last (grammar-derive EA-OB-rg '(b))) (los->symbol '(b))))  
(check-equal? (sm-testequiv? EVEN-A-ODD-B EA-OB) #t)  
  
      :  
  
(check-equal? (eq? (sm-apply a\*Ub\* '(a b)) 'reject)  
                 (string=? (grammar-derive a\*Ub\*-rg '(a b))  
                          "(a b) is not in L(G)."))  
(check-equal? (eq? (sm-apply a\*Ub\* '()) 'accept)  
                 (eq? (last (grammar-derive a\*Ub\*-rg '())) EMP))  
  
      :

# Regular Grammars

- `;; rg → ndfa`

`;; Purpose: Build a ndfa for the language of the given rg`  
`(define (rg2ndfa rg)`  
 `(let* [(final-state (generate-symbol 'Z (grammar-nts rg)))]`

# Regular Grammars

- `;; rg → ndfa`  
;; Purpose: Build a ndfa for the language of the given rg  
`(define (rg2ndfa rg)`  
    `(let* [(final-state (generate-symbol 'Z (grammar-nts rg)))`  
        `(states (cons final-state (grammar-nts rg)))`  
        `(sigma (grammar-sigma rg))`  
        `(start (grammar-start rg))`  
        `(finals (list final-state))`

# Regular Grammars

- `;; rg → ndfa`  
;; Purpose: Build a ndfa for the language of the given rg  
`(define (rg2ndfa rg)`  
    `(let* [(final-state (generate-symbol 'Z (grammar-nts rg)))`  
        `(states (cons final-state (grammar-nts rg)))`  
        `(sigma (grammar-sigma rg))`  
        `(start (grammar-start rg))`  
        `(finals (list final-state))`  
        `(simple-prs (filter`  
            `(λ (pr) (= (length (symbol->fsmlos (third pr))) 1))`  
            `(grammar-rules rg)))`  
        `(cmpnd-prs`  
            `(filter (λ (pr) (= (length (symbol->fsmlos (third pr))) 2))`  
            `(grammar-rules rg)))`

# Regular Grammars

- `;; rg → ndfa`  
`;; Purpose: Build a ndfa for the language of the given rg`  
`(define (rg2ndfa rg)`  
 `(let* [(final-state (generate-symbol 'Z (grammar-nts rg)))`  
 `(states (cons final-state (grammar-nts rg))))`  
 `(sigma (grammar-sigma rg))`  
 `(start (grammar-start rg))`  
 `(finals (list final-state)))`  
 `(simple-prs (filter`  
 `(λ (pr) (= (length (symbol->fsmlos (third pr))) 1))`  
 `(grammar-rules rg)))`  
 `(cmpnd-prs`  
 `(filter (λ (pr) (= (length (symbol->fsmlos (third pr))) 2))`  
 `(grammar-rules rg)))`  
 `(rules (append`  
 `(map (λ (spr)`  
 `(list (first spr) (third spr) final-state))`  
 `simple-prs)`  
 `(map (λ (pr)`  
 `(let [(rhs (symbol->fsmlos (third pr)))]`  
 `(list (first pr) (first rhs) (second rhs))))`  
 `cmpnd-prs))))]`

# Regular Grammars

- `;; rg → ndfa`  
`;; Purpose: Build a ndfa for the language of the given rg`  
`(define (rg2ndfa rg)`  
    `(let* [(final-state (generate-symbol 'Z (grammar-nts rg)))`  
    `(states (cons final-state (grammar-nts rg))))`  
    `(sigma (grammar-sigma rg))`  
    `(start (grammar-start rg))`  
    `(finals (list final-state))`  
    `(simple-prs (filter`  
        `(λ (pr) (= (length (symbol->fsmlos (third pr))) 1))`  
        `(grammar-rules rg)))`  
    `(cmpnd-prs`  
        `(filter (λ (pr) (= (length (symbol->fsmlos (third pr))) 2))`  
        `(grammar-rules rg)))`  
    `(rules (append`  
        `(map (λ (spr)`  
            `(list (first spr) (third spr) final-state))`  
            `simple-prs)`  
        `(map (λ (pr)`  
            `(let [[(rhs (symbol->fsmlos (third pr)))]`  
                `(list (first pr) (first rhs) (second rhs))))`  
                `cmpnd-prs)))]`  
    `)`  
    `(make-ndfa states sigma start finals rules)))`

# Regular Grammars

## Lemma

$S \xrightarrow{+} w \Leftrightarrow ((a_1 \dots a_n) A) \vdash^+ (() Q)$ , where  $Q=Z$  if  $w$  ends with a terminal symbol and  $Q \in N$  if  $w$  ends with a nonterminal

- ( $\Rightarrow$ ) Assume  $S \xrightarrow{+} w$ .
- We must show that  $(a_1 \dots a_n A) \vdash^+ (() Q)$ , where  $Q=Z$  if  $w$  ends with a terminal symbol and  $Q \in N$  if  $w$  ends with a nonterminal

# Regular Grammars

## Lemma

$S \xrightarrow{+} w \Leftrightarrow ((a_1 \dots a_n) A) \vdash^+ (() Q)$ , where  $Q=Z$  if  $w$  ends with a terminal symbol and  $Q \in N$  if  $w$  ends with a nonterminal

- ( $\Rightarrow$ ) Assume  $S \xrightarrow{+} w$ .
- We must show that  $(a_1 \dots a_n A) \vdash^+ (() Q)$ , where  $Q=Z$  if  $w$  ends with a terminal symbol and  $Q \in N$  if  $w$  ends with a nonterminal
- Induction on,  $n$ , the number of steps in the derivation

# Regular Grammars

## Lemma

$S \xrightarrow{+} w \Leftrightarrow ((a_1 \dots a_n) A) \vdash^+ (() Q)$ , where  $Q=Z$  if  $w$  ends with a terminal symbol and  $Q \in N$  if  $w$  ends with a nonterminal

- ( $\Rightarrow$ ) Assume  $S \xrightarrow{+} w$ .
- We must show that  $(a_1 \dots a_n A) \vdash^+ (() Q)$ , where  $Q=Z$  if  $w$  ends with a terminal symbol and  $Q \in N$  if  $w$  ends with a nonterminal
- Induction on,  $n$ , the number of steps in the derivation
- Base Case:  $n=1$
- The derivation uses only a single production rule
- There are two cases:
- If it is a simple production rule,  $(S \rightarrow a)$ , then  $w=a$ . By construction of  $M$ , we have that  $(S \xrightarrow{} Z) \in \delta$ . Therefore,  $(a A) \vdash^+ (() Z) = (() Q)$

# Regular Grammars

## Lemma

$S \xrightarrow{+} w \Leftrightarrow ((a_1 \dots a_n) A) \vdash^+ (() Q)$ , where  $Q=Z$  if  $w$  ends with a terminal symbol and  $Q \in N$  if  $w$  ends with a nonterminal

- $(\Rightarrow)$  Assume  $S \xrightarrow{+} w$ .
- We must show that  $(a_1 \dots a_n A) \vdash^+ (() Q)$ , where  $Q=Z$  if  $w$  ends with a terminal symbol and  $Q \in N$  if  $w$  ends with a nonterminal
- Induction on,  $n$ , the number of steps in the derivation
- Base Case:  $n=1$
- The derivation uses only a single production rule
- There are two cases:
- If it is a simple production rule,  $(S \rightarrow a)$ , then  $w=a$ . By construction of  $M$ , we have that  $(S a Z) \in \delta$ . Therefore,  $(a A) \vdash^+ (() Z) = (() Q)$
- If it is a compound production rule,  $(S \rightarrow aK)$ , then  $w=aK$
- By construction of  $M$ , we have that  $(S a K) \in \delta$
- Therefore,  $(a A) \vdash^+ (() K) = (() Q)$

# Regular Grammars

## Lemma

$S \xrightarrow{+} w \Leftrightarrow ((a_1 \dots a_n) A) \vdash^+ (() Q)$ , where  $Q=Z$  if  $w$  ends with a terminal symbol and  $Q \in N$  if  $w$  ends with a nonterminal

- ( $\Rightarrow$ ) Assume  $S \xrightarrow{+} w$ .
- We must show that  $(a_1 \dots a_n A) \vdash^+ (() Q)$ , where  $Q=Z$  if  $w$  ends with a terminal symbol and  $Q \in N$  if  $w$  ends with a nonterminal
- Induction on,  $n$ , the number of steps in the derivation
- Base Case:  $n=1$
- The derivation uses only a single production rule
- There are two cases:
- If it is a simple production rule,  $(S \rightarrow a)$ , then  $w=a$ . By construction of  $M$ , we have that  $(S a Z) \in \delta$ . Therefore,  $(a A) \vdash^+ (() Z) = (() Q)$
- If it is a compound production rule,  $(S \rightarrow aK)$ , then  $w=aK$
- By construction of  $M$ , we have that  $(S a K) \in \delta$
- Therefore,  $(a A) \vdash^+ (() K) = (() Q)$
- Inductive Step:
- Assume:  $S \xrightarrow{+} w \Rightarrow (a_1 \dots a_k A) \vdash^+ (() Q)$ , where  $Q=Z$  if  $w$  ends with a terminal symbol and  $Q \in N$  if  $w$  ends with a nonterminal, for  $n=k$ .
- Show:  $S \xrightarrow{+} w \Rightarrow (a_1 \dots a_{k+1} A) \vdash^+ (() Q)$ , where  $Q=Z$  if  $w$  ends with a terminal symbol and  $Q \in N$  if  $w$  ends with a nonterminal, for  $n=k+1$

# Regular Grammars

## Lemma

$S \xrightarrow{+} w \Leftrightarrow ((a_1 \dots a_n) A) \vdash^+ (() Q)$ , where  $Q=Z$  if  $w$  ends with a terminal symbol and  $Q \in N$  if  $w$  ends with a nonterminal

- ( $\Rightarrow$ ) Assume  $S \xrightarrow{+} w$ .
- We must show that  $(a_1 \dots a_n A) \vdash^+ (() Q)$ , where  $Q=Z$  if  $w$  ends with a terminal symbol and  $Q \in N$  if  $w$  ends with a nonterminal
- Induction on,  $n$ , the number of steps in the derivation
- Base Case:  $n=1$
- The derivation uses only a single production rule
- There are two cases:
- If it is a simple production rule,  $(S \rightarrow a)$ , then  $w=a$ . By construction of  $M$ , we have that  $(S a Z) \in \delta$ . Therefore,  $(a A) \vdash^+ (() Z) = (() Q)$
- If it is a compound production rule,  $(S \rightarrow aK)$ , then  $w=aK$
- By construction of  $M$ , we have that  $(S a K) \in \delta$
- Therefore,  $(a A) \vdash^+ (() K) = (() Q)$
- Inductive Step:
- Assume:  $S \xrightarrow{+} w \Rightarrow (a_1 \dots a_k A) \vdash^+ (() Q)$ , where  $Q=Z$  if  $w$  ends with a terminal symbol and  $Q \in N$  if  $w$  ends with a nonterminal, for  $n=k$ .
- Show:  $S \xrightarrow{+} w \Rightarrow (a_1 \dots a_{k+1} A) \vdash^+ (() Q)$ , where  $Q=Z$  if  $w$  ends with a terminal symbol and  $Q \in N$  if  $w$  ends with a nonterminal, for  $n=k+1$
- Assume  $S \xrightarrow{+} w$  for  $n=k+1$

# Regular Grammars

## Lemma

$S \rightarrow^+ w \Leftrightarrow ((a_1 \dots a_n) A) \vdash^+ (() Q)$ , where  $Q=Z$  if  $w$  ends with a terminal symbol and  $Q \in N$  if  $w$  ends with a nonterminal

- ( $\Rightarrow$ ) Assume  $S \rightarrow^+ w$ .
- We must show that  $(a_1 \dots a_n A) \vdash^+ (() Q)$ , where  $Q=Z$  if  $w$  ends with a terminal symbol and  $Q \in N$  if  $w$  ends with a nonterminal
- Induction on,  $n$ , the number of steps in the derivation
- Base Case:  $n=1$ 
  - The derivation uses only a single production rule
  - There are two cases:
    - If it is a simple production rule,  $(S \rightarrow a)$ , then  $w=a$ . By construction of  $M$ , we have that  $(S a Z) \in \delta$ . Therefore,  $(a A) \vdash^+ (() Z) = (() Q)$
    - If it is a compound production rule,  $(S \rightarrow aK)$ , then  $w=aK$
  - By construction of  $M$ , we have that  $(S a K) \in \delta$
  - Therefore,  $(a A) \vdash^+ (() K) = (() Q)$
- Inductive Step:
  - Assume:  $S \rightarrow^+ w \Rightarrow (a_1 \dots a_k A) \vdash^+ (() Q)$ , where  $Q=Z$  if  $w$  ends with a terminal symbol and  $Q \in N$  if  $w$  ends with a nonterminal, for  $n=k$ .
  - Show:  $S \rightarrow^+ w \Rightarrow (a_1 \dots a_{k+1} A) \vdash^+ (() Q)$ , where  $Q=Z$  if  $w$  ends with a terminal symbol and  $Q \in N$  if  $w$  ends with a nonterminal, for  $n=k+1$
  - Assume  $S \rightarrow^+ w$  for  $n=k+1$
  - Given that  $k \geq 1$ ,  $k+1 > 1$ . This means that the derivation of  $w$  is either:
    - $S \rightarrow \dots \rightarrow a_1 \dots a_k U \rightarrow a_1 \dots a_k a_{k+1}$ , where  $U \in N$
    - $S \rightarrow \dots \rightarrow a_1 \dots a_k U \rightarrow a_1 \dots a_k a_{k+1} V$ , where  $U, V \in N$

# Regular Grammars

## Lemma

$S \rightarrow^+ w \Leftrightarrow ((a_1 \dots a_n) A) \vdash^+ (() Q)$ , where  $Q=Z$  if  $w$  ends with a terminal symbol and  $Q \in N$  if  $w$  ends with a nonterminal

- ( $\Rightarrow$ ) Assume  $S \rightarrow^+ w$ .
- We must show that  $(a_1 \dots a_n A) \vdash^+ (() Q)$ , where  $Q=Z$  if  $w$  ends with a terminal symbol and  $Q \in N$  if  $w$  ends with a nonterminal
- Induction on,  $n$ , the number of steps in the derivation
- Base Case:  $n=1$ 
  - The derivation uses only a single production rule
  - There are two cases:
    - If it is a simple production rule,  $(S \rightarrow a)$ , then  $w=a$ . By construction of  $M$ , we have that  $(S a Z) \in \delta$ . Therefore,  $(a A) \vdash^+ (() Z) = (() Q)$
    - If it is a compound production rule,  $(S \rightarrow aK)$ , then  $w=aK$
  - By construction of  $M$ , we have that  $(S a K) \in \delta$
  - Therefore,  $(a A) \vdash^+ (() K) = (() Q)$
- Inductive Step:
  - Assume:  $S \rightarrow^+ w \Rightarrow (a_1 \dots a_k A) \vdash^+ (() Q)$ , where  $Q=Z$  if  $w$  ends with a terminal symbol and  $Q \in N$  if  $w$  ends with a nonterminal, for  $n=k$ .
  - Show:  $S \rightarrow^+ w \Rightarrow (a_1 \dots a_{k+1} A) \vdash^+ (() Q)$ , where  $Q=Z$  if  $w$  ends with a terminal symbol and  $Q \in N$  if  $w$  ends with a nonterminal, for  $n=k+1$
  - Assume  $S \rightarrow^+ w$  for  $n=k+1$
  - Given that  $k \geq 1$ ,  $k+1 > 1$ . This means that the derivation of  $w$  is either:
    - $S \rightarrow \dots \rightarrow a_1 \dots a_k U \rightarrow a_1 \dots a_k a_{k+1}$ , where  $U \in N$
    - $S \rightarrow \dots \rightarrow a_1 \dots a_k U \rightarrow a_1 \dots a_k a_{k+1} V$ , where  $U, V \in N$
  - By inductive hypothesis, we have:  
 $((a_1 \dots a_k a_{k+1}) A) \vdash^* ((a_{k+1}) U)$

# Regular Grammars

## Lemma

$S \rightarrow^+ w \Leftrightarrow ((a_1 \dots a_n) A) \vdash^+ (() Q)$ , where  $Q=Z$  if  $w$  ends with a terminal symbol and  $Q \in N$  if  $w$  ends with a nonterminal

- ( $\Rightarrow$ ) Assume  $S \rightarrow^+ w$ .
- We must show that  $(a_1 \dots a_n A) \vdash^+ (() Q)$ , where  $Q=Z$  if  $w$  ends with a terminal symbol and  $Q \in N$  if  $w$  ends with a nonterminal
- Induction on,  $n$ , the number of steps in the derivation
- Base Case:  $n=1$ 
  - The derivation uses only a single production rule
  - There are two cases:
    - If it is a simple production rule,  $(S \rightarrow a)$ , then  $w=a$ . By construction of  $M$ , we have that  $(S a Z) \in \delta$ . Therefore,  $(a A) \vdash^+ (() Z) = (() Q)$
    - If it is a compound production rule,  $(S \rightarrow aK)$ , then  $w=aK$
  - By construction of  $M$ , we have that  $(S a K) \in \delta$
  - Therefore,  $(a A) \vdash^+ (() K) = (() Q)$
- Inductive Step:
  - Assume:  $S \rightarrow^+ w \Rightarrow (a_1 \dots a_k A) \vdash^+ (() Q)$ , where  $Q=Z$  if  $w$  ends with a terminal symbol and  $Q \in N$  if  $w$  ends with a nonterminal, for  $n=k$ .
  - Show:  $S \rightarrow^+ w \Rightarrow (a_1 \dots a_{k+1} A) \vdash^+ (() Q)$ , where  $Q=Z$  if  $w$  ends with a terminal symbol and  $Q \in N$  if  $w$  ends with a nonterminal, for  $n=k+1$
  - Assume  $S \rightarrow^+ w$  for  $n=k+1$
  - Given that  $k \geq 1$ ,  $k+1 > 1$ . This means that the derivation of  $w$  is either:
    - $S \rightarrow \dots \rightarrow a_1 \dots a_k U \rightarrow a_1 \dots a_k a_{k+1}$ , where  $U \in N$
    - $S \rightarrow \dots \rightarrow a_1 \dots a_k U \rightarrow a_1 \dots a_k a_{k+1} V$ , where  $U, V \in N$
  - By inductive hypothesis, we have:  
 $((a_1 \dots a_k a_{k+1}) A) \vdash^* ((a_{k+1}) U)$
  - If the last production rule used in the derivation is a simple production rule,  $(U \rightarrow a_{k+1})$ , then by construction of  $M$ ,  $(U a_{k+1} Z) \in \delta$
  - Therefore,  $((a_1 \dots a_k a_{k+1}) A) \vdash^* (() Z) = (() Q)$ .

# Regular Grammars

## Lemma

$S \rightarrow^+ w \Leftrightarrow ((a_1 \dots a_n) A) \vdash^+ (() Q)$ , where  $Q=Z$  if  $w$  ends with a terminal symbol and  $Q \in N$  if  $w$  ends with a nonterminal

- ( $\Rightarrow$ ) Assume  $S \rightarrow^+ w$ .
- We must show that  $(a_1 \dots a_n A) \vdash^+ (() Q)$ , where  $Q=Z$  if  $w$  ends with a terminal symbol and  $Q \in N$  if  $w$  ends with a nonterminal
- Induction on,  $n$ , the number of steps in the derivation
- Base Case:  $n=1$ 
  - The derivation uses only a single production rule
  - There are two cases:
    - If it is a simple production rule,  $(S \rightarrow a)$ , then  $w=a$ . By construction of  $M$ , we have that  $(S a Z) \in \delta$ . Therefore,  $(a A) \vdash^+ (() Z) = (() Q)$
    - If it is a compound production rule,  $(S \rightarrow aK)$ , then  $w=aK$
  - By construction of  $M$ , we have that  $(S a K) \in \delta$
  - Therefore,  $(a A) \vdash^+ (() K) = (() Q)$
- Inductive Step:
  - Assume:  $S \rightarrow^+ w \Rightarrow (a_1 \dots a_k A) \vdash^+ (() Q)$ , where  $Q=Z$  if  $w$  ends with a terminal symbol and  $Q \in N$  if  $w$  ends with a nonterminal, for  $n=k$ .
  - Show:  $S \rightarrow^+ w \Rightarrow (a_1 \dots a_{k+1} A) \vdash^+ (() Q)$ , where  $Q=Z$  if  $w$  ends with a terminal symbol and  $Q \in N$  if  $w$  ends with a nonterminal, for  $n=k+1$
  - Assume  $S \rightarrow^+ w$  for  $n=k+1$
  - Given that  $k \geq 1$ ,  $k+1 > 1$ . This means that the derivation of  $w$  is either:
    - $S \rightarrow \dots \rightarrow a_1 \dots a_k U \rightarrow a_1 \dots a_k a_{k+1}$ , where  $U \in N$
    - $S \rightarrow \dots \rightarrow a_1 \dots a_k U \rightarrow a_1 \dots a_k a_{k+1} V$ , where  $U, V \in N$
  - By inductive hypothesis, we have:  
 $((a_1 \dots a_k a_{k+1}) A) \vdash^* ((a_{k+1}) U)$
  - If the last production rule used in the derivation is a simple production rule,  $(U \rightarrow a_{k+1})$ , then by construction of  $M$ ,  $(U a_{k+1} Z) \in \delta$
  - Therefore,  $((a_1 \dots a_k a_{k+1}) A) \vdash^* (() Z) = (() Q)$ .
  - If the last production rule used in the derivation is a compound production rule,  $(U \rightarrow a_{k+1} V)$ , then by construction of  $M$ ,  $(U a_{k+1} V) \in \delta$
  - Therefore,  $((a_1 \dots a_k a_{k+1}) A) \vdash^* (() V) = (() Q)$

# Regular Grammars

## Lemma

$S \xrightarrow{+} w \Leftrightarrow ((a_1 \dots a_n) A) \vdash^+ (() Q)$ , where  $Q=Z$  if  $w$  ends with a terminal symbol and  $Q \in N$  if  $w$  ends with a nonterminal

- ( $\Leftarrow$ ) Assume  $((a_1 \dots a_n) A) \vdash^+ (() Q)$ , where  $Q=Z$  if  $w$  ends with a terminal symbol and  $Q \in N$  if  $w$  ends with a nonterminal
- We must show that  $S \xrightarrow{+} w$
- Induction on,  $n$ , the number of transitions in M's computation

# Regular Grammars

## Lemma

$S \xrightarrow{+} w \Leftrightarrow ((a_1 \dots a_n) A) \vdash^+ (() Q)$ , where  $Q=Z$  if  $w$  ends with a terminal symbol and  $Q \in N$  if  $w$  ends with a nonterminal

- ( $\Leftarrow$ ) Assume  $((a_1 \dots a_n) A) \vdash^+ (() Q)$ , where  $Q=Z$  if  $w$  ends with a terminal symbol and  $Q \in N$  if  $w$  ends with a nonterminal
- We must show that  $S \xrightarrow{+} w$
- Induction on,  $n$ , the number of transitions in M's computation
- Base Case:  $n=1$
- This means that  $w$  ends with a terminal
- M's computation is either:  
 $(() A) \vdash (() Z) \vee ((a) A) \vdash (() Z)$

# Regular Grammars

## Lemma

$S \xrightarrow{+} w \Leftrightarrow ((a_1 \dots a_n) A) \vdash^+ (() Q)$ , where  $Q=Z$  if  $w$  ends with a terminal symbol and  $Q \in N$  if  $w$  ends with a nonterminal

- ( $\Leftarrow$ ) Assume  $((a_1 \dots a_n) A) \vdash^+ (() Q)$ , where  $Q=Z$  if  $w$  ends with a terminal symbol and  $Q \in N$  if  $w$  ends with a nonterminal
- We must show that  $S \xrightarrow{+} w$
- Induction on,  $n$ , the number of transitions in  $M$ 's computation
- Base Case:  $n=1$
- This means that  $w$  ends with a terminal
- $M$ 's computation is either:  
 $(() A) \vdash (() Z) \vee ((a) A) \vdash (() Z)$
- For the first computation, by construction of  $M$ ,  $G$  must have  $(S \rightarrow \epsilon)$

# Regular Grammars

## Lemma

$S \xrightarrow{+} w \Leftrightarrow ((a_1 \dots a_n) A) \vdash^{+} (() Q)$ , where  $Q=Z$  if  $w$  ends with a terminal symbol and  $Q \in N$  if  $w$  ends with a nonterminal

- ( $\Leftarrow$ ) Assume  $((a_1 \dots a_n) A) \vdash^{+} (() Q)$ , where  $Q=Z$  if  $w$  ends with a terminal symbol and  $Q \in N$  if  $w$  ends with a nonterminal
- We must show that  $S \xrightarrow{+} w$
- Induction on,  $n$ , the number of transitions in M's computation
- Base Case:  $n=1$
- This means that  $w$  ends with a terminal
- M's computation is either:  
 $(() A) \vdash (() Z) \vee ((a) A) \vdash (() Z)$
- For the first computation, by construction of M, G must have  $(S \rightarrow \epsilon)$
- For the second computation, by construction of M, G must have  $(S \rightarrow a)$

# Regular Grammars

## Lemma

$S \xrightarrow{+} w \Leftrightarrow ((a_1 \dots a_n) A) \vdash^{+} (() Q)$ , where  $Q=Z$  if  $w$  ends with a terminal symbol and  $Q \in N$  if  $w$  ends with a nonterminal

- ( $\Leftarrow$ ) Assume  $((a_1 \dots a_n) A) \vdash^{+} (() Q)$ , where  $Q=Z$  if  $w$  ends with a terminal symbol and  $Q \in N$  if  $w$  ends with a nonterminal
- We must show that  $S \xrightarrow{+} w$
- Induction on,  $n$ , the number of transitions in  $M$ 's computation
- Base Case:  $n=1$
- This means that  $w$  ends with a terminal
- $M$ 's computation is either:  
 $(() A) \vdash (() Z) \vee ((a) A) \vdash (() Z)$
- For the first computation, by construction of  $M$ ,  $G$  must have  $(S \rightarrow \epsilon)$
- For the second computation, by construction of  $M$ ,  $G$  must have  $(S \rightarrow a)$
- Therefore,  $S \xrightarrow{+} w$ .

# Regular Grammars

## Lemma

$S \xrightarrow{+} w \Leftrightarrow ((a_1 \dots a_n) A) \vdash^{+} (() Q)$ , where  $Q=Z$  if  $w$  ends with a terminal symbol and  $Q \in N$  if  $w$  ends with a nonterminal

- ( $\Leftarrow$ ) Assume  $((a_1 \dots a_n) A) \vdash^{+} (() Q)$ , where  $Q=Z$  if  $w$  ends with a terminal symbol and  $Q \in N$  if  $w$  ends with a nonterminal
- We must show that  $S \xrightarrow{+} w$
- Induction on,  $n$ , the number of transitions in M's computation
- Base Case:  $n=1$
- This means that  $w$  ends with a terminal
- M's computation is either:  
 $(() A) \vdash (() Z) \vee ((a) A) \vdash (() Z)$
- For the first computation, by construction of M, G must have  $(S \rightarrow \epsilon)$
- For the second computation, by construction of M, G must have  $(S \rightarrow a)$
- Therefore,  $S \xrightarrow{+} w$ .
- Inductive Step:
- Assume:  $((a_1 \dots a_n) A) \vdash^{+} (() Q)$ , where  $Q=Z$  if  $w$  ends with a terminal symbol and  $Q \in N$  if  $w$  ends with a nonterminal  $\Rightarrow S \xrightarrow{+} w$ , for  $n=k$ .

# Regular Grammars

## Lemma

$S \xrightarrow{+} w \Leftrightarrow ((a_1 \dots a_n) A) \vdash^+ ((Q) Q)$ , where  $Q=Z$  if  $w$  ends with a terminal symbol and  $Q \in N$  if  $w$  ends with a nonterminal

- ( $\Leftarrow$ ) Assume  $((a_1 \dots a_n) A) \vdash^+ ((Q) Q)$ , where  $Q=Z$  if  $w$  ends with a terminal symbol and  $Q \in N$  if  $w$  ends with a nonterminal
- We must show that  $S \xrightarrow{+} w$
- Induction on,  $n$ , the number of transitions in M's computation
- Base Case:  $n=1$
- This means that  $w$  ends with a terminal
- M's computation is either:  
 $((Q) A) \vdash ((Z) Z) \vee ((a) A) \vdash ((Z) Z)$
- For the first computation, by construction of M, G must have  $(S \rightarrow \epsilon)$
- For the second computation, by construction of M, G must have  $(S \rightarrow a)$
- Therefore,  $S \xrightarrow{+} w$ .
- Inductive Step:
- Assume:  $((a_1 \dots a_n) A) \vdash^+ ((Q) Q)$ , where  $Q=Z$  if  $w$  ends with a terminal symbol and  $Q \in N$  if  $w$  ends with a nonterminal  $\Rightarrow S \xrightarrow{+} w$ , for  $n=k$ .
- Show:  $((a_1 \dots a_{n+1}) A) \vdash^+ ((Q) Q)$ , where  $Q=Z$  if  $w$  ends with a terminal symbol and  $Q \in N$  if  $w$  ends with a nonterminal  $\Rightarrow S \xrightarrow{+} w$ , for  $n=k+1$ .

# Regular Grammars

## Lemma

$S \xrightarrow{+} w \Leftrightarrow ((a_1 \dots a_n) A) \vdash^+ (() Q)$ , where  $Q=Z$  if  $w$  ends with a terminal symbol and  $Q \in N$  if  $w$  ends with a nonterminal

- ( $\Leftarrow$ ) Assume  $((a_1 \dots a_n) A) \vdash^+ (() Q)$ , where  $Q=Z$  if  $w$  ends with a terminal symbol and  $Q \in N$  if  $w$  ends with a nonterminal
- We must show that  $S \xrightarrow{+} w$
- Induction on,  $n$ , the number of transitions in M's computation
- Base Case:  $n=1$
- This means that  $w$  ends with a terminal
- M's computation is either:  
 $(() A) \vdash (() Z) \vee ((a) A) \vdash (() Z)$
- For the first computation, by construction of M, G must have  $(S \rightarrow \epsilon)$
- For the second computation, by construction of M, G must have  $(S \rightarrow a)$
- Therefore,  $S \xrightarrow{+} w$ .
- Inductive Step:
- Assume:  $((a_1 \dots a_n) A) \vdash^+ (() Q)$ , where  $Q=Z$  if  $w$  ends with a terminal symbol and  $Q \in N$  if  $w$  ends with a nonterminal  $\Rightarrow S \xrightarrow{+} w$ , for  $n=k$ .
- Show:  $((a_1 \dots a_{n+1}) A) \vdash^+ (() Q)$ , where  $Q=Z$  if  $w$  ends with a terminal symbol and  $Q \in N$  if  $w$  ends with a nonterminal  $\Rightarrow S \xrightarrow{+} w$ , for  $n=k+1$ .
- Assume  $((a_1 \dots a_{k+1}) A) \vdash^+ (() Q)$ , where  $Q=Z$  if  $w$  ends with a terminal symbol and  $Q \in N$  if  $w$  ends with a nonterminal. Given that  $k \geq 1$ ,  $k+1 > 1$

# Regular Grammars

## Lemma

$S \xrightarrow{+} w \Leftrightarrow ((a_1 \dots a_n) A) \vdash^+ ((Q) Q)$ , where  $Q=Z$  if  $w$  ends with a terminal symbol and  $Q \in N$  if  $w$  ends with a nonterminal

- ( $\Leftarrow$ ) Assume  $((a_1 \dots a_n) A) \vdash^+ ((Q) Q)$ , where  $Q=Z$  if  $w$  ends with a terminal symbol and  $Q \in N$  if  $w$  ends with a nonterminal
- We must show that  $S \xrightarrow{+} w$
- Induction on,  $n$ , the number of transitions in M's computation
- Base Case:  $n=1$
- This means that  $w$  ends with a terminal
- M's computation is either:  
 $((Q) A) \vdash ((Z) Z) \vee ((a) A) \vdash ((Z) Z)$
- For the first computation, by construction of M, G must have  $(S \rightarrow \epsilon)$
- For the second computation, by construction of M, G must have  $(S \rightarrow a)$
- Therefore,  $S \xrightarrow{+} w$ .
- Inductive Step:
- Assume:  $((a_1 \dots a_n) A) \vdash^+ ((Q) Q)$ , where  $Q=Z$  if  $w$  ends with a terminal symbol and  $Q \in N$  if  $w$  ends with a nonterminal  $\Rightarrow S \xrightarrow{+} w$ , for  $n=k$ .
- Show:  $((a_1 \dots a_{n+1}) A) \vdash^+ ((Q) Q)$ , where  $Q=Z$  if  $w$  ends with a terminal symbol and  $Q \in N$  if  $w$  ends with a nonterminal  $\Rightarrow S \xrightarrow{+} w$ , for  $n=k+1$ .
- Assume  $((a_1 \dots a_{k+1}) A) \vdash^+ ((Q) Q)$ , where  $Q=Z$  if  $w$  ends with a terminal symbol and  $Q \in N$  if  $w$  ends with a nonterminal. Given that  $k \geq 1$ ,  $k+1 > 1$
- This means that M's computation on  $(a_1 \dots a_{k+1})$  is:  
 $((a_1 \dots a_{k+1}) A) \vdash^* ((a_{k+1}) R) \vdash ((Q) Q)$

# Regular Grammars

## Lemma

$S \xrightarrow{+} w \Leftrightarrow ((a_1 \dots a_n) A) \vdash^+ (() Q)$ , where  $Q=Z$  if  $w$  ends with a terminal symbol and  $Q \in N$  if  $w$  ends with a nonterminal

- ( $\Leftarrow$ ) Assume  $((a_1 \dots a_n) A) \vdash^+ (() Q)$ , where  $Q=Z$  if  $w$  ends with a terminal symbol and  $Q \in N$  if  $w$  ends with a nonterminal
- We must show that  $S \xrightarrow{+} w$
- Induction on,  $n$ , the number of transitions in M's computation
- Base Case:  $n=1$
- This means that  $w$  ends with a terminal
- M's computation is either:  
 $(() A) \vdash (() Z) \vee ((a) A) \vdash (() Z)$
- For the first computation, by construction of M, G must have  $(S \rightarrow \epsilon)$
- For the second computation, by construction of M, G must have  $(S \rightarrow a)$
- Therefore,  $S \xrightarrow{+} w$ .
- Inductive Step:
- Assume:  $((a_1 \dots a_n) A) \vdash^+ (() Q)$ , where  $Q=Z$  if  $w$  ends with a terminal symbol and  $Q \in N$  if  $w$  ends with a nonterminal  $\Rightarrow S \xrightarrow{+} w$ , for  $n=k$ .
- Show:  $((a_1 \dots a_{n+1}) A) \vdash^+ (() Q)$ , where  $Q=Z$  if  $w$  ends with a terminal symbol and  $Q \in N$  if  $w$  ends with a nonterminal  $\Rightarrow S \xrightarrow{+} w$ , for  $n=k+1$ .
- Assume  $((a_1 \dots a_{k+1}) A) \vdash^+ (() Q)$ , where  $Q=Z$  if  $w$  ends with a terminal symbol and  $Q \in N$  if  $w$  ends with a nonterminal. Given that  $k \geq 1$ ,  $k+1 > 1$
- This means that M's computation on  $(a_1 \dots a_{k+1})$  is:  
 $((a_1 \dots a_{k+1}) A) \vdash^* ((a_{k+1}) R) \vdash (() Q)$
- By inductive hypothesis, we have:  
 $A \xrightarrow{*} a_1 \dots a_k R$

# Regular Grammars

## Lemma

$S \xrightarrow{+} w \Leftrightarrow ((a_1 \dots a_n) A) \vdash^+ (() Q)$ , where  $Q=Z$  if  $w$  ends with a terminal symbol and  $Q \in N$  if  $w$  ends with a nonterminal

- ( $\Leftarrow$ ) Assume  $((a_1 \dots a_n) A) \vdash^+ (() Q)$ , where  $Q=Z$  if  $w$  ends with a terminal symbol and  $Q \in N$  if  $w$  ends with a nonterminal
- We must show that  $S \xrightarrow{+} w$
- Induction on,  $n$ , the number of transitions in M's computation
- Base Case:  $n=1$
- This means that  $w$  ends with a terminal
- M's computation is either:  
 $(() A) \vdash (() Z) \vee ((a) A) \vdash (() Z)$
- For the first computation, by construction of M, G must have  $(S \rightarrow \epsilon)$
- For the second computation, by construction of M, G must have  $(S \rightarrow a)$
- Therefore,  $S \xrightarrow{+} w$ .
- Inductive Step:
- Assume:  $((a_1 \dots a_n) A) \vdash^+ (() Q)$ , where  $Q=Z$  if  $w$  ends with a terminal symbol and  $Q \in N$  if  $w$  ends with a nonterminal  $\Rightarrow S \xrightarrow{+} w$ , for  $n=k$ .
- Show:  $((a_1 \dots a_{n+1}) A) \vdash^+ (() Q)$ , where  $Q=Z$  if  $w$  ends with a terminal symbol and  $Q \in N$  if  $w$  ends with a nonterminal  $\Rightarrow S \xrightarrow{+} w$ , for  $n=k+1$ .
- Assume  $((a_1 \dots a_{k+1}) A) \vdash^+ (() Q)$ , where  $Q=Z$  if  $w$  ends with a terminal symbol and  $Q \in N$  if  $w$  ends with a nonterminal. Given that  $k \geq 1$ ,  $k+1 > 1$
- This means that M's computation on  $(a_1 \dots a_{k+1})$  is:  
 $((a_1 \dots a_{k+1}) A) \vdash^* ((a_{k+1}) R) \vdash (() Q)$
- By inductive hypothesis, we have:  
 $A \xrightarrow{*} a_1 \dots a_k R$
- The last transition in M's computation has either  $Q=Z$  or  $Q \neq Z$

# Regular Grammars

## Lemma

$S \xrightarrow{+} w \Leftrightarrow ((a_1 \dots a_n) A) \vdash^+ ((Q) Q)$ , where  $Q=Z$  if  $w$  ends with a terminal symbol and  $Q \in N$  if  $w$  ends with a nonterminal

- ( $\Leftarrow$ ) Assume  $((a_1 \dots a_n) A) \vdash^+ ((Q) Q)$ , where  $Q=Z$  if  $w$  ends with a terminal symbol and  $Q \in N$  if  $w$  ends with a nonterminal
- We must show that  $S \xrightarrow{+} w$
- Induction on,  $n$ , the number of transitions in M's computation
- Base Case:  $n=1$
- This means that  $w$  ends with a terminal
- M's computation is either:  
 $((Q) A) \vdash ((Z) Z) \vee ((a) A) \vdash ((Z) Z)$
- For the first computation, by construction of M, G must have  $(S \rightarrow \epsilon)$
- For the second computation, by construction of M, G must have  $(S \rightarrow a)$
- Therefore,  $S \xrightarrow{+} w$ .
- Inductive Step:
- Assume:  $((a_1 \dots a_n) A) \vdash^+ ((Q) Q)$ , where  $Q=Z$  if  $w$  ends with a terminal symbol and  $Q \in N$  if  $w$  ends with a nonterminal  $\Rightarrow S \xrightarrow{+} w$ , for  $n=k$ .
- Show:  $((a_1 \dots a_{n+1}) A) \vdash^+ ((Q) Q)$ , where  $Q=Z$  if  $w$  ends with a terminal symbol and  $Q \in N$  if  $w$  ends with a nonterminal  $\Rightarrow S \xrightarrow{+} w$ , for  $n=k+1$ .
- Assume  $((a_1 \dots a_{k+1}) A) \vdash^+ ((Q) Q)$ , where  $Q=Z$  if  $w$  ends with a terminal symbol and  $Q \in N$  if  $w$  ends with a nonterminal. Given that  $k \geq 1$ ,  $k+1 > 1$
- This means that M's computation on  $(a_1 \dots a_{k+1})$  is:  
 $((a_1 \dots a_{k+1}) A) \vdash^* ((a_{k+1}) R) \vdash ((Q) Q)$
- By inductive hypothesis, we have:  
 $A \xrightarrow{*} a_1 \dots a_k R$
- The last transition in M's computation has either  $Q=Z$  or  $Q \neq Z$
- If  $Q=Z$  then, by construction of G,  $(R \rightarrow a_{k+1}) \in P$ . Therefore, we have:  
 $A \xrightarrow{*} a_1 \dots a_{k+1}$

# Regular Grammars

## Lemma

$S \xrightarrow{+} w \Leftrightarrow ((a_1 \dots a_n) A) \vdash^+ (() Q)$ , where  $Q=Z$  if  $w$  ends with a terminal symbol and  $Q \in N$  if  $w$  ends with a nonterminal

- ( $\Leftarrow$ ) Assume  $((a_1 \dots a_n) A) \vdash^+ (() Q)$ , where  $Q=Z$  if  $w$  ends with a terminal symbol and  $Q \in N$  if  $w$  ends with a nonterminal
- We must show that  $S \xrightarrow{+} w$
- Induction on,  $n$ , the number of transitions in M's computation
- Base Case:  $n=1$
- This means that  $w$  ends with a terminal
- M's computation is either:  
 $(() A) \vdash (() Z) \vee ((a) A) \vdash (() Z)$
- For the first computation, by construction of M, G must have  $(S \rightarrow \epsilon)$
- For the second computation, by construction of M, G must have  $(S \rightarrow a)$
- Therefore,  $S \xrightarrow{+} w$ .
- Inductive Step:
- Assume:  $((a_1 \dots a_n) A) \vdash^+ (() Q)$ , where  $Q=Z$  if  $w$  ends with a terminal symbol and  $Q \in N$  if  $w$  ends with a nonterminal  $\Rightarrow S \xrightarrow{+} w$ , for  $n=k$ .
- Show:  $((a_1 \dots a_{n+1}) A) \vdash^+ (() Q)$ , where  $Q=Z$  if  $w$  ends with a terminal symbol and  $Q \in N$  if  $w$  ends with a nonterminal  $\Rightarrow S \xrightarrow{+} w$ , for  $n=k+1$ .
- Assume  $((a_1 \dots a_{k+1}) A) \vdash^+ (() Q)$ , where  $Q=Z$  if  $w$  ends with a terminal symbol and  $Q \in N$  if  $w$  ends with a nonterminal. Given that  $k \geq 1$ ,  $k+1 > 1$
- This means that M's computation on  $(a_1 \dots a_{k+1})$  is:  
 $((a_1 \dots a_{k+1}) A) \vdash^* ((a_{k+1}) R) \vdash (() Q)$
- By inductive hypothesis, we have:  
 $A \rightarrow^* a_1 \dots a_k R$
- The last transition in M's computation has either  $Q=Z$  or  $Q \neq Z$
- If  $Q=Z$  then, by construction of G,  $(R \rightarrow a_{k+1}) \in P$ . Therefore, we have:  
 $A \rightarrow^* a_1 \dots a_{k+1}$
- If  $Q \neq Z$  then, by construction, of M  $(R \rightarrow a_{k+1} Q)$ .

# Regular Grammars

## Lemma

$S \xrightarrow{+} w \Leftrightarrow ((a_1 \dots a_n) A) \vdash^+ ((Q) Q)$ , where  $Q=Z$  if  $w$  ends with a terminal symbol and  $Q \in N$  if  $w$  ends with a nonterminal

- ( $\Leftarrow$ ) Assume  $((a_1 \dots a_n) A) \vdash^+ ((Q) Q)$ , where  $Q=Z$  if  $w$  ends with a terminal symbol and  $Q \in N$  if  $w$  ends with a nonterminal
- We must show that  $S \xrightarrow{+} w$
- Induction on,  $n$ , the number of transitions in M's computation
- Base Case:  $n=1$
- This means that  $w$  ends with a terminal
- M's computation is either:  
 $((Q) A) \vdash ((Z) Z) \vee ((a) A) \vdash ((Z) Z)$
- For the first computation, by construction of M, G must have  $(S \rightarrow \epsilon)$
- For the second computation, by construction of M, G must have  $(S \rightarrow a)$
- Therefore,  $S \xrightarrow{+} w$ .
- Inductive Step:
- Assume:  $((a_1 \dots a_n) A) \vdash^+ ((Q) Q)$ , where  $Q=Z$  if  $w$  ends with a terminal symbol and  $Q \in N$  if  $w$  ends with a nonterminal  $\Rightarrow S \xrightarrow{+} w$ , for  $n=k$ .
- Show:  $((a_1 \dots a_{n+1}) A) \vdash^+ ((Q) Q)$ , where  $Q=Z$  if  $w$  ends with a terminal symbol and  $Q \in N$  if  $w$  ends with a nonterminal  $\Rightarrow S \xrightarrow{+} w$ , for  $n=k+1$ .
- Assume  $((a_1 \dots a_{k+1}) A) \vdash^+ ((Q) Q)$ , where  $Q=Z$  if  $w$  ends with a terminal symbol and  $Q \in N$  if  $w$  ends with a nonterminal. Given that  $k \geq 1$ ,  $k+1 > 1$
- This means that M's computation on  $(a_1 \dots a_{k+1})$  is:  
 $((a_1 \dots a_{k+1}) A) \vdash^* ((a_{k+1}) R) \vdash ((Q) Q)$
- By inductive hypothesis, we have:  
 $A \rightarrow^* a_1 \dots a_k R$
- The last transition in M's computation has either  $Q=Z$  or  $Q \neq Z$
- If  $Q=Z$  then, by construction of G,  $(R \rightarrow a_{k+1}) \in P$ . Therefore, we have:  
 $A \rightarrow^* a_1 \dots a_{k+1}$
- If  $Q \neq Z$  then, by construction, of M  $(R \rightarrow a_{k+1} Q)$ .
- Therefore, we have:  $A \rightarrow^* a_1 \dots a_{k+1} Q$

# Regular Grammars

## Theorem

$L$  is generated by a regular grammar  $\Rightarrow L$  is regular.

- A assume  $L$  is generated by a regular grammar

# Regular Grammars

## Theorem

$L$  is generated by a regular grammar  $\Rightarrow L$  is regular.

- Assume  $L$  is generated by a regular grammar
- Let  $G$  be a regular grammar such that  $L = L(G)$ , let  $w \in L$ , and let  $M = (\text{rg2ndfa } G)$

# Regular Grammars

## Theorem

$L$  is generated by a regular grammar  $\Rightarrow L$  is regular.

- Assume  $L$  is generated by a regular grammar
- Let  $G$  be a regular grammar such that  $L = L(G)$ , let  $w \in L$ , and let  $M = (\text{rg2ndfa } G)$
- Given that  $G$  can be transformed to  $M$ ,  $S \xrightarrow{*} w \Leftrightarrow (w \ A) \vdash^{+} ((\ ) \ Z)$

# Regular Grammars

## Theorem

$L$  is generated by a regular grammar  $\Rightarrow L$  is regular.

- Assume  $L$  is generated by a regular grammar
- Let  $G$  be a regular grammar such that  $L = L(G)$ , let  $w \in L$ , and let  $M = (\text{rg2ndfa } G)$
- Given that  $G$  can be transformed to  $M$ ,  $S \xrightarrow{*} w \Leftrightarrow (w \ A) \vdash^{+} ((\ ) \ Z)$
- Given that  $w$  is an arbitrary word,  $M$  decides  $L$
- Thus,  $L$  is regular

# Regular Grammars

- HOMEWORK: 9, 10, 11, 12
- QUIZ: 7 (due in a week)

# Pumping Theorem for RLs

- Variety of techniques to establish that a language,  $L$ , is regular

# Pumping Theorem for RLs

- Variety of techniques to establish that a language,  $L$ , is regular
- You might already suspect, however, that not all languages in the universe are regular
- This belief is likely rooted in the fact that the amount of memory is bounded
- Loss of knowledge
- It would be foolish, however, to dismiss these models as irrelevant to modern Computer Science

# Pumping Theorem for RLs

- Variety of techniques to establish that a language,  $L$ , is regular
- You might already suspect, however, that not all languages in the universe are regular
- This belief is likely rooted in the fact that the amount of memory is bounded
- Loss of knowledge
- It would be foolish, however, to dismiss these models as irrelevant to modern Computer Science
- Computers have a finite amount of memory
- The finite-state machines that we use on a daily basis are, indeed, quite powerful
- Does it surprise you or do you doubt that modern computers are finite-state machines?

# Pumping Theorem for RLs

- Variety of techniques to establish that a language,  $L$ , is regular
- You might already suspect, however, that not all languages in the universe are regular
- This belief is likely rooted in the fact that the amount of memory is bounded
- Loss of knowledge
- It would be foolish, however, to dismiss these models as irrelevant to modern Computer Science
- Computers have a finite amount of memory
- The finite-state machines that we use on a daily basis are, indeed, quite powerful
- Does it surprise you or do you doubt that modern computers are finite-state machines?
- Does a finite amount of memory limit what can be computed?

# Pumping Theorem for RLs

- Consider the following language:  
$$L = \{a^n b^n \mid n \geq 0\}$$
- On the surface it appears to be a rather simple and uninteresting language

# Pumping Theorem for RLs

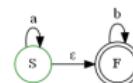
- Consider the following language:  
$$L = \{a^n b^n \mid n \geq 0\}$$
- On the surface it appears to be a rather simple and uninteresting language
- The problem with  $L$  is that  $n$  is a natural number of arbitrary size
- How can a finite-state machine read  $n$  as and then read  $n$  bs?

# Pumping Theorem for RLs

- Consider the following language:

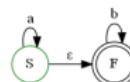
$$L = \{a^n b^n \mid n \geq 0\}$$

- On the surface it appears to be a rather simple and uninteresting language
- The problem with  $L$  is that  $n$  is a natural number of arbitrary size
- How can a finite-state machine read  $n$  as and then read  $n$  bs?
- You may argue this is easy by implementing a finite-state machine that has a loop to read  $n$  as and then a loop to read  $n$  bs:



# Pumping Theorem for RLs

- Consider the following language:  
 $L = \{a^n b^n \mid n \geq 0\}$
- On the surface it appears to be a rather simple and uninteresting language
- The problem with  $L$  is that  $n$  is a natural number of arbitrary size
- How can a finite-state machine read  $n$  as and then read  $n$  bs?
- You may argue this is easy by implementing a finite-state machine that has a loop to read  $n$  as and then a loop to read  $n$  bs:



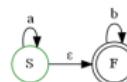
- If we define this machine as  $a2n-b2n$  the following tests pass:

```
;; Tests for a2n-b2n
(check-equal? (sm-apply a2n-b2n '(b b a a)) 'reject)
(check-equal? (sm-apply a2n-b2n '()) 'accept)
(check-equal? (sm-apply a2n-b2n '(a b)) 'accept)
(check-equal? (sm-apply a2n-b2n '(a a b b)) 'accept)
(check-equal? (sm-apply a2n-b2n '(a a a b b)) 'accept)
```

- Should this give us confidence that the machine decides  $L$ ?

# Pumping Theorem for RLs

- Consider the following language:  
 $L = \{a^n b^n \mid n \geq 0\}$
- On the surface it appears to be a rather simple and uninteresting language
- The problem with  $L$  is that  $n$  is a natural number of arbitrary size
- How can a finite-state machine read  $n$  as and then read  $n$  bs?
- You may argue this is easy by implementing a finite-state machine that has a loop to read  $n$  as and then a loop to read  $n$  bs:



- If we define this machine as  $a2n-b2n$  the following tests pass:

```
; ; Tests for a2n-b2n
(check-equal? (sm-apply a2n-b2n '(b b a a)) 'reject)
(check-equal? (sm-apply a2n-b2n '()) 'accept)
(check-equal? (sm-apply a2n-b2n '(a b)) 'accept)
(check-equal? (sm-apply a2n-b2n '(a a b b)) 'accept)
(check-equal? (sm-apply a2n-b2n '(a a a b b)) 'accept)
```

- Should this give us confidence that the machine decides  $L$ ?
- Unfortunately, the answer is an unequivocal no
- The tests are not thorough enough:

```
(check-equal? (sm-apply a2n-b2n '(a a)) 'reject)
(check-equal? (sm-apply a2n-b2n '(b)) 'reject)
(check-equal? (sm-apply a2n-b2n '(a a a b)) 'reject)
```

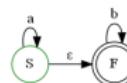
- These tests fail and this should not happen

# Pumping Theorem for RLs

- Consider the following language:

$$L = \{a^n b^n \mid n \geq 0\}$$

- On the surface it appears to be a rather simple and uninteresting language
- The problem with L is that n is a natural number of arbitrary size
- How can a finite-state machine read n as and then read n bs?
- You may argue this is easy by implementing a finite-state machine that has a loop to read n as and then a loop to read n bs:



- If we define this machine as a2n-b2n the following tests pass:

```
;; Tests for a2n-b2n
(check-equal? (sm-apply a2n-b2n '(b b a a)) 'reject)
(check-equal? (sm-apply a2n-b2n '()) 'accept)
(check-equal? (sm-apply a2n-b2n '(a b)) 'accept)
(check-equal? (sm-apply a2n-b2n '(a a b b)) 'accept)
(check-equal? (sm-apply a2n-b2n '(a a a b b)) 'accept)
```

- Should this give us confidence that the machine decides L?
- Unfortunately, the answer is an unequivocal no
- The tests are not thorough enough:

```
(check-equal? (sm-apply a2n-b2n '(a a)) 'reject)
(check-equal? (sm-apply a2n-b2n '(b)) 'reject)
(check-equal? (sm-apply a2n-b2n '(a a a b)) 'reject)
```

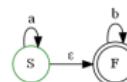
- These tests fail and this should not happen
- The machine needs to remember the number of as
- To remember an arbitrary number of as the machine needs an arbitrary number of states

# Pumping Theorem for RLs

- Consider the following language:

$$L = \{a^n b^n \mid n \geq 0\}$$

- On the surface it appears to be a rather simple and uninteresting language
- The problem with L is that n is a natural number of arbitrary size
- How can a finite-state machine read n as and then read n bs?
- You may argue this is easy by implementing a finite-state machine that has a loop to read n as and then a loop to read n bs:



- If we define this machine as a2n-b2n the following tests pass:

```
; ; Tests for a2n-b2n
(check-equal? (sm-apply a2n-b2n '(b b a a)) 'reject)
(check-equal? (sm-apply a2n-b2n '()) 'accept)
(check-equal? (sm-apply a2n-b2n '(a b)) 'accept)
(check-equal? (sm-apply a2n-b2n '(a a b b)) 'accept)
(check-equal? (sm-apply a2n-b2n '(a a a b b)) 'accept)
```

- Should this give us confidence that the machine decides L?
- Unfortunately, the answer is an unequivocal no
- The tests are not thorough enough:

```
(check-equal? (sm-apply a2n-b2n '(a a)) 'reject)
(check-equal? (sm-apply a2n-b2n '(b)) 'reject)
(check-equal? (sm-apply a2n-b2n '(a a a b)) 'reject)
```

- These tests fail and this should not happen
- The machine needs to remember the number of as
- To remember an arbitrary number of as the machine needs an arbitrary number of states
- This strongly suggests that L is interesting because it is not a regular language
- In general, how can we tell if a language is not regular?

# Pumping Theorem for RLs

- Cycles in the transition diagram of a finite-state machine and Kleene stars in a regular expression suggest that there is a repetitive structure

# Pumping Theorem for RLs

- Cycles in the transition diagram of a finite-state machine and Kleene stars in a regular expression suggest that there is a repetitive structure
- For long enough words repetition must occur 1 or more times
- What does this observation suggest?

# Pumping Theorem for RLs

- Cycles in the transition diagram of a finite-state machine and Kleene stars in a regular expression suggest that there is a repetitive structure
- For long enough words repetition must occur 1 or more times
- What does this observation suggest?
- For long enough words in  $L$  there must be a repeated subword
- Call the repeated subword  $y$
- $w = xyz \in L$

# Pumping Theorem for RLs

- Cycles in the transition diagram of a finite-state machine and Kleene stars in a regular expression suggest that there is a repetitive structure
- For long enough words repetition must occur 1 or more times
- What does this observation suggest?
- For long enough words in  $L$  there must be a repeated subword
- Call the repeated subword  $y$
- $w = xyz \in L$
- $xyyz \in L$ ,  $xyyyz \in L$ ,  $xyyyyyz \in L$  and so on are also in the machine's language.  
The loop is traversed one or more times

# Pumping Theorem for RLs

- Cycles in the transition diagram of a finite-state machine and Kleene stars in a regular expression suggest that there is a repetitive structure
- For long enough words repetition must occur 1 or more times
- What does this observation suggest?
- For long enough words in  $L$  there must be a repeated subword
- Call the repeated subword  $y$
- $w = xyz \in L$
- $xyyz \in L$ ,  $xyyz \in L$ ,  $xyyz \in L$  and so on are also in the machine's language. The loop is traversed one or more times
- It is also the case that  $xz$  is in the machine's language

# Pumping Theorem for RLs

Regular  
Expressions

Deterministic  
Finite  
Automata

Nondeterministic  
Finite  
Automata

Finite-State  
Automata and  
Regular  
Expressions

Regular  
Grammars

Pumping  
Theorem for  
Regular  
Languages

- Cycles in the transition diagram of a finite-state machine and Kleene stars in a regular expression suggest that there is a repetitive structure
- For long enough words repetition must occur 1 or more times
- What does this observation suggest?
- For long enough words in  $L$  there must be a repeated subword
- Call the repeated subword  $y$
- $w = xyz \in L$
- $xyyz \in L$ ,  $xyyyz \in L$ ,  $xyyyyyz \in L$  and so on are also in the machine's language. The loop is traversed one or more times
- It is also the case that  $xz$  is in the machine's language
- Generalize:  $xy^iz \in L$ , where  $i \geq 0$
- If  $w \in L$  then we can "pump" up or down on  $y$  and still have a word that is in  $L$

# Pumping Theorem for RLs

## Theorem

*For a regular language,  $L$ , there is a word length  $n \geq 1$  such that any  $w \in L$  may be written as  $w = xyz$ , where  $y \neq \epsilon$ ,  $|xy| \leq n$ , and  $xy^i z \in L$  for  $i \geq 0$ .*

# Pumping Theorem for RLs

## Theorem

*For a regular language,  $L$ , there is a word length  $n \geq 1$  such that any  $w \in L$  may be written as  $w = xyz$ , where  $y \neq \epsilon$ ,  $|xy| \leq n$ , and  $xy^i z \in L$  for  $i \geq 0$ .*

- Let us be sure we understand what the theorem is stating

# Pumping Theorem for RLs

## Theorem

*For a regular language,  $L$ , there is a word length  $n \geq 1$  such that any  $w \in L$  may be written as  $w = xyz$ , where  $y \neq \epsilon$ ,  $|xy| \leq n$ , and  $xy^i z \in L$  for  $i \geq 0$ .*

- Let us be sure we understand what the theorem is stating
- $w \in L$  of length greater than or equal to some positive integer,  $n$ , may be divided into three parts,  $x$ ,  $y$  and  $z$

# Pumping Theorem for RLs

## Theorem

*For a regular language,  $L$ , there is a word length  $n \geq 1$  such that any  $w \in L$  may be written as  $w = xyz$ , where  $y \neq \epsilon$ ,  $|xy| \leq n$ , and  $xy^i z \in L$  for  $i \geq 0$ .*

- Let us be sure we understand what the theorem is stating
- $w \in L$  of length greater than or equal to some positive integer,  $n$ , may be divided into three parts,  $x$ ,  $y$  and  $z$
- $y$  is nonempty

# Pumping Theorem for RLs

## Theorem

*For a regular language,  $L$ , there is a word length  $n \geq 1$  such that any  $w \in L$  may be written as  $w = xyz$ , where  $y \neq \epsilon$ ,  $|xy| \leq n$ , and  $xy^i z \in L$  for  $i \geq 0$ .*

- Let us be sure we understand what the theorem is stating
- $w \in L$  of length greater than or equal to some positive integer,  $n$ , may be divided into three parts,  $x$ ,  $y$  and  $z$
- $y$  is nonempty
- The length of  $xy$  cannot be longer than  $n$
- That is,  $xy$  must be at the beginning of  $w$

# Pumping Theorem for RLs

## Theorem

*For a regular language,  $L$ , there is a word length  $n \geq 1$  such that any  $w \in L$  may be written as  $w = xyz$ , where  $y \neq \epsilon$ ,  $|xy| \leq n$ , and  $xy^i z \in L$  for  $i \geq 0$ .*

- Let us be sure we understand what the theorem is stating
- $w \in L$  of length greater than or equal to some positive integer,  $n$ , may be divided into three parts,  $x$ ,  $y$  and  $z$
- $y$  is nonempty
- The length of  $xy$  cannot be longer than  $n$
- That is,  $xy$  must be at the beginning of  $w$
- What is this theorem good for?

# Pumping Theorem for RLs

## Theorem

*For a regular language,  $L$ , there is a word length  $n \geq 1$  such that any  $w \in L$  may be written as  $w = xyz$ , where  $y \neq \epsilon$ ,  $|xy| \leq n$ , and  $xy^i z \in L$  for  $i \geq 0$ .*

- Let us be sure we understand what the theorem is stating
- $w \in L$  of length greater than or equal to some positive integer,  $n$ , may be divided into three parts,  $x$ ,  $y$  and  $z$
- $y$  is nonempty
- The length of  $xy$  cannot be longer than  $n$
- That is,  $xy$  must be at the beginning of  $w$
- What is this theorem good for?
- For a concrete  $w \in L$  that is long enough we must be able to identify a nonempty  $y$  that may safely be repeated an arbitrary number of times and still end with a word in  $L$
- If such a  $y$  does not exist then the language is not regular

# Pumping Theorem for RLs

## Theorem

For a regular language,  $L$ , there is a word length  $n \geq 1$  such that any  $w \in L$  may be written as  $w = xyz$ , where  $y \neq \epsilon$ ,  $|xy| \leq n$ , and  $xy^i z \in L$  for  $i \geq 0$ .

- Given that  $L$  is regular,  $M = (\text{make-dfa } K \Sigma S F \delta)$  decides  $L$

# Pumping Theorem for RLs

## Theorem

For a regular language,  $L$ , there is a word length  $n \geq 1$  such that any  $w \in L$  may be written as  $w = xyz$ , where  $y \neq \epsilon$ ,  $|xy| \leq n$ , and  $xy^i z \in L$  for  $i \geq 0$ .

- Given that  $L$  is regular,  $M = (\text{make-dfa } K \ \Sigma \ S \ F \ \delta)$  decides  $L$
- $n = |K|$
- $w = (a_1 \ a_2 \ \dots \ a_n \ \dots \ a_m) \in L$

# Pumping Theorem for RLs

## Theorem

For a regular language,  $L$ , there is a word length  $n \geq 1$  such that any  $w \in L$  may be written as  $w = xyz$ , where  $y \neq \epsilon$ ,  $|xy| \leq n$ , and  $xy^i z \in L$  for  $i \geq 0$ .

- Given that  $L$  is regular,  $M = (\text{make-dfa } K \ \Sigma \ S \ F \ \delta)$  decides  $L$
- $n = |K|$
- $w = (a_1 \ a_2 \ \dots \ a_n \ \dots \ a_m) \in L$
- The first  $n$  steps of  $M$ 's computation on  $w$  are as follows:  
 $((a_1 \ a_2 \ \dots \ a_n) \ S) \vdash ((a_2 \dots \ a_n) \ Q_1) \vdash ((a_3 \dots \ a_n) \ Q_2) \vdash \dots \ ((\ ) \ Q_n)$
- Observe that the computation has  $n+1$  configurations

# Pumping Theorem for RLs

## Theorem

For a regular language,  $L$ , there is a word length  $n \geq 1$  such that any  $w \in L$  may be written as  $w = xyz$ , where  $y \neq \epsilon$ ,  $|xy| \leq n$ , and  $xy^i z \in L$  for  $i \geq 0$ .

- Given that  $L$  is regular,  $M = (\text{make-dfa } K \ \Sigma \ S \ F \ \delta)$  decides  $L$
- $n = |K|$
- $w = (a_1 \ a_2 \ \dots \ a_n \ \dots \ a_m) \in L$
- The first  $n$  steps of  $M$ 's computation on  $w$  are as follows:  
 $((a_1 \ a_2 \ \dots \ a_n) \ S) \vdash ((a_2 \dots \ a_n) \ Q_1) \vdash ((a_3 \dots \ a_n) \ Q_2) \vdash \dots \ ((\ ) \ Q_n)$
- Observe that the computation has  $n+1$  configurations
- Since  $M$  only has  $n$  states by the pigeonhole principle there must be a repeated state in the computation:  $Q_i = Q_j$
- This means there is a loop in  $M$ :  
 $((a_i \dots a_j) \ Q_{i-1}) \vdash ((a_{i+1} \dots a_j) \ Q_i) \vdash^* ((\ ) \ Q_i)$

# Pumping Theorem for RLs

## Theorem

For a regular language,  $L$ , there is a word length  $n \geq 1$  such that any  $w \in L$  may be written as  $w = xyz$ , where  $y \neq \epsilon$ ,  $|xy| \leq n$ , and  $xy^i z \in L$  for  $i \geq 0$ .

- Given that  $L$  is regular,  $M = (\text{make-dfa } K \ \Sigma \ S \ F \ \delta)$  decides  $L$
- $n = |K|$
- $w = (a_1 \ a_2 \ \dots \ a_n \ \dots \ a_m) \in L$
- The first  $n$  steps of  $M$ 's computation on  $w$  are as follows:  
 $((a_1 \ a_2 \ \dots \ a_n) \ S) \vdash ((a_2 \dots \ a_n) \ Q_1) \vdash ((a_3 \dots \ a_n) \ Q_2) \vdash \dots \ ((\ ) \ Q_n)$
- Observe that the computation has  $n+1$  configurations
- Since  $M$  only has  $n$  states by the pigeonhole principle there must be a repeated state in the computation:  $Q_i = Q_j$
- This means there is a loop in  $M$ :  
 $((a_1 \dots a_j) \ Q_{i-1}) \vdash ((a_{i+1} \dots a_j) \ Q_i) \vdash^* ((\ ) \ Q_i)$
- Given that  $i < j$  and  $M$  is a dfa,  $(a_i \dots a_j)$  is not empty
- $(a_i \dots a_j)$  may be removed from  $w$  or repeated an arbitrary number of times and the resulting word is still in  $L$

# Pumping Theorem for RLs

## Theorem

For a regular language,  $L$ , there is a word length  $n \geq 1$  such that any  $w \in L$  may be written as  $w = xyz$ , where  $y \neq \epsilon$ ,  $|xy| \leq n$ , and  $xy^i z \in L$  for  $i \geq 0$ .

- Given that  $L$  is regular,  $M = (\text{make-dfa } K \ \Sigma \ S \ F \ \delta)$  decides  $L$
- $n = |K|$
- $w = (a_1 \ a_2 \ \dots \ a_n \ \dots \ a_m) \in L$
- The first  $n$  steps of  $M$ 's computation on  $w$  are as follows:  
 $((a_1 \ a_2 \ \dots \ a_n) \ S) \vdash ((a_2 \ \dots \ a_n) \ Q_1) \vdash ((a_3 \ \dots \ a_n) \ Q_2) \vdash \dots \ ((\ ) \ Q_n)$
- Observe that the computation has  $n+1$  configurations
- Since  $M$  only has  $n$  states by the pigeonhole principle there must be a repeated state in the computation:  $Q_i = Q_j$
- This means there is a loop in  $M$ :  
 $((a_1 \ \dots \ a_j) \ Q_{i-1}) \vdash ((a_{i+1} \ \dots \ a_j) \ Q_i) \vdash^* ((\ ) \ Q_i)$
- Given that  $i < j$  and  $M$  is a dfa,  $(a_i \ \dots \ a_j)$  is not empty
- $(a_i \ \dots \ a_j)$  may be removed from  $w$  or repeated an arbitrary number of times and the resulting word is still in  $L$
- If we define  $x = (a_1 \ \dots \ a_{i-1})$ ,  $y = (a_i \ \dots \ a_j)$ , and  $z = (a_{j+1} \ \dots \ a_n)$  then  $xy^i z \in L$

# Pumping Theorem for RLs

## Theorem

For a regular language,  $L$ , there is a word length  $n \geq 1$  such that any  $w \in L$  may be written as  $w = xyz$ , where  $y \neq \epsilon$ ,  $|xy| \leq n$ , and  $xy^iz \in L$  for  $i \geq 0$ .

- Given that  $L$  is regular,  $M = (\text{make-dfa } K \ \Sigma \ S \ F \ \delta)$  decides  $L$
- $n = |K|$
- $w = (a_1 \ a_2 \ \dots \ a_n \ \dots \ a_m) \in L$
- The first  $n$  steps of  $M$ 's computation on  $w$  are as follows:  
 $((a_1 \ a_2 \ \dots \ a_n) \ S) \vdash ((a_2 \ \dots \ a_n) \ Q_1) \vdash ((a_3 \ \dots \ a_n) \ Q_2) \vdash \dots \ ((\ ) \ Q_n)$
- Observe that the computation has  $n+1$  configurations
- Since  $M$  only has  $n$  states by the pigeonhole principle there must be a repeated state in the computation:  $Q_i = Q_j$
- This means there is a loop in  $M$ :  
 $((a_1 \ \dots \ a_j) \ Q_{i-1}) \vdash ((a_{i+1} \ \dots \ a_j) \ Q_i) \vdash^* ((\ ) \ Q_i)$
- Given that  $i < j$  and  $M$  is a dfa,  $(a_i \ \dots \ a_j)$  is not empty
- $(a_i \ \dots \ a_j)$  may be removed from  $w$  or repeated an arbitrary number of times and the resulting word is still in  $L$
- If we define  $x = (a_1 \ \dots \ a_{i-1})$ ,  $y = (a_i \ \dots \ a_j)$ , and  $z = (a_{j+1} \ \dots \ a_n)$  then  $xy^iz \in L$
- Finally, observe that  $|(a_1 \ \dots \ a_j)| \leq n$  because the loop can contain at most all of  $M$ 's states
- Therefore,  $|xy| \leq n$

# Pumping Theorem for RLs

## Theorem

$L = \{a^n b^n \mid n \geq 0\}$  is not regular

- Assume  $L$  is regular

# Pumping Theorem for RLs

## Theorem

$L = \{a^n b^n \mid n \geq 0\}$  is not regular

- Assume  $L$  is regular
- Let  $M = (\text{make-dfa } K \Sigma S F \delta)$  decide  $L$
- Let  $n = |K|$

# Pumping Theorem for RLs

## Theorem

$L = \{a^n b^n \mid n \geq 0\}$  is not regular

- Assume  $L$  is regular
- Let  $M = (\text{make-dfa } K \ \Sigma \ S \ F \ \delta)$  decide  $L$
- Let  $n = |K|$
- The pumping theorem requires picking a  $w \in L$  such that  $|w| \geq n$
- Let  $w = a^n b^n$

# Pumping Theorem for RLs

## Theorem

$L = \{a^n b^n \mid n \geq 0\}$  is not regular

- Assume  $L$  is regular
- Let  $M = (\text{make-dfa } K \ \Sigma \ S \ F \ \delta)$  decide  $L$
- Let  $n = |K|$
- The pumping theorem requires picking a  $w \in L$  such that  $|w| \geq n$
- Let  $w = a^n b^n$
- $M$ 's computation on  $w$  must visit at least one state twice
- That is,  $w$  is long enough

# Pumping Theorem for RLs

## Theorem

$L = \{a^n b^n \mid n \geq 0\}$  is not regular

- Assume  $L$  is regular
- Let  $M = (\text{make-dfa } K \ \Sigma \ S \ F \ \delta)$  decide  $L$
- Let  $n = |K|$
- The pumping theorem requires picking a  $w \in L$  such that  $|w| \geq n$
- Let  $w = a^n b^n$
- $M$ 's computation on  $w$  must visit at least one state twice
- That is,  $w$  is long enough
- Argue that for any valid choice for  $y$  pumping up or down some number of times results in a word not in  $L$

# Pumping Theorem for RLs

## Theorem

$L = \{a^n b^n \mid n \geq 0\}$  is not regular

- Assume  $L$  is regular
- Let  $M = (\text{make-dfa } K \ \Sigma \ S \ F \ \delta)$  decide  $L$
- Let  $n = |K|$
- The pumping theorem requires picking a  $w \in L$  such that  $|w| \geq n$
- Let  $w = a^n b^n$
- $M$ 's computation on  $w$  must visit at least one state twice
- That is,  $w$  is long enough
- Argue that for any valid choice for  $y$  pumping up or down some number of times results in a word not in  $L$
- We can observe that  $y$  can only contain as
- If it contained any  $b$ s then  $|xy|$  would be too long

# Pumping Theorem for RLs

## Theorem

$L = \{a^n b^n \mid n \geq 0\}$  is not regular

- Assume  $L$  is regular
- Let  $M = (\text{make-dfa } K \ \Sigma \ S \ F \ \delta)$  decide  $L$
- Let  $n = |K|$
- The pumping theorem requires picking a  $w \in L$  such that  $|w| \geq n$
- Let  $w = a^n b^n$
- $M$ 's computation on  $w$  must visit at least one state twice
- That is,  $w$  is long enough
- Argue that for any valid choice for  $y$  pumping up or down some number of times results in a word not in  $L$
- We can observe that  $y$  can only contain as
- If it contained any  $b$ s then  $|xy|$  would be too long
- Thus,  $y = a^j$ , where  $j > 0$
- We may write  $w$  as follows:

$$w = xyz = a^{n-j-r} a^j a^r b^n, \text{ where } x = a^{n-j-r} \wedge z = a^r b^n$$

# Pumping Theorem for RLs

## Theorem

$L = \{a^n b^n \mid n \geq 0\}$  is not regular

- Assume  $L$  is regular
- Let  $M = (\text{make-dfa } K \ \Sigma \ S \ F \ \delta)$  decide  $L$
- Let  $n = |K|$
- The pumping theorem requires picking a  $w \in L$  such that  $|w| \geq n$
- Let  $w = a^n b^n$
- $M$ 's computation on  $w$  must visit at least one state twice
- That is,  $w$  is long enough
- Argue that for any valid choice for  $y$  pumping up or down some number of times results in a word not in  $L$
- We can observe that  $y$  can only contain as
- If it contained any  $b$ s then  $|xy|$  would be too long
- Thus,  $y = a^j$ , where  $j > 0$
- We may write  $w$  as follows:

$$w = xyz = a^{n-j-r} a^r b^n, \text{ where } x = a^{n-j-r} \wedge z = a^r b^n$$

- If we pump up once on  $y$  then we get:

$$w' = a^{n-j-r} a^{2j} a^r b^n = a^{n+j} b^n$$

# Pumping Theorem for RLs

## Theorem

$L = \{a^n b^n \mid n \geq 0\}$  is not regular

- Assume  $L$  is regular
- Let  $M = (\text{make-dfa } K \ \Sigma \ S \ F \ \delta)$  decide  $L$
- Let  $n = |K|$
- The pumping theorem requires picking a  $w \in L$  such that  $|w| \geq n$
- Let  $w = a^n b^n$
- $M$ 's computation on  $w$  must visit at least one state twice
- That is,  $w$  is long enough
- Argue that for any valid choice for  $y$  pumping up or down some number of times results in a word not in  $L$
- We can observe that  $y$  can only contain as
- If it contained any  $b$ s then  $|xy|$  would be too long
- Thus,  $y = a^j$ , where  $j > 0$
- We may write  $w$  as follows:

$$w = xyz = a^{n-j-r} a^r b^n, \text{ where } x = a^{n-j-r} \wedge z = a^r b^n$$

- If we pump up once on  $y$  then we get:

$$w' = a^{n-j-r} a^{2j} a^r b^n = a^{n+j} b^n$$

- Clearly,  $w' \notin L$

# Pumping Theorem for RLs

## Theorem

$L = \{a^n b^n \mid n \geq 0\}$  is not regular

- Assume  $L$  is regular
- Let  $M = (\text{make-dfa } K \ \Sigma \ S \ F \ \delta)$  decide  $L$
- Let  $n = |K|$
- The pumping theorem requires picking a  $w \in L$  such that  $|w| \geq n$
- Let  $w = a^n b^n$
- $M$ 's computation on  $w$  must visit at least one state twice
- That is,  $w$  is long enough
- Argue that for any valid choice for  $y$  pumping up or down some number of times results in a word not in  $L$
- We can observe that  $y$  can only contain as
- If it contained any  $b$ s then  $|xy|$  would be too long
- Thus,  $y = a^j$ , where  $j > 0$
- We may write  $w$  as follows:

$$w = xyz = a^{n-j-r} a^r b^n, \text{ where } x = a^{n-j-r} \wedge z = a^r b^n$$

- If we pump up once on  $y$  then we get:  
 $w' = a^{n-j-r} a^{2j} a^r b^n = a^{n+j} b^n$
- Clearly,  $w' \notin L$
- Therefore, the assumption that  $L$  is regular is wrong

# Pumping Theorem for RLs

## Theorem

$L = \{a^n b^m \mid n > m\}$  is not regular.

- Assume L is regular

# Pumping Theorem for RLs

## Theorem

$L = \{a^n b^m \mid n > m\}$  is not regular.

- Assume  $L$  is regular
- Let  $M = (\text{make-dfa } K \ \Sigma \ S \ F \ \delta)$  decide  $L$
- Let  $n = |K|$

# Pumping Theorem for RLs

## Theorem

$L = \{a^n b^m \mid n > m\}$  is not regular.

- Assume  $L$  is regular
- Let  $M = (\text{make-dfa } K \ \Sigma \ S \ F \ \delta)$  decide  $L$
- Let  $n = |K|$
- Let  $w = a^{n+1}b^n$

# Pumping Theorem for RLs

## Theorem

$L = \{a^n b^m \mid n > m\}$  is not regular.

- Assume  $L$  is regular
- Let  $M = (\text{make-dfa } K \ \Sigma \ S \ F \ \delta)$  decide  $L$
- Let  $n = |K|$
- Let  $w = a^{n+1}b^n$
- Clearly,  $M$ 's computation on  $w$  must visit at least one state twice
- We must now argue that there is an  $i \geq 0$  such that pumping up or down  $i$  times results in a word not in  $L$

# Pumping Theorem for RLs

## Theorem

$L = \{a^n b^m \mid n > m\}$  is not regular.

- Assume  $L$  is regular
- Let  $M = (\text{make-dfa } K \ \Sigma \ S \ F \ \delta)$  decide  $L$
- Let  $n = |K|$
- Let  $w = a^{n+1}b^n$
- Clearly,  $M$ 's computation on  $w$  must visit at least one state twice
- We must now argue that there is an  $i \geq 0$  such that pumping up or down  $i$  times results in a word not in  $L$
- The only possibility for  $y$  is  $y = a^p$ , where  $p > 0$

# Pumping Theorem for RLs

## Theorem

$L = \{a^n b^m \mid n > m\}$  is not regular.

- Assume  $L$  is regular
- Let  $M = (\text{make-dfa } K \ \Sigma \ S \ F \ \delta)$  decide  $L$
- Let  $n = |K|$
- Let  $w = a^{n+1}b^n$
- Clearly,  $M$ 's computation on  $w$  must visit at least one state twice
- We must now argue that there is an  $i \geq 0$  such that pumping up or down  $i$  times results in a word not in  $L$
- The only possibility for  $y$  is  $y = a^p$ , where  $p > 0$
- If we pump down once the resulting word is  $w' = a^{n+1-p}b^n$

# Pumping Theorem for RLs

## Theorem

$L = \{a^n b^m \mid n > m\}$  is not regular.

- Assume  $L$  is regular
- Let  $M = (\text{make-dfa } K \ \Sigma \ S \ F \ \delta)$  decide  $L$
- Let  $n = |K|$
- Let  $w = a^{n+1}b^n$
- Clearly,  $M$ 's computation on  $w$  must visit at least one state twice
- We must now argue that there is an  $i \geq 0$  such that pumping up or down  $i$  times results in a word not in  $L$
- The only possibility for  $y$  is  $y = a^p$ , where  $p > 0$
- If we pump down once the resulting word is  $w' = a^{n+1-p}b^n$
- Observe that  $n+1-p \leq n$
- Clearly,  $w'$  is not in  $L$
- Therefore, the assumption that  $L$  is regular is wrong

# Pumping Theorem for RLs

## Theorem

$L = \{w \mid w \in (a b)^* \wedge w \text{ has an equal number of } as \text{ and } bs\}$  is not regular

- It's ok not to use the Pumping Theorem
- Use closure properties

# Pumping Theorem for RLs

## Theorem

$L = \{w \mid w \in (a b)^* \wedge w \text{ has an equal number of } as \text{ and } bs\}$  is not regular

- It's ok not to use the Pumping Theorem
- Use closure properties
- Assume L is regular

# Pumping Theorem for RLs

## Theorem

$L = \{w \mid w \in (a b)^* \wedge w \text{ has an equal number of } as \text{ and } bs\}$  is not regular

- It's ok not to use the Pumping Theorem
- Use closure properties
- Assume L is regular
- Consider the following regular language:

```
L' = (concat-regexp
      (kleenestar-reg-exp (singleton-regexp a))
      (kleenestar-reg-exp (singleton-regexp b)))
```

# Pumping Theorem for RLs

## Theorem

$L = \{w \mid w \in (a b)^* \wedge w \text{ has an equal number of } as \text{ and } bs\}$  is not regular

- It's ok not to use the Pumping Theorem
- Use closure properties
- Assume L is regular
- Consider the following regular language:

```
L' = (concat-regexp
      (kleenestar-reg-exp (singleton-regexp a))
      (kleenestar-reg-exp (singleton-regexp b)))
```

- If L is regular then by closure under intersection  $L \cap L'$  is also regular

# Pumping Theorem for RLs

## Theorem

$L = \{w \mid w \in (a b)^* \wedge w \text{ has an equal number of } as \text{ and } bs\}$  is not regular

- It's ok not to use the Pumping Theorem
- Use closure properties
- Assume  $L$  is regular
- Consider the following regular language:

$$\begin{aligned} L' &= (\text{concat-regexp} \\ &\quad (\text{kleenestar-reg-exp} (\text{singleton-regexp } a)) \\ &\quad (\text{kleenestar-reg-exp} (\text{singleton-regexp } b))) \end{aligned}$$

- If  $L$  is regular then by closure under intersection  $L \cap L'$  is also regular
- However, we have that:  
$$L \cap L' = a^n b^n$$
- $a^n b^n$  is not regular
- Therefore, the assumption that  $L$  is regular must be wrong

# Pumping Theorem for RLs

- HOMEWORK: 2–4, 8–11