Part III: Context-Free Languages

Marco T. Morazán

Seton Hall University

Outline

Context-Free Grammars

- We shall now study languages that are not regular

Context-Free Grammars

- We shall now study languages that are not regular
- A context-free language is generated by a *context-free grammar*

Context-Free Grammars

- We shall now study languages that are not regular
- A context-free language is generated by a *context-free grammar*
- CFG are used to describe the syntax of programming languages:

```
<definition> → (define <identifier> <expression>)
             → (define (<identifier> <params>) <expression>)
    <params> → ϵ
             → <identifier> <parameters>
<expression> → <number>
             → <identifier>
             → (+ <expression> <expression>)
             → (- <expression> <expression>)
             → (* <expression> <expression>)
             → (quotient <expression> <expression>)
             → (remainder <expression> <expression>)
```

Context-Free Grammars

- `A context-free grammar is an instance of (make-cfg N Σ R S)`
- `N` is the set of capital letters in the Roman alphabet representing the nonterminal symbols (i.e., syntactic categories)
- Σ is the set of lowercase symbols in the Roman alphabet called the alphabet (or terminal symbols)
- `S` is the starting nonterminal symbol
- `R` is the set of production rules
  - Each production rule is of the form $(N \rightarrow (N \cup \Sigma \cup \epsilon)^+)$
  - There is a single nonterminal on the left hand side of a production rule
  - There is a symbol consisting of one or more nonterminals, terminals, or ϵ on the right hand side of a production rule

Context-Free Grammars

- `A context-free grammar is an instance of (make-cfg N Σ R S)`
- `N` is the set of capital letters in the Roman alphabet representing the nonterminal symbols (i.e., syntactic categories)
- Σ is the set of lowercase symbols in the Roman alphabet called the alphabet (or terminal symbols)
- `S` is the starting nonterminal symbol
- `R` is the set of production rules
  - Each production rule is of the form $(N \rightarrow (N \cup \Sigma \cup \epsilon)^+)$
  - There is a single nonterminal on the left hand side of a production rule
  - There is a symbol consisting of one or more nonterminals, terminals, or ϵ on the right hand side of a production rule
- A derivation consists of 1 or more derivation steps
- A derivation step is the application of a production rule: $\rightarrow_G$ or $\rightarrow$ if `G` is clear from the context
- Zero or more derivation steps is denoted by $\rightarrow_G^*$ or $\rightarrow^*$ if `G` is clear from the context

Context-Free Grammars

- A context-free grammar is an instance of (make-cfg N Σ R S)
- N is the set of capital letters in the Roman alphabet representing the nonterminal symbols (i.e., syntactic categories)
- Σ is the set of lowercase symbols in the Roman alphabet called the alphabet (or terminal symbols)
- S is the starting nonterminal symbol
- R is the set of production rules
  - Each production rule is of the form $(N \rightarrow (N \cup \Sigma \cup \epsilon)^+)$
  - There is a single nonterminal on the left hand side of a production rule
  - There is a symbol consisting of one or more nonterminals, terminals, or ϵ on the right hand side of a production rule
- A derivation consists of 1 or more derivation steps
- A derivation step is the application of a production rule: $\rightarrow_G$ or $\rightarrow$ if G is clear from the context
- Zero or more derivation steps is denoted by $\rightarrow_G^*$ or $\rightarrow^*$ if G is clear from the context
- L(G) denotes the language generated by G: $\{w \mid w \in \Sigma^* \wedge S \rightarrow_G^* w\}$
- A language, L, is context-free if L = L(G) for some context-free grammar G

Context-Free Grammars

- $L = \{a^n b^n \mid n \geq 0\}$
- Follow the steps of the DR for grammars

Context-Free Grammars

- L = $\{a^n b^n \mid n \geq 0\}$
- Follow the steps of the DR for grammars
- Name = a2nb2n and Σ = {a b}

Context-Free Grammars

- L = $\{a^n b^n \mid n \geq 0\}$
- Follow the steps of the DR for grammars
- Name = a2nb2n and Σ = {a b}
- S, generates all words that start with n as followed by n bs
- How can this be done?

Context-Free Grammars

- L = $\{a^n b^n \mid n \geq 0\}$
- Follow the steps of the DR for grammars
- Name = a2nb2n and Σ = {a b}
- S, generates all words that start with n as followed by n bs
- How can this be done?
- First a has a matching b at the end of the word
- The same is true for the remaining part of the word

Context-Free Grammars

- L = $\{a^n b^n \mid n \geq 0\}$
- Follow the steps of the DR for grammars
- Name = a2nb2n and Σ = {a b}
- S, generates all words that start with n as followed by n bs
- How can this be done?
- First a has a matching b at the end of the word
- The same is true for the remaining part of the word
- This suggests S ought to generate ϵ
- It also suggests that nonempty words in L may be generated from the outside in

Context-Free Grammars

- L = $\{a^n b^n \mid n \geq 0\}$
- Follow the steps of the DR for grammars
- Name = a2nb2n and Σ = {a b}
- S, generates all words that start with n as followed by n bs
- How can this be done?
- First a has a matching b at the end of the word
- The same is true for the remaining part of the word
- This suggests S ought to generate ϵ
- It also suggests that nonempty words in L may be generated from the outside in
- Generate an a as the first letter in the word, generate a b for the last letter in the word, and use S to generate the rest of the word in the middle

Context-Free Grammars

- Production Rules

```
`(S ,ARROW ,EMP)

`(S ,ARROW aSb)
```

Context-Free Grammars

- Production Rules

```
`(S ,ARROW ,EMP)

`(S ,ARROW aSb)
```

- Tests

```
;; Tests for a2nb2n
#:rejects '((b b b) (a b a))
#:accepts '(() (a b) (a a b b) (a a a b b b) (a a a a a b b b b b))
     Both methods are acceptable
(check-not-derive? a2nb2n '(b b b))
(check-not-derive? a2nb2n '(a b a))
(check-derive? a2nb2n '())
(check-derive? a2nb2n '(a b))
(check-derive? a2nb2n '(a a b b))
(check-derive? a2nb2n '(a a a b b b))
(check-derive? a2nb2n '(a a a a a b b b b b))
```

Context-Free Grammars

- ```
  #lang fsm
  ;; L = a^nb^n Syntactic Categories
  ;;  S = words that start with n a and end with n b
  (define a2nb2n
    (make-cfg '(S)
              '(a b)
              `((S ,ARROW ,EMP)
                (S ,ARROW aSb))
              'S
              #:rejects '((b b b) (a b a))
              #:accepts '(() (a b) (a a b b) (a a a b b b)
                             (a a a a a b b b b))))
  ```
  ```

Context-Free Grammars

- ```
  #lang fsm
  ;; L = a^nb^n Syntactic Categories
  ;;  S = words that start with n a and end with n b
  (define a2nb2n
    (make-cfg '(S)
              '(a b)
              `((S ,ARROW ,EMP)
                (S ,ARROW aSb))
              'S
              #:rejects '((b b b) (a b a))
              #:accepts '(() (a b) (a a b b) (a a a b b b)
                             (a a a a a b b b b b))))
  ```
- Illustrate derivation tree using grammar-viz
  ```

Context-Free Grammars

- ```
  > (grammar-test a2nb2n 10)
  '(((b a b b b a) "(b a b b b a) is not in L(G)")
    (() "The word () is too short to test.")
    ((a a b b) (S -> aSb -> aaSbb -> aabb))
    ((a a) "(a a) is not in L(G)")
    ((a a b b b b b a b) "(a a b b b b b a b) is not in L(G)")
    ((a a a b a a) "(a a a b a a) is not in L(G)")
    ((a a a b b a a) "(a a a b b a a) is not in L(G)")
    ((a b) (S -> aSb -> ab))
    ((a) "The word (a) is too short to test.")
    ((b b b b) "(b b b b) is not in L(G)"))
  ```
- Be mindful that trying to randomly generate words in the language is not easy
- Most of the time a word that is too short or that is not in the language is generated
- It is important for unit tests to be thorough
  ```

Context-Free Grammars

- ```
  #lang fsm
  ;; L = a^nb^n Syntactic Categories
  ;;  S = words that start with n a and end with n b
  (define a2nb2n
    (make-cfg '(S)
              '(a b)
              `((S ,ARROW ,EMP)
                (S ,ARROW aSb))
              'S
              #:rejects '((b b b) (a b a))
              #:accepts '(() (a b) (a a b b) (a a a b b b)
                             (a a a a a b b b b b))))
  ```
  ```

Context-Free Grammars

- ```
  #lang fsm
  ;; L = a^nb^n Syntactic Categories
  ;;  S = words that start with n a and end with n b
  (define a2nb2n
    (make-cfg '(S)
              '(a b)
              `((S ,ARROW ,EMP)
                (S ,ARROW aSb))
              'S
              #:rejects '((b b b) (a b a))
              #:accepts '(() (a b) (a a b b) (a a a b b b)
                             (a a a a a b b b b b))))
  ```
- ```
  ;; word --> Boolean
  ;; Purpose: Determine if given word is in L(a2nb2n)
  (define (S-INV w)
    (let* [(as (takef w (λ (s) (eq? s 'a))))
           (bs (drop w (length as)))]
      (and (equal? w (append as bs))
           (= (length as) (length bs)))))
  (check-equal? (S-INV '(a)) #f)
  (check-equal? (S-INV '(b b a a)) #f)
  (check-equal? (S-INV '(a a a b b)) #f)
  (check-equal? (S-INV '()) #t)
  (check-equal? (S-INV '(a a b b)) #t)
  (check-equal? (S-INV '(a b)) #t)
  ```

Context-Free Grammars

- Proof invariants hold by induction on, h, the height of the derivation tree.

Context-Free Grammars

- Proof invariants hold by induction on, h, the height of the derivation tree.
- Base Case: h=1

  This means the only rule used is (S ,ARROW ,EMP). Observe that 0 as followed by 0 bs is generated. Thus, S-INV holds.

Context-Free Grammars

- Proof invariants hold by induction on, h, the height of the derivation tree.
- Base Case: h=1

  This means the only rule used is (S ,ARROW ,EMP). Observe that 0 as followed by 0 bs is generated. Thus, S-INV holds.
- Inductive Step

  Assume: invariants hold for h=k
  Show: invariants hold for h=k+1

Context-Free Grammars

- Proof invariants hold by induction on, h, the height of the derivation tree.
- Base Case: h=1

  This means the only rule used is (S ,ARROW ,EMP). Observe that 0 as followed by 0 bs is generated. Thus, S-INV holds.
- Inductive Step

  Assume: invariants hold for h=k
  Show: invariants hold for h=k+1
- k>=0 ==> k+1>0

  This means that (S ,ARROW aSb) is used to derive the word. By inductive hypothesis, the S in aSb generates a^rb^r. Using this rule adds an a to the front and a b to the back yielding a^r+1b^r+1. Thus, S-INV holds.

Context-Free Grammars

- Proof invariants hold by induction on, h, the height of the derivation tree.
- Base Case: h=1

  This means the only rule used is (S ,ARROW ,EMP). Observe that 0 as followed by 0 bs is generated. Thus, S-INV holds.
- Inductive Step

  Assume: invariants hold for h=k
  Show: invariants hold for h=k+1
- k>=0 ==> k+1>0

  This means that (S ,ARROW aSb) is used to derive the word. By inductive hypothesis, the S in aSb generates a^rb^r. Using this rule adds an a to the front and a b to the back yielding a^r+1b^r+1. Thus, S-INV holds.
- L=L(a2nb2n)

Context-Free Grammars

- Proof invariants hold by induction on, h, the height of the derivation tree.
- Base Case: h=1

  This means the only rule used is (S ,ARROW ,EMP). Observe that 0 as followed by 0 bs is generated. Thus, S-INV holds.
- Inductive Step

  Assume: invariants hold for h=k
  Show: invariants hold for h=k+1
- k>=0 ==> k+1>0

  This means that (S ,ARROW aSb) is used to derive the word. By inductive hypothesis, the S in aSb generates a^rb^r. Using this rule adds an a to the front and a b to the back yielding a^r+1b^r+1. Thus, S-INV holds.
- L=L(a2nb2n)

- w∈L <==> w∈L(a2nb2n)
  (⇒) Assume w∈L
  This means w=$a^n b^n$. Given that invariants always hold, w is generated using a2nb2n by first applying (S ,ARROW aSb) r times and then using (S ,ARROW ,EMP). Thus, w∈L(a2nb2n).

Context-Free Grammars

- Proof invariants hold by induction on, h, the height of the derivation tree.
- Base Case: h=1

  This means the only rule used is (S ,ARROW ,EMP). Observe that 0 as followed by 0 bs is generated. Thus, S-INV holds.
- Inductive Step

  Assume: invariants hold for h=k
  Show: invariants hold for h=k+1
- k>=0 ==> k+1>0

  This means that (S ,ARROW aSb) is used to derive the word. By inductive hypothesis, the S in aSb generates a^rb^r. Using this rule adds an a to the front and a b to the back yielding a^r+1b^r+1. Thus, S-INV holds.
- L=L(a2nb2n)

- w∈L <==> w∈L(a2nb2n)
  (⇒) Assume w∈L
  This means w=$a^n b^n$. Given that invariants always hold, w is generated using a2nb2n by first applying (S ,ARROW aSb) r times and then using (S ,ARROW ,EMP). Thus, w∈L(a2nb2n).
- (⇐) Assume w∈L(a2nb2n)
  Given that invariants always hold, this means w=$a^n b^n$. Thus, w∈L.

Context-Free Grammars

- Proof invariants hold by induction on, h, the height of the derivation tree.
- Base Case: h=1

  This means the only rule used is (S ,ARROW ,EMP). Observe that 0 as followed by 0 bs is generated. Thus, S-INV holds.
- Inductive Step

  Assume: invariants hold for h=k
  Show: invariants hold for h=k+1
- k>=0 ==> k+1>0

  This means that (S ,ARROW aSb) is used to derive the word. By inductive hypothesis, the S in aSb generates a^rb^r. Using this rule adds an a to the front and a b to the back yielding a^r+1b^r+1. Thus, S-INV holds.
- L=L(a2nb2n)

- w∈L <==> w∈L(a2nb2n)
  (⇒) Assume w∈L
  This means w=$a^n b^n$. Given that invariants always hold, w is generated using a2nb2n by first applying (S ,ARROW aSb) r times and then using (S ,ARROW ,EMP). Thus, w∈L(a2nb2n).
- (⇐) Assume w∈L(a2nb2n)
  Given that invariants always hold, this means w=$a^n b^n$. Thus, w∈L.
- w∉L <==> w∉in L(a2nb2n)
  (⇒) Assume w∉L
  This means w!=$a^n b^n$. Given that invariants always hold, S cannot generate w. Thus, w∉L(a2nb2n).

Context-Free Grammars

- Proof invariants hold by induction on, h, the height of the derivation tree.
- Base Case: h=1

  This means the only rule used is (S ,ARROW ,EMP). Observe that 0 as followed by 0 bs is generated. Thus, S-INV holds.
- Inductive Step

  Assume: invariants hold for h=k
  Show: invariants hold for h=k+1
- k>=0 ==> k+1>0

  This means that (S ,ARROW aSb) is used to derive the word. By inductive hypothesis, the S in aSb generates a^rb^r. Using this rule adds an a to the front and a b to the back yielding a^r+1b^r+1. Thus, S-INV holds.
- L=L(a2nb2n)

- w∈L <==> w∈L(a2nb2n)
  (⇒) Assume w∈L
  This means w=$a^n b^n$. Given that invariants always hold, w is generated using a2nb2n by first applying (S ,ARROW aSb) r times and then using (S ,ARROW ,EMP). Thus, w∈L(a2nb2n).
- (⇐) Assume w∈L(a2nb2n)
  Given that invariants always hold, this means w=$a^n b^n$. Thus, w∈L.
- w∉L <==> w∉in L(a2nb2n)
  (⇒) Assume w∉L
  This means w!=$a^n b^n$. Given that invariants always hold, S cannot generate w. Thus, w∉L(a2nb2n).
- (⇐) Assume w∉L(a2nb2n)
  Given that invariants always hold, this means w!=$a^n b^n$. Thus, w∉L.

Context-Free Grammars

- $\texttt{L} = \{\texttt{w} \mid \texttt{w} \in (\texttt{a b})^* \wedge \texttt{w has more bs than as}\}$

Context-Free Grammars

- $\texttt{L} = \{\texttt{w} \mid \texttt{w} \in \texttt{(a b)}^* \wedge \texttt{w has more bs than as}\}$
- Every word in the language must have at least one more `b` than `a`s
- Suggests that at least a `b` must be generated
- After that, for every `a` generated there must be one or more `b`s generated

Context-Free Grammars

- `#lang fsm`

Context-Free Grammars

- `#lang fsm`
- ```
  ;; Syntactic Categories
  ;;  S = words such that number of b > number of a
  ```
  ```

Context-Free Grammars

- `#lang fsm`
- `;; Syntactic Categories`
  `;;  S = words such that number of b > number of a`
- `;;  A = words such that number of b >= number of a`

Context-Free Grammars

```
#lang fsm
;; Syntactic Categories
;;  S = words such that number of b > number of a
;;  A = words such that number of b >= number of a
;; L = w | w in (a b)* AND  w has more b than a
(define numb>numa
  (make-cfg '(S A)
            '(a b)
            'S))
```

Context-Free Grammars

```
#lang fsm
;; Syntactic Categories
;;  S = words such that number of b > number of a
     ;;  A = words such that number of b >= number of a
;; L = w | w in (a b)* AND  w has more b than a
(define numb>numa
  (make-cfg '(S A)
            '(a b)
            `((S ,ARROW b)
              (S ,ARROW AbA)
              (A ,ARROW AaAbA)
              (A ,ARROW AbAaA)
              (A ,ARROW ,EMP)
              (A ,ARROW bA))
            'S))
```

Context-Free Grammars

```
#lang fsm
;; Syntactic Categories
;;  S = words such that number of b > number of a
     ;;  A = words such that number of b >= number of a
;; L = w | w in (a b)* AND  w has more b than a
(define numb>numa
  (make-cfg '(S A)
            '(a b)
            `((S ,ARROW b)
              (S ,ARROW AbA)
              (A ,ARROW AaAbA)
              (A ,ARROW AbAaA)
              (A ,ARROW ,EMP)
              (A ,ARROW bA))
            #:rejects '((a b) (a b a) (a a a a a))
            #:accepts '((b b b) (b b a b a a b) (a a a b b b b))
            'S))
```

Context-Free Grammars

- ```
  > (grammar-test numb>numa 5)
  '(((a b b b a b b b b)
     (S -> AbA -> AaAbAbA -> AaAbAaAbAbA -> aAbAaAbAbA
      -> abAaAbAbA -> abbAaAbAbA -> abbbAaAbAbA -> abbbaAbAbA
      -> abbbabAbA -> abbbabbA -> abbbabbbA ->  abbbabbbbA
      -> abbbabbbb))
    ((a a b b b)
     (S -> AbA -> AaAbAbA -> aAbAbA -> aAaAbAbAbA -> aaAbAbAbA
      -> aabAbAbA -> aabbAbA -> aabbbA -> aabbb))
    (() "The word () is too short to test.")
    ((a b b a b a a b) "(a b b a b a a b) is not in L(G)")
    ((a b b b b a b)
     (S -> AbA -> AaAbAbA -> aAbAbA -> abAbA -> abAbAaAbA
      -> abbAaAbA -> abbbAaAbA -> abbbbAaAbA -> abbbbaAbA
      -> abbbbabA -> abbbbab)))
  ```
- Be mindful about tests taking very long to run
- Interrupt execution if necessary
  ```

Context-Free Grammars

- HOMEWORK: 1, 3, 5, 8

Context-Free Grammars

- Are all regular languages context-free?

Context-Free Grammars

- Are all regular languages context-free?

Theorem

All regular languages are context-free.

-

Context-Free Grammars

- Are all regular languages context-free?

Theorem

All regular languages are context-free.

-

Proof.

- Assume L is a regular language. This means that there exist a regular grammar:

  `G = (make-rg N Σ P S),`

such that L = L(G). All the production rules in `P` are of the form:

1. $\texttt{A} \rightarrow \epsilon$
2. $\texttt{A} \rightarrow \texttt{a}$, where $\texttt{a} \in \Sigma$
3. $\texttt{A} \rightarrow \texttt{aB}$, where $\texttt{a} \in \Sigma \wedge \texttt{B} \in \texttt{N}$

This means $\texttt{P} \subset (\texttt{N} \rightarrow (\texttt{N} \cup \Sigma \cup \epsilon)^{+})$. That is, $\texttt{p} \in \texttt{P}$ is a valid production rule for a context-free grammar. Given that `G`'s `N`, Σ, and `S` may also be used in a `cfg`, we may conclude that `L` is a context-free language. □

Context-Free Grammars

```
(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))
```

Context-Free Grammars

- ```
  (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))
  ```
- ```
  (define MULT3-as (make-cfg '(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))
  ```

Context-Free Grammars

- Consider the language of words containing balanced parenthesis

Context-Free Grammars

- Consider the language of words containing balanced parenthesis
- Opening parenthesis: o and closing parenthesis: c

Context-Free Grammars

- Consider the language of words containing balanced parenthesis
- Opening parenthesis: `o` and closing parenthesis: `c`
- `#lang fsm`

Context-Free Grammars

- Consider the language of words containing balanced parenthesis
- Opening parenthesis: o and closing parenthesis: c
- `#lang fsm`

- ```
  ;; Syntactic categories: S = words with balanced parenthesis
  ;; L = {w | w has balanced parenthesis}, where o = ( and c = )
  (define BP (make-cfg '(S)
                       '(o c)
  ```

- ```
                       'S))
  ```

Context-Free Grammars

- Consider the language of words containing balanced parenthesis
- Opening parenthesis: o and closing parenthesis: c
- `#lang fsm`

- ```
  ;; Syntactic categories: S = words with balanced parenthesis
  ;; L = {w | w has balanced parenthesis}, where o = ( and c = )
  (define BP (make-cfg '(S)
  ```
- ```
                       '(o c)
                       `((S ,ARROW ,EMP)
                         (S ,ARROW SS)
                         (S ,ARROW oSc))
  ```
- ```
                       'S))
  ```
  ```

Context-Free Grammars

- Consider the language of words containing balanced parenthesis
- Opening parenthesis: o and closing parenthesis: c
- `#lang fsm`

- ```
  ;; Syntactic categories: S = words with balanced parenthesis
  ;; L = {w | w has balanced parenthesis}, where o = ( and c = )
  (define BP (make-cfg '(S)
                       '(o c)
  ```
- ```
                       `((S ,ARROW ,EMP)
                         (S ,ARROW SS)
                         (S ,ARROW oSc))
  ```
- ```
                       'S))
  ```
- ```
  #;(check-not-derive? BP infinite derivations due to (S ,ARROW SS)
                       (parensstr->los "))((")
                       (parensstr->los "()("))
  (check-derive? BP (parensstr->los "")
                    (parensstr->los "()")
                    (parensstr->los "(())()"))
  ```

Context-Free Grammars

- Consider the language of words containing balanced parenthesis
- Opening parenthesis: o and closing parenthesis: c
- `#lang fsm`
- ```
  ;; string → (listof (o or c)) throws error
  ;; Purpose: Converts given string of parens to a list of parens
  (define (parensstr->los s)
    (map (λ (p)
           (cond [(eq? p #\( ) 'o]
                 [(eq? p #\) ) 'c]
                 [else (error (format "parensstr->los: non-parens symbol in: ~s"
                                      s))]))
         (string->list s)))
  ;; Tests for parensstr->los
  (check-equal? (parensstr->los "") '())
  (check-equal? (parensstr->los "(())()") '(o o c c o c))
  ```
- ```
  ;; Syntactic categories: S = words with balanced parenthesis
  ;; L = {w | w has balanced parenthesis}, where o = ( and c = )
  (define BP (make-cfg '(S)
                       '(o c)
  ```
- ```
                       `((S ,ARROW ,EMP)
                         (S ,ARROW SS)
                         (S ,ARROW oSc))
  ```
- ```
                       'S))
  ```
- ```
  #;(check-not-derive? BP infinite derivations due to (S ,ARROW SS)
                       (parensstr->los "))((")
                       (parensstr->los "()("))
  (check-derive? BP (parensstr->los "")
                    (parensstr->los "()")
                    (parensstr->los "(())()"))
  ```
  ```

Context-Free Grammars

- Consider deriving the `"(()())"`:

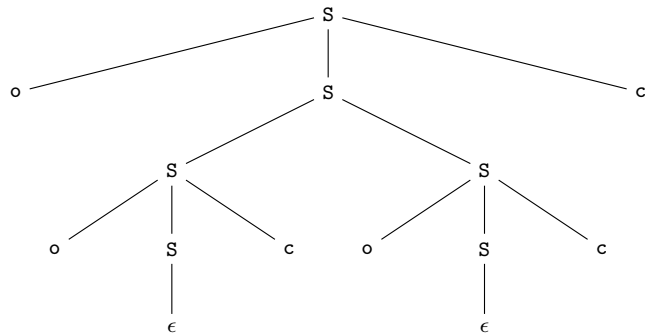
```
'(S -> oSc -> oSSc -> ooScSc -> oocSc -> oocoScc -> oococc)
```

Context-Free Grammars

- Consider deriving the `"(()())"`:

  ```
  '(S -> oSc -> oSSc -> ooScSc -> oocSc -> oocoScc -> oococc)
  ```

- A derivation may be visualized as a tree:



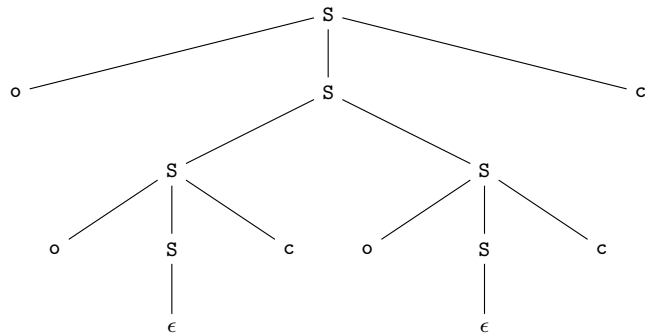
- Such a visualization is called a parse tree

Context-Free Grammars

- Consider deriving the `"(()())"`:

  `'(S -> oSc -> oSSc -> ooScSc -> oocSc -> oocoScc -> oococc)`

- A derivation may be visualized as a tree:

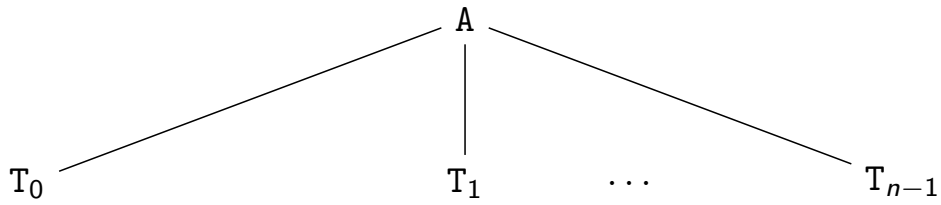


- Such a visualization is called a parse tree
- The leaves are terminal symbols or ϵ
- The interior nodes are rooted at a nonterminal
- The edges out of an interior node indicate what is generated from the nonterminal
- The *yield* of a parse tree is the concatenation of all the leaves from left to right

Context-Free Grammars

- A parse tree is formally defined as follows:
  - For each $a \in \Sigma$ the following is a parse tree: a
  - For $A \rightarrow \epsilon$ the following is a parse tree:

  - If $T_0 T_1 \ldots T_{n-1}$ are the parse trees for $a_0 \ldots a_{n-1}$, where a_i is a nonterminal or a terminal symbol, then for $A \rightarrow a_0 a_1 \ldots a_{n-1}$ the following is a parse tree:



Context-Free Grammars

- Parse trees represent derivations eliminating irrelevant differences like the order in which production rules are applied
- Two derivations are similar if they are captured by the same parse tree:

```
S -> oSc -> oSSc -> ooScSc -> oocSc -> oocoScc -> oococc
S -> oSc -> oSSc -> oSoScc -> oSocc -> ooScocc -> oococc
```

Context-Free Grammars

- Parse trees represent derivations eliminating irrelevant differences like the order in which production rules are applied
- Two derivations are similar if they are captured by the same parse tree:

```
S -> oSc -> oSSc -> ooScSc -> oocSc -> oocoScc -> oococc
S -> oSc -> oSSc -> oSoScc -> oSocc -> ooScocc -> oococc
```

- Both of these derivations are captured by the previous parse tree
- Use the same production rules on the same nonterminals. The only difference is the order in which the production rules are used

Context-Free Grammars

- Parse trees represent derivations eliminating irrelevant differences like the order in which production rules are applied
- Two derivations are similar if they are captured by the same parse tree:

```
S -> oSc -> oSSc -> ooScSc -> oocSc -> oocoScc -> oococc
S -> oSc -> oSSc -> oSoScc -> oSocc -> ooScocc -> oococc
```

- Both of these derivations are captured by the previous parse tree
- Use the same production rules on the same nonterminals. The only difference is the order in which the production rules are used
- The first is a *leftmost derivation*
- The second is a *rightmost derivation*
- Every parse tree has a unique leftmost and a unique rightmost derivation

Context-Free Grammars

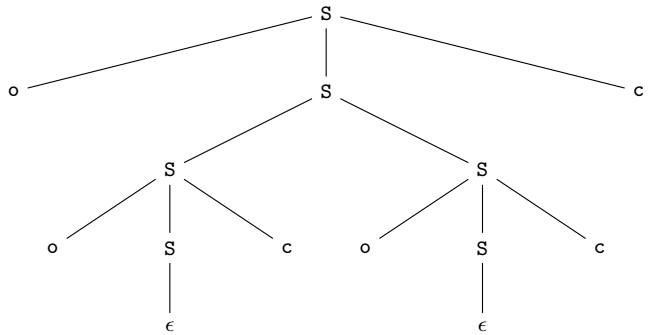
- Equivalent:
  - $\mathtt{A} \rightarrow_G^* \mathtt{w}$
  - $\exists$ a parse tree with root $\mathtt{A}$ and yield $\mathtt{w}$
  - $\exists$ a leftmost derivation $\mathtt{A} \overset{L}{\rightarrow}_G^* \mathtt{w}$
  - $\exists$ a rightmost derivation $\mathtt{A} \overset{R}{\rightarrow}_G^* \mathtt{w}$

Context-Free Grammars

- Not all derivations of a word may be similar

```
S -> SS -> S -> oSSc -> ooScSc -> oocSc -> oocoScc -> oococc
```

- Not captured by:

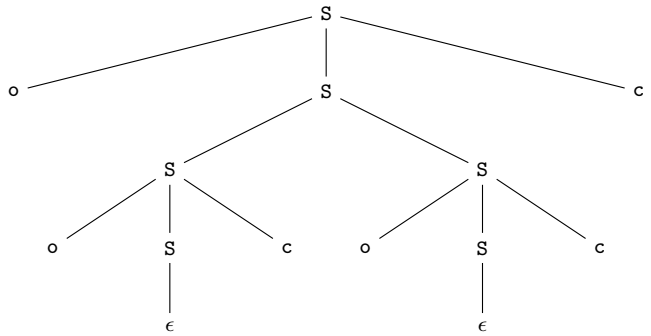


Context-Free Grammars

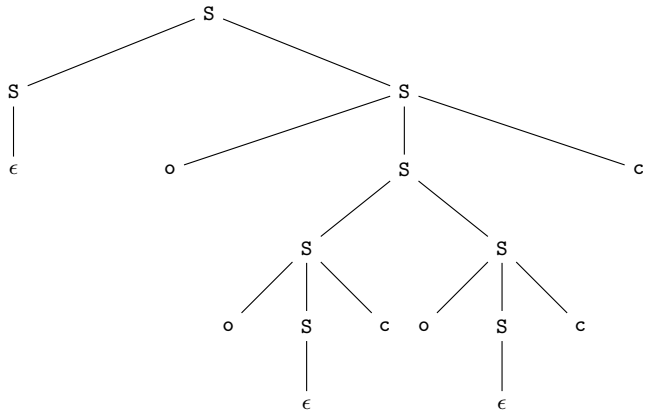
- Not all derivations of a word may be similar

```
S -> SS -> S -> oSSc -> ooScSc -> oocSc -> oocoScc -> oococc
```

- Not captured by:



- Captured by:

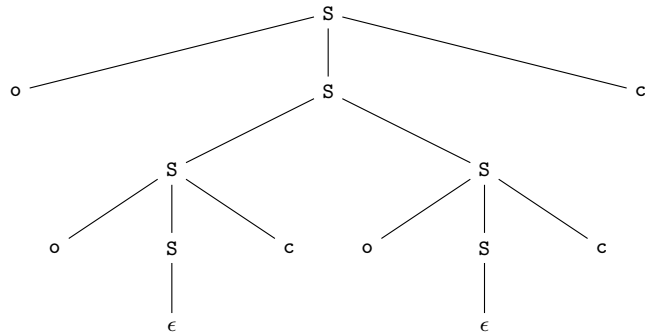


Context-Free Grammars

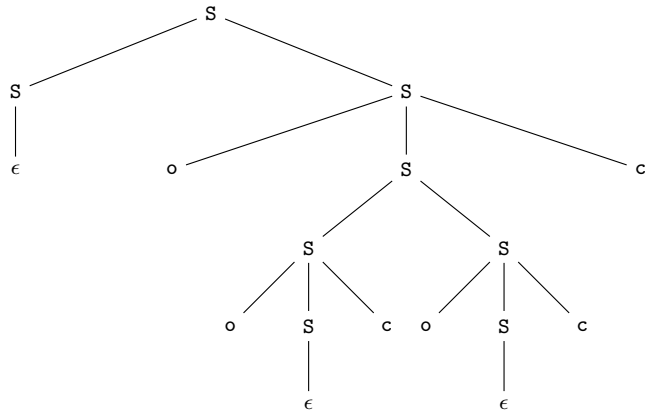
- Not all derivations of a word may be similar

  `S -> SS -> S -> oSSc -> ooScSc -> oocSc -> oocoScc -> oococc`

- Not captured by:



- Captured by:



- A `cfg`, `G`, is called *ambiguous* if for any `w`$\in$`L(G)` there are 2 or more parse trees
- `BP` is ambiguous

Context-Free Grammars

- Ambiguity is a problem when meaning must be assigned to a word as done in the implementation of programming languages

Context-Free Grammars

- Ambiguity is a problem when meaning must be assigned to a word as done in the implementation of programming languages
- Arithmetic expressions grammar:

```
#lang fsm
(define AE (make-cfg '(S I)
                     '(p t x y z)
                     `((S ,ARROW SpS)
                       (S ,ARROW StS)
                       (S ,ARROW I)
                       (I ,ARROW x)
                       (I ,ARROW y)
                       (I ,ARROW z))
                     'S))
;; Tests for AE
(check-derive? AE '(x p x)
                  '(x t z)
                  '(x t z p y)
                  '(x p z t z p y))
```

- p stands for +, t stands for *
- x, y, and z are variables

Context-Free Grammars

- `'(x t z p y)`, has two derivations that are not similar:

```
S -> SpS -> StSpS -> ItSpS -> xtSpS -> xtIpS -> xtzpS
  -> xtzpI -> xtzpy

S -> StS -> ItS -> xtS -> xtSpS -> xtIpS -> xtzpS -> xtzpI
  -> xtzpy
```

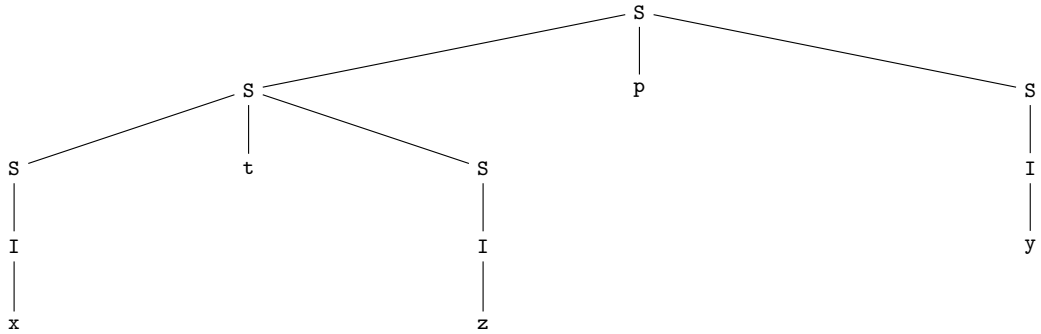
Context-Free Grammars

- `'(x t z p y)`, has two derivations that are not similar:

```
S -> SpS -> StSpS -> ItSpS -> xtSpS -> xtIpS -> xtzpS
  -> xtzpI -> xtzpy

S -> StS -> ItS -> xtS -> xtSpS -> xtIpS -> xtzpS -> xtzpI
  -> xtzpy
```

- Parse trees:



- Suggests that `x` and `z` are multiplied and then their product and `y` are added

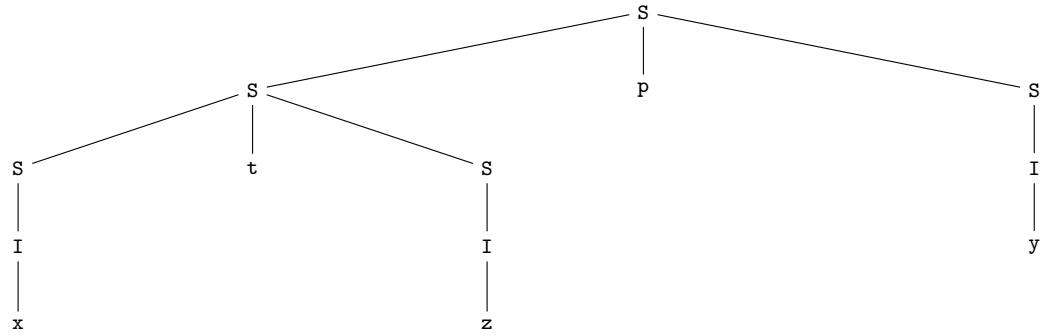
Context-Free Grammars

- `'(x t z p y)`, has two derivations that are not similar:

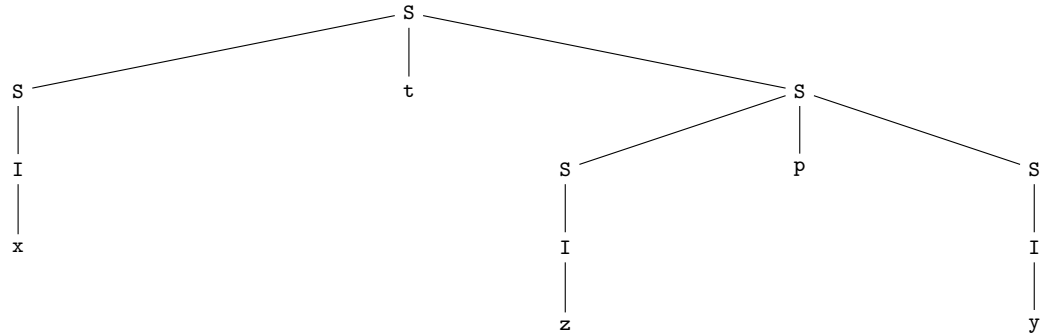
```
S -> SpS -> StSpS -> ItSpS -> xtSpS -> xtIpS -> xtzpS
  -> xtzpI -> xtzpy

S -> StS -> ItS -> xtS -> xtSpS -> xtIpS -> xtzpS -> xtzpI
  -> xtzpy
```

- Parse trees:



- Suggests that `x` and `z` are multiplied and then their product and `y` are added



- This parse tree suggests that first `z` and `y` are added and then their sum is multiplied by `x`

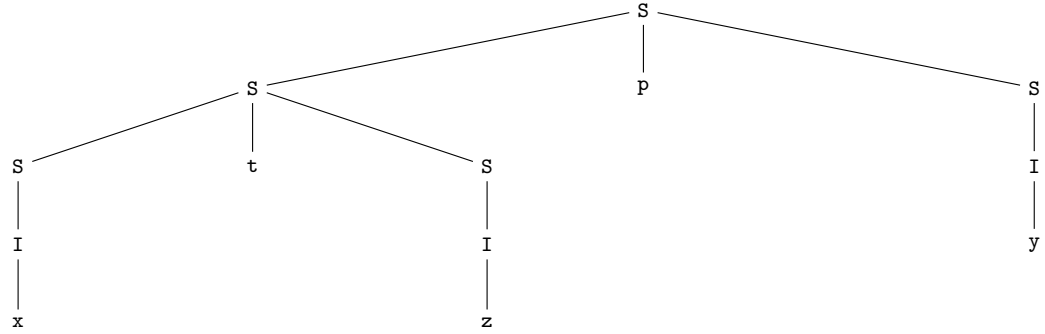
Context-Free Grammars

- `'(x t z p y)`, has two derivations that are not similar:

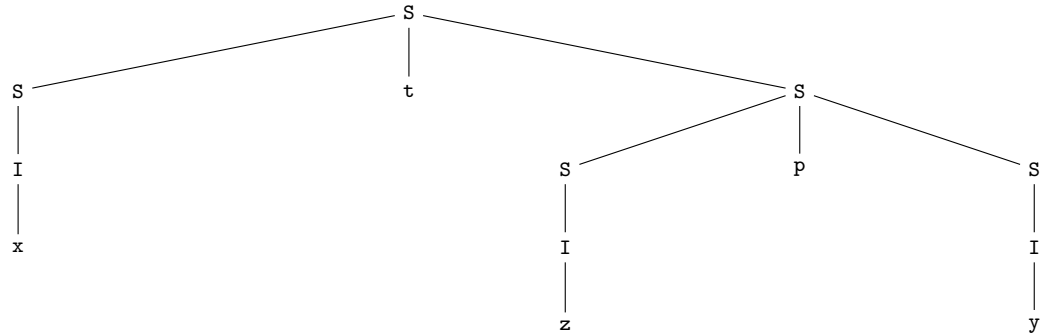
```
S -> SpS -> StSpS -> ItSpS -> xtSpS -> xtIpS -> xtzpS
  -> xtzpI -> xtzpy

S -> StS -> ItS -> xtS -> xtSpS -> xtIpS -> xtzpS -> xtzpI
  -> xtzpy
```

- Parse trees:



- Suggests that `x` and `z` are multiplied and then their product and `y` are added



- This parse tree suggests that first `z` and `y` are added and then their sum is multiplied by `x`
- A word may have different meanings when a grammar is ambiguous

Context-Free Grammars

- Disambiguate `AE`: first produce the summing subexpressions, if any, and then produce the product subexpressions

Context-Free Grammars

- Disambiguate `AE`: first produce the summing subexpressions, if any, and then produce the product subexpressions
- A new syntactic category, `T`, is introduced to generate product subexpressions
- A product subexpression may only be generated after summing subexpressions, if any, are generated:
-
```
#lang fsm
(define AE2 (make-cfg '(S I T)
                      '(p t x y z)
                      `((S ,ARROW SpS)
                        (S ,ARROW T)
                        (S ,ARROW I)
                        (T ,ARROW TtT)
                        (T ,ARROW I)
                        (I ,ARROW x)
                        (I ,ARROW y)
                        (I ,ARROW z))
                      'S))
;; Tests for AE2
(check-derive? AE2 '(x p x)
                   '(x t z)
                   '(x t z p y)
                   '(x p z t z p y))
```

Context-Free Grammars

- Disambiguate `AE`: first produce the summing subexpressions, if any, and then produce the product subexpressions
- A new syntactic category, `T`, is introduced to generate product subexpressions
- A product subexpression may only be generated after summing subexpressions, if any, are generated:
- ```
  #lang fsm
  (define AE2 (make-cfg '(S I T)
                        '(p t x y z)
                        `((S ,ARROW SpS)
                          (S ,ARROW T)
                          (S ,ARROW I)
                          (T ,ARROW TtT)
                          (T ,ARROW I)
                          (I ,ARROW x)
                          (I ,ARROW y)
                          (I ,ARROW z))
                        'S))
  ;; Tests for AE2
  (check-derive? AE2 '(x p x)
                     '(x t z)
                     '(x t z p y)
                     '(x p z t z p y))
  ```
- Unfortunately, some context-free languages can only be generated by an ambiguous `cfg`
  ```

Context-Free Grammars

- HOMEWORK: 10, 12, 14
- Quiz: in class (1 week)

Pushdown Automata

- Would like to have a machine that decides if a given word is a member of a CFL
- What should such a machine look like?

Pushdown Automata

- Would like to have a machine that decides if a given word is a member of a CFL
- What should such a machine look like?
- Write a program to decide $a^n b^n$
- Call an auxiliary function that takes as input `w`'s sub-word without the leading as, if any, and an accumulator with `w`'s leading as
- 3 conditions:
  1. If the given word is empty then the value from testing if the given accumulator is empty is returned.
  2. If the first element of the given word is a then false is returned.
  3. Otherwise, return the conjunction of testing if the accumulator is not empty and checking the rest of both the given word and the given accumulator.

Pushdown Automata

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

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

Pushdown Automata

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

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

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

Pushdown Automata

- ```
  #lang fsm
  ;; word → Boolean
  ;; Purpose: Decide if given word is in a^nb^n
  (define (is-in-a^nb^n? w)
  ```
- ```
    ;; word (listof symbol) → Boolean
    ;; Purpose: Determine if first given word has only bs that
    ;;          match as in the second given word
    ;; Accumulator Invariant
    ;;   acc = the unmatched as at the beginning of w
    ;; Assume: w in (a b)*
    (define (check wrd acc)
      (cond [(empty? wrd) (empty? acc)]
            [(eq? (first wrd) 'a) #f]
            [else (and (not (empty? acc))
                       (check (rest wrd) (rest acc)))]))
  ```
- ```
  (check (dropf w (λ (s) (eq? s 'a)))   ;; everything after initial as
         (takef w (λ (s) (eq? s 'a)))) ;; the initial as
  ```
- ```
  ;; Tests for is-in-anbn?
  (check-pred (λ (w) (not (is-in-a^nb^n? w))) '(a))
  (check-pred (λ (w) (not (is-in-a^nb^n? w))) '(b b))
  (check-pred (λ (w) (not (is-in-a^nb^n? w))) '(a b b))
  (check-pred (λ (w) (not (is-in-a^nb^n? w))) '(a b a a b b))
  (check-pred is-in-a^nb^n? '())
  (check-pred is-in-a^nb^n? '(a a b b))
  ```

Pushdown Automata

- ```
  #lang fsm
  ;; word → Boolean
  ;; Purpose: Decide if given word is in a^nb^n
  (define (is-in-a^nb^n? w)
  ```
- ```
    ;; word (listof symbol) → Boolean
    ;; Purpose: Determine if first given word has only bs that
    ;;          match as in the second given word
    ;; Accumulator Invariant
    ;;   acc = the unmatched as at the beginning of w
    ;; Assume: w in (a b)*
    (define (check wrd acc)
      (cond [(empty? wrd) (empty? acc)]
            [(eq? (first wrd) 'a) #f]
            [else (and (not (empty? acc))
                       (check (rest wrd) (rest acc)))]))
  ```
- ```
  (check (dropf w (λ (s) (eq? s 'a)))   ;; everything after initial as
         (takef w (λ (s) (eq? s 'a)))) ;; the initial as
  ```
- ```
  ;; Tests for is-in-anbn?
  (check-pred (λ (w) (not (is-in-a^nb^n? w))) '(a))
  (check-pred (λ (w) (not (is-in-a^nb^n? w))) '(b b))
  (check-pred (λ (w) (not (is-in-a^nb^n? w))) '(a b b))
  (check-pred (λ (w) (not (is-in-a^nb^n? w))) '(a b a a b b))
  (check-pred is-in-a^nb^n? '())
  (check-pred is-in-a^nb^n? '(a a b b))
  ```
- The acc is a stack
- The program first pushes all the as at the beginning of the given word onto the accumulator
- The auxiliary function pops an a for each recursive
- Suggests extending an ndfa with a stack to remember part of the consumed input

Pushdown Automata

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

  (make-ndpda K Σ Γ S F δ)

Pushdown Automata

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

  (make-ndpda K Σ Γ S F δ)

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

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

Pushdown Automata

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

  (make-ndpda K Σ Γ S F δ)

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

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

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

Pushdown Automata

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

  (make-ndpda K Σ Γ S F δ)

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

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

- ((A a p) (B g)) means that the machine is in state A, reads a from the input tape, pops p off the stack, pushes g onto the stack, and moves to B
- May be nondeterministic
- ((P EMP EMP) (Q j)) is a push operation that does not consult the input tape nor the stack
- ((P EMP j) (Q EMP)) is a pop operation

Pushdown Automata

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

Pushdown Automata

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

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

Pushdown Automata

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

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

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

Pushdown Automata

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

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

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

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

Pushdown Automata

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

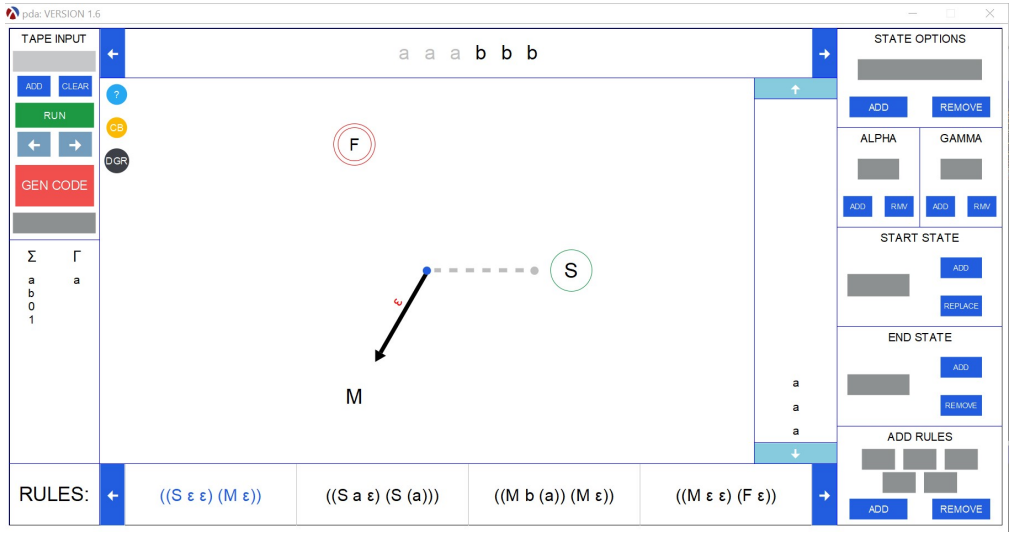
  $$(P, xw, a) \vdash (Q, w, g)$$

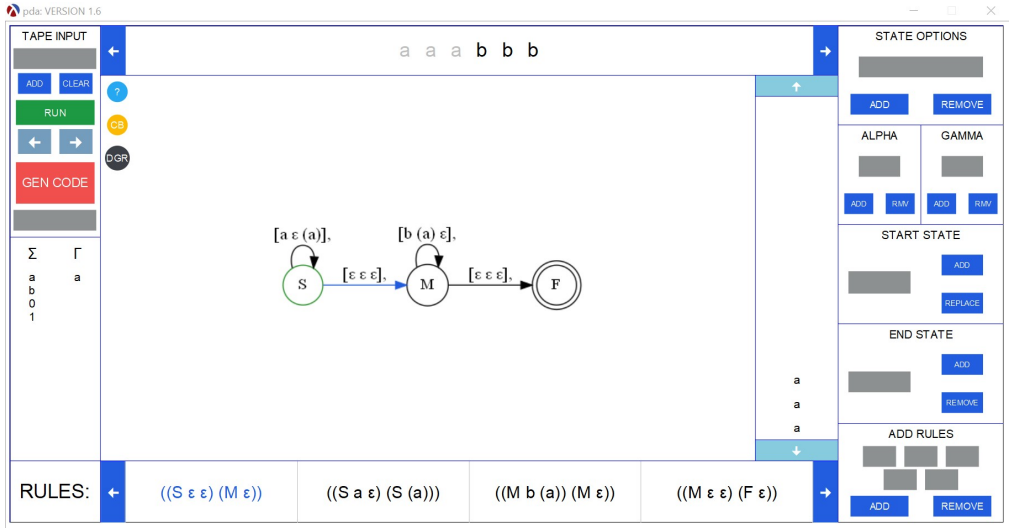
- Zero or more steps are denoted using $\vdash^*$
- A computation of length n on a word, w, is denoted by:

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

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

Pushdown Automata

- 

- 

Pushdown Automata

- A state invariant predicate may be associated with each state
- An invariant predicate has two inputs: the consumed input and the stack
- It must test and relate the invariant conditions for and between the consumed input and the stack.

Pushdown Automata

- Design and implement a pda for $L = a^n b^n$
- How may the stack be used?

Pushdown Automata

- Design and implement a pda for $L = a^n b^n$
- How may the stack be used?
- Accumulate the read as

Pushdown Automata

- Design and implement a pda for $L = a^n b^n$
- How may the stack be used?
- Accumulate the read as
- Match bs by popping an as

Pushdown Automata

- Design and implement a pda for $L = a^n b^n$
- How may the stack be used?
- Accumulate the read as
- Match bs by popping an as
- After all the input is read, the machine ought to move to a final state
- It accepts if the stack is empty. Otherwise, it rejects

Pushdown Automata

- Name: a^nb^n Σ = {a b}

Pushdown Automata

- Name: a^nb^n Σ = {a b}
- ```
  ;; Tests for a^nb^n
  #:rejects '((b b b) (a b a))
  #:accepts '(() (a b) (a a b b) (a a a b b b) (a a a a a b b b b b))
  ```
  ```

Pushdown Automata

- Name: a^nb^n Σ = {a b}
- ```
  ;; Tests for a^nb^n
  #:rejects '((b b b) (a b a))
  #:accepts '(() (a b) (a a b b) (a a a b b b) (a a a a a b b b b b))
  ```
- States

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

Pushdown Automata

- Name: a^nb^n Σ = {a b}
- ```
  ;; Tests for a^nb^n
  #:rejects '((b b b) (a b a))
  #:accepts '(() (a b) (a a b b) (a a a b b b) (a a a a a b b b b b))
  ```
- States

```
;; States
;;  S:   ci = a* = stack, start state
;;  M:   ci = (append (listof a) (listof b))
;;     ∧ stack = a*
;;     ∧ |ci as| = |stack| + |ci bs|
```
```

Pushdown Automata

- Name: a^nb^n Σ = {a b}
- ```
  ;; Tests for a^nb^n
  #:rejects '((b b b) (a b a))
  #:accepts '(() (a b) (a a b b) (a a a b b b) (a a a a a b b b b b))
  ```
- States

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

- ```
  ;;  M:   ci = (append (listof a) (listof b))
  ;;     ∧ stack = a*
  ;;     ∧ |ci as| = |stack| + |ci bs|
  ```
- ```
  ;;  F:    ci = (append (listof a) (listof b))
         ∧ |stack| = 0
  ;;     ∧ |ci as|=|ci bs|, final state
  ```
  ```

Pushdown Automata

- Transition Relation
- Push all a and nondeterministically move to M

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

Pushdown Automata

- Transition Relation
- Push all a and nondeterministically move to M

  `((S ,EMP ,EMP) (M ,EMP))` `((S a ,EMP) (S (a)))`
- Match all b and nondeterministically move to F

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

Pushdown Automata

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

Pushdown Automata

```
;; word stack → Boolean
;; Purpose: Determine if ci and stack are the same (listof a)
(define (S-INV ci stck)
 (and (= (length ci) (length stck))
      (andmap (λ (i g) (and (eq? i 'a) (eq? g 'a))) ci stck)))
;; Tests for S-INV
(check-equal? (S-INV '() '(a a)) #f) (check-equal? (S-INV '(a) '()) #f)
(check-equal? (S-INV '() '()) #t)
(check-equal? (S-INV '(a a a) '(a a a)) #t)
```

Pushdown Automata

- ```
  ;; word stack → Boolean
  ;; Purpose: Determine if ci and stack are the same (listof a)
  (define (S-INV ci stck)
   (and (= (length ci) (length stck))
        (andmap (λ (i g) (and (eq? i 'a) (eq? g 'a))) ci stck)))
  ;; Tests for S-INV
  (check-equal? (S-INV '() '(a a)) #f) (check-equal? (S-INV '(a) '()) #f)
  (check-equal? (S-INV '() '()) #t)
  (check-equal? (S-INV '(a a a) '(a a a)) #t)
  ```
- ```
  ;; word stack → Boolean
  ;; Purpose: Determine if ci = EMP or a+b+ AND the stack
  ;;          only contains a AND |ci as| = |stack| + |ci bs|
  (define (M-INV ci stck)
    (let* [(as (takef ci (λ (s) (eq? s 'a))))
           (bs (takef (drop ci (length as)) (λ (s) (eq? s 'b))))]
      (and (equal? (append as bs) ci)
           (andmap (λ (s) (eq? s 'a)) stck)
           (= (length as) (+ (length bs) (length stck))))))
  ;; Tests for M-INV
  (check-equal? (M-INV '(a a b) '(a a)) #f)
  (check-equal? (M-INV '(a a a b) '(a)) #f)
  (check-equal? (M-INV '() '()) #t)
  (check-equal? (M-INV '(a) '(a)) #t)
  (check-equal? (M-INV '(a a a b b) '(a)) #t)
  ```

Pushdown Automata

- ```
  ;; word stack → Boolean
  ;; Purpose: Determine if ci and stack are the same (listof a)
  (define (S-INV ci stck)
   (and (= (length ci) (length stck))
        (andmap (λ (i g) (and (eq? i 'a) (eq? g 'a))) ci stck)))
  ;; Tests for S-INV
  (check-equal? (S-INV '() '(a a)) #f) (check-equal? (S-INV '(a) '()) #f)
  (check-equal? (S-INV '() '()) #t)
  (check-equal? (S-INV '(a a a) '(a a a)) #t)
  ```
- ```
    ;; word stack → Boolean
    ;; Purpose: Determine if ci = EMP or a+b+ AND the stack
    ;;          only contains a AND |ci as| = |stack| + |ci bs|
    (define (M-INV ci stck)
      (let* [(as (takef ci (λ (s) (eq? s 'a))))
             (bs (takef (drop ci (length as)) (λ (s) (eq? s 'b))))]
        (and (equal? (append as bs) ci)
             (andmap (λ (s) (eq? s 'a)) stck)
             (= (length as) (+ (length bs) (length stck))))))
    ;; Tests for M-INV
    (check-equal? (M-INV '(a a b) '(a a)) #f)
    (check-equal? (M-INV '(a a a b) '(a)) #f)
    (check-equal? (M-INV '() '()) #t)
    (check-equal? (M-INV '(a) '(a)) #t)
    (check-equal? (M-INV '(a a a b b) '(a)) #t)
  ```
- ```
   ;; word stack → Boolean Purpose: Determine if ci=a^nb^n & empty stack
   (define (F-INV ci stck)
     (let* [(as (takef ci (λ (s) (eq? s 'a))))
            (bs (takef (drop ci (length as)) (λ (s) (eq? s 'b))))]
       (and (empty? stck) (equal? (append as bs) ci)  (= (length as) (length bs)))))
   ;; Tests for F-INV
   (check-equal? (F-INV '(a a b) '()) #f)
   (check-equal? (F-INV '(a a b b a b) '(a a a)) #f)
   (check-equal? (F-INV '() '()) #t)
   (check-equal? (F-INV '(a a b b) '()) #t)
  ```
  ```

Pushdown Automata

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

Theorem

The state invariants hold when P is applied to w.

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

Pushdown Automata

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

Pushdown Automata

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

Pushdown Automata

- Assume INVs hold for k steps. Show they hold for the k+1 step
- Proof invariants hold after each nonempty transition:
  `((S a EMP) (S (a)))`
  - By inductive hypothesis, `S-INV` holds
  - After consuming an a and pushing an a, P may reach S and by empty transition M
  - Note that an empty transition into F with a nonempty stack cannot lead to an accept. Therefore, we do not concern ourselves about P making such a transition
  - `S-INV` and `M-INV` hold because both the length of the consumed input and of the stack increased by 1, thus, remaining of equal length and because both continue to only contain as

Pushdown Automata

- Assume INVs hold for k steps. Show they hold for the k+1 step
- Proof invariants hold after each nonempty transition:
  `((S a EMP) (S (a)))`
  - By inductive hypothesis, `S-INV` holds
  - After consuming an `a` and pushing an `a`, P may reach S and by empty transition M
  - Note that an empty transition into F with a nonempty stack cannot lead to an accept. Therefore, we do not concern ourselves about P making such a transition
  - `S-INV` and `M-INV` hold because both the length of the consumed input and of the stack increased by 1, thus, remaining of equal length and because both continue to only contain as
- `((M b (a)) (M ,EMP))`:
  - By inductive hypothesis, `M-INV` holds
  - After consuming a `b` and popping an `a`, P may reach M or nondeterministically reach F because it may eventually accept
  - `M-INV` holds because ci continues to be as followed by bs, the stack can only contain as, and the number of as in ci remains equal to the sum of the number of bs in ci and the length of the stack.
  - `F-INV` holds because for a computation that ends with an accept the read `b` is the last symbol in the given word and popping an `a` makes the stack empty, ci continues to be the read as followed by the read bs, and, given that the stack is empty, the number of as equals the number of bs in the consumed input.

Pushdown Automata

- Proving `L = L(P)`

Pushdown Automata

- Proving `L = L(P)`

Lemma

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

Proof.

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

-

Pushdown Automata

- Proving `L = L(P)`

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

Proof.

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

Lemma
$w \notin L \Leftrightarrow w \notin L(P)$

Proof.

($\Rightarrow$) Assume w∉L

- $w \neq a^n b^n$
- 3 possibilities for the structure of w: not $a^n b^m$, w in $a^n b^m$, where $n > m$, or w in $a^n b^m$, where $n < m$
- In the first case, w either starts with a b or has at least one a after a b: cannot transition out of S or cannot transition out of M
- In the second case, never transition into F with an empty stack
- In the third case, can't read all the input

($\Leftarrow$) Assume w∉L(P). This means that P cannot transition into F with an empty stack having consumed w. Given that the state invariants always hold, w must either start with a b, have an a after a b, have too many as, or have too many bs. In all cases, w∉L. □

Pushdown Automata

- ```
  ;; L = wcw^R | w in (a b)*
  ```
- What is the design idea?
  ```

Pushdown Automata

- ```
  ;; L = wcw^R | w in (a b)*
  ```
- What is the design idea?
- Read `w` and push it on the stack
  ```

Pushdown Automata

- ```
  ;; L = wcw^R | w in (a b)*
  ```
- What is the design idea?
- Read `w` and push it on the stack
- Read `c`
  ```

Pushdown Automata

- ```
  ;; L = wcw^R | w in (a b)*
  ```
- What is the design idea?
- Read `w` and push it on the stack
- Read `c`
- Match elements of $\texttt{w}^{\texttt{R}}$ with the stack elements
- If all the elements after the `c` match all the elements on the stack then the machine accepts.
  ```

Pushdown Automata

- ```
  ;; L = wcw^R | w in (a b)*
  ```

- ```
  (define wcw^r (make-ndpda
  ```

Pushdown Automata

- ```
  ;; L = wcw^R | w in (a b)*
  ```

- ```
  (define wcw^r (make-ndpda
  ```

- ```
  ;; Tests for wcw^r
  (check-reject? wcw^r '(a) '(a c) '(b c a) '(a a b c b a b))
  (check-accept? wcw^r '(c) '(a c a) '(a b b b c b b b a))
  ```
  ```

Pushdown Automata

- ```
  ;; L = wcw^R | w in (a b)*
  ```
- ```
  ;; States
  ;;  S ci is empty and stack is empty
  ```
- ```
  (define wcw^r (make-ndpda
  ```
- ```
  ;; Tests for wcw^r
  (check-reject? wcw^r '(a) '(a c) '(b c a) '(a a b c b a b))
  (check-accept? wcw^r '(c) '(a c a) '(a b b b c b b b a))
  ```

Pushdown Automata

- ```
  ;; L = wcw^R | w in (a b)*
  ```
- ```
  ;; States
  ;;  S ci is empty and stack is empty
  ```
- ```
  ;;  P ci = stack^R AND c not in ci
  ```

- ```
  (define wcw^r (make-ndpda
  ```

- ```
  ;; Tests for wcw^r
  (check-reject? wcw^r '(a) '(a c) '(b c a) '(a a b c b a b))
  (check-accept? wcw^r '(c) '(a c a) '(a b b b c b b b a))
  ```
  ```

Pushdown Automata

- ```
  ;; L = wcw^R | w in (a b)*
  ```
- ```
  ;; States
  ;;  S ci is empty and stack is empty
  ```
- ```
  ;;  P ci = stack^R AND c not in ci
  ```
- ```
  ;;  Q ci = (append w (list 'c) v) AND
  ;;     w = stack^R v^R
  ```
- ```
  (define wcw^r (make-ndpda
  ```
- ```
  ;; Tests for wcw^r
  (check-reject? wcw^r '(a) '(a c) '(b c a) '(a a b c b a b))
  (check-accept? wcw^r '(c) '(a c a) '(a b b b c b b b a))
  ```

Pushdown Automata

- ```
  ;; L = wcw^R | w in (a b)*
  ```
- ```
  ;; States
  ;;  S ci is empty and stack is empty
  ```
- ```
  ;;  P ci = stack^R AND c not in ci
  ```
- ```
  ;;  Q ci = (append w (list 'c) v) AND
  ;;     w = stack^R v^R
  ```
- ```
  ;;  F stack = '() AND ci = (append w (list c) w^R)
  ```
- ```
  (define wcw^r (make-ndpda
  ```

- ```
  ;; Tests for wcw^r
  (check-reject? wcw^r '(a) '(a c) '(b c a) '(a a b c b a b))
  (check-accept? wcw^r '(c) '(a c a) '(a b b b c b b b a))
  ```
  ```

Pushdown Automata

- ```
  ;; L = wcw^R | w in (a b)*
  ```
- ```
  ;; States
  ;;  S ci is empty and stack is empty
  ```
- ```
  ;;  P ci = stack^R AND c not in ci
  ```
- ```
  ;;  Q ci = (append w (list 'c) v) AND
  ;;     w = stack^R v^R
  ```
- ```
  ;;  F stack = '() AND ci = (append w (list c) w^R)
  ```
- ```
  (define wcw^r (make-ndpda
  ```
- ```
                                `(((S ,EMP ,EMP) (P ,EMP))
  ```
- ```
  ;; Tests for wcw^r
  (check-reject? wcw^r '(a) '(a c) '(b c a) '(a a b c b a b))
  (check-accept? wcw^r '(c) '(a c a) '(a b b b c b b b a))
  ```

Pushdown Automata

```
;; L = wcwˆR | w in (a b)*
```

```
;; States
;;  S ci is empty and stack is empty
```

```
;;  P ci = stackˆR AND c not in ci
```

```
;;  Q ci = (append w (list 'c) v) AND
;;     w = stackˆR vˆR
```

```
;;  F stack = '() AND ci = (append w (list c) wˆR)
```

```
(define wcwˆr (make-ndpda
```

```
                                `(((S ,EMP ,EMP) (P ,EMP))
```

```
                                  ((P a ,EMP) (P (a)))
                                  ((P b ,EMP) (P (b)))
```

```
;; Tests for wcwˆr
(check-reject? wcwˆr '(a) '(a c) '(b c a) '(a a b c b a b))
(check-accept? wcwˆr '(c) '(a c a) '(a b b b c b b b a))
```

Pushdown Automata

```
;; L = wcw^R | w in (a b)*
```

```
;; States
;;  S ci is empty and stack is empty
```

```
;;  P ci = stack^R AND c not in ci
```

```
;;  Q ci = (append w (list 'c) v) AND
;;     w = stack^R v^R
```

```
;;  F stack = '() AND ci = (append w (list c) w^R)
```

```
(define wcw^r (make-ndpda
```

```
                              `(((S ,EMP ,EMP) (P ,EMP))
```

```
                                ((P a ,EMP) (P (a)))
                                ((P b ,EMP) (P (b)))
```

```
                                ((P c ,EMP) (Q ,EMP))
```

```
;; Tests for wcw^r
(check-reject? wcw^r '(a) '(a c) '(b c a) '(a a b c b a b))
(check-accept? wcw^r '(c) '(a c a) '(a b b b c b b b a))
```

Pushdown Automata

```
;; L = wcwˆR | w in (a b)*
```

```
;; States
;;  S ci is empty and stack is empty
```

```
;;  P ci = stackˆR AND c not in ci
```

```
;;  Q ci = (append w (list 'c) v) AND
;;     w = stackˆR vˆR
```

```
;;  F stack = '() AND ci = (append w (list c) wˆR)
```

```
(define wcwˆr (make-ndpda
```

```
                          `(((S ,EMP ,EMP) (P ,EMP))
```

```
                            ((P a ,EMP) (P (a)))
                            ((P b ,EMP) (P (b)))
```

```
                            ((P c ,EMP) (Q ,EMP))
```

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

```
;; Tests for wcwˆr
(check-reject? wcwˆr '(a) '(a c) '(b c a) '(a a b c b a b))
(check-accept? wcwˆr '(c) '(a c a) '(a b b b c b b b a))
```

Pushdown Automata

```
;; L = wcw^R | w in (a b)*
;; States
;;  S ci is empty and stack is empty
;;  P ci = stack^R AND c not in ci
;;  Q ci = (append w (list 'c) v) AND
;;     w = stack^R v^R
;;  F stack = '() AND ci = (append w (list c) w^R)
(define wcw^r (make-ndpda
                          '(S P Q F)
                          '(a b c)
                          '(a b)
                          'S
                          '(F)
                          `(((S ,EMP ,EMP) (P ,EMP))
                            ((P a ,EMP) (P (a)))
                            ((P b ,EMP) (P (b)))
                            ((P c ,EMP) (Q ,EMP))
                            ((Q a (a)) (Q ,EMP))
                            ((Q b (b)) (Q ,EMP))
                            ((Q ,EMP ,EMP) (F ,EMP)))))
;; Tests for wcw^r
(check-reject? wcw^r '(a) '(a c) '(b c a) '(a a b c b a b))
(check-accept? wcw^r '(c) '(a c a) '(a b b b c b b b a))
```

Pushdown Automata

- ```
  ;; word stack → Boolean
  ;; Purpose: Determine in the given word and stack are empty
  (define (S-INV ci s) (and (empty? ci) (empty? s)))

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

Pushdown Automata

- ```
  ;; word stack → Boolean
  ;; Purpose: Determine if the given ci is the reverse of
  ;;          the given stack AND c is not in ci
  (define (P-INV ci s)
    (and (equal? ci (reverse s)) (not (member 'c ci))))

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

Pushdown Automata

```
;; word stack → Boolean
;; Purpose: Determine if ci=s^Rv^Rcv
(define (Q-INV ci s)
  (let* [(w (takef ci (λ (s) (not (eq? s 'c)))))
         (v (if (member 'c ci)
                (drop ci (add1 (length w)))
                '()))]
    (and (equal? ci (append w (list 'c) v))
         (equal? w (append (reverse s) (reverse v))))))

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

Pushdown Automata

```
;; word stack → Boolean
;; Purpose: Determine if ci=s^Rv^Rcv AND stack is empty
(define (F-INV ci s)
  (let* [(w (takef ci (λ (s) (not (eq? s 'c)))))]
    (and (empty? s)
         (equal? ci (append w (list 'c) (reverse w))))))

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

Pushdown Automata

- L = wcw^R M = wcw^R w $\in$ (sm-sigma M)*
  s = the stack F = (sm-finals M) ci = consumed input

Pushdown Automata

- `L = wcw`^{R} `M = wcw^R` `w` $\in$ `(sm-sigma M)`*
  `s = the stack` `F = (sm-finals M)` `ci = consumed input`

Theorem

The state invariants hold when M accepts w.

-

Pushdown Automata

- Base Case

Pushdown Automata

- Base Case
- When `M` starts S-INV holds because ci = `'()` and s = `'()`

Pushdown Automata

- Base Case
- When `M` starts S-INV holds because ci = `'()` and s = `'()`
- A computation that leads to accept may also reach `P` by consuming no input
- ci = `'()` and s = `'()` when the machine moves to `P`
- `P-INV` holds because c is not a member of ci and ci = (reverse s)

Pushdown Automata

- Inductive Step
- `((P a ,EMP) (P (a)))`
  - By inductive hypothesis, `P-INV` holds: ci does not contain c and that ci = (reverse s)
  - Reading and pushing an a, ci still does not contain c and we still have that ci = (reverse s)
  - `P-INV` holds

Pushdown Automata

- Inductive Step
- `((P a ,EMP) (P (a)))`
  - By inductive hypothesis, `P-INV` holds: ci does not contain c and that ci = (reverse s)
  - Reading and pushing an a, ci still does not contain c and we still have that ci = (reverse s)
  - `P-INV` holds
- `((P b ,EMP) (P (b)))`:
  - By inductive hypothesis, `P-INV` holds: ci does not contain c and that ci = (reverse s)
  - By reading and pushing a b, ci still does not contain c and we still have that ci = (reverse s)
  - `P-INV` holds

Pushdown Automata

- Inductive Step
- `((P c ,EMP) (Q ,EMP))`:
  - By inductive hypothesis, `P-INV` holds: ci does not contain c and that ci = (reverse s)
  - By reading c, ci contains c
  - After the transition ci = (reverse s)c = (reverse s)(reverse `'()`)c(reverse `'()`) = (reverse s)(reverse v)cv
  - That is, v = `'()`. Thus, `Q-INV` holds
  - After using this transition rule, M may also reach F by consuming no input on a computation that leads to accept
  - Given that Q-INV holds and M only moves to F if v = s = `'()`, we have that ci = c = `'()c'()` = wc(reverse w). That is, w = EMP. Thus, F-INV holds

Pushdown Automata

- Inductive Step
- `((P c ,EMP) (Q ,EMP))`:
  - By inductive hypothesis, `P-INV` holds: ci does not contain c and that ci = (reverse s)
  - By reading c, ci contains c
  - After the transition ci = (reverse s)c = (reverse s)(reverse '())c(reverse '()) = (reverse s)(reverse v)cv
  - That is, v = '(). Thus, `Q-INV` holds
  - After using this transition rule, M may also reach F by consuming no input on a computation that leads to accept
  - Given that Q-INV holds and M only moves to F if v = s = '(), we have that ci = c = '()c'() = wc(reverse w). That is, w = EMP. Thus, F-INV holds
- `((Q a (a)) (Q ,EMP))`:
  - By inductive hypothesis, Q-INV holds: ci = (reverse s)(reverse v)cv
  - Reading and popping a moves the a on the top of the stack to v.
  - Thus, ci = (reverse s)(reverse v)cv after using this transition and Q-INV holds.
  - If a is the last element of the input and the stack is empty then M moves to F. This means that ci = (reverse s)(reverse v)cv = EMP(reverse v)cv = (reverse v)cv = (reverse w)cw. That is, v = w. Therefore, F-INV holds

Pushdown Automata

- Inductive Step
- `((P c ,EMP) (Q ,EMP))`:
  - By inductive hypothesis, `P-INV` holds: ci does not contain c and that ci = (reverse s)
  - By reading c, ci contains c
  - After the transition ci = (reverse s)c = (reverse s)(reverse '())c(reverse '()) = (reverse s)(reverse v)cv
  - That is, v = '(). Thus, `Q-INV` holds
  - After using this transition rule, M may also reach F by consuming no input on a computation that leads to accept
  - Given that Q-INV holds and M only moves to F if v = s = '(), we have that ci = c = '()c'() = wc(reverse w). That is, w = EMP. Thus, F-INV holds
- `((Q a (a)) (Q ,EMP))`:
  - By inductive hypothesis, Q-INV holds: ci = (reverse s)(reverse v)cv
  - Reading and popping a moves the a on the top of the stack to v.
  - Thus, ci = (reverse s)(reverse v)cv after using this transition and Q-INV holds.
  - If a is the last element of the input and the stack is empty then M moves to F. This means that ci = (reverse s)(reverse v)cv = EMP(reverse v)cv = (reverse v)cv = (reverse w)cw. That is, v = w. Therefore, F-INV holds
- `((Q b (b)) (Q ,EMP))`: idem, but with b.

Pushdown Automata

- `L = L(M)`

Pushdown Automata

- `L = L(M)`

Lemma

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

-

Pushdown Automata

- `L = L(M)`

Lemma

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

-
- (⇒) Assume w∈L
  - w = vcv^R
  - Given that state invariants always hold, there is a computation that has M consume w and reach F with an empty stack
  - w∈L(M)

Pushdown Automata

- `L = L(M)`

Lemma

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

-
- (⇒) Assume w∈L
  - w = vcv^R
  - Given that state invariants always hold, there is a computation that has M consume w and reach F with an empty stack
  - w∈L(M)
- (⇐) Assume w∈L(M)
  - M halts in F with an empty stack having consumed w
  - Given that the state invariants always hold: w = vcv^R
  - w∈L

Pushdown Automata

Lemma

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

-

Pushdown Automata

Lemma

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

-
- ($\Rightarrow$) Assume w$\notin$L
  - $w \neq v^R cv$
  - Given that the state invariant predicates always hold, there is no computation that has M consume w and end in F with an empty stack
  - w$\notin$L(M)

Pushdown Automata

Lemma

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

-
- ($\Rightarrow$) Assume w$\notin$L
  - $w \neq v^R cv$
  - Given that the state invariant predicates always hold, there is no computation that has M consume w and end in F with an empty stack
  - w$\notin$L(M)
- ($\Leftarrow$) Assume w$\notin$L(M)
  - M cannot transition into F with an empty stack having consumed w
  - Given that the state invariants always hold: $w \neq vcv^R$
  - w$\notin$L

Pushdown Automata

- HOMEWORK: 1–5, 8
- Quiz:

Pushdown Automata

- For every language, `L`, decided by an `ndfa`, is there is a `pda` that decides `L`?

Pushdown Automata

- For every language, `L`, decided by an `ndfa`, is there is a `pda` that decides `L`?
- If a language, `L`, is regular then there is a `ndfa` that decides `L`
- Intuitively, an `ndfa` may be thought of as a `pda` that never operates on its stack
- We ought to be able to build a `pda` from an `ndfa`

Pushdown Automata

- Design Idea
- Given an `ndfa`, `M`, a `pda` is constructed using:
- `M`'s states, alphabet, starting state, and final states

Pushdown Automata

- Design Idea
- Given an `ndfa`, `M`, a `pda` is constructed using:
- `M`'s states, alphabet, starting state, and final states
- The stack alphabet for the constructed `pda` may be `'()` because nothing shall every be pushed onto the stack

Pushdown Automata

- Design Idea
- Given an `ndfa`, `M`, a `pda` is constructed using:
- `M`'s states, alphabet, starting state, and final states
- The stack alphabet for the constructed `pda` may be `'()` because nothing shall every be pushed onto the stack
- The `pda`'s transition relation is built using `M`'s transition relation:

Pushdown Automata

- Design Idea
- Given an `ndfa`, `M`, a `pda` is constructed using:
- `M`'s states, alphabet, starting state, and final states
- The stack alphabet for the constructed `pda` may be `'()` because nothing shall every be pushed onto the stack
- The `pda`'s transition relation is built using `M`'s transition relation:
- Each of `M`'s rules is converted to a `pda`-rule

Pushdown Automata

- Design Idea
- Given an `ndfa`, `M`, a `pda` is constructed using:
- `M`'s states, alphabet, starting state, and final states
- The stack alphabet for the constructed `pda` may be `'()` because nothing shall every be pushed onto the stack
- The `pda`'s transition relation is built using `M`'s transition relation:
- Each of `M`'s rules is converted to a `pda`-rule
- For an `ndfa`-rule, `(P a R)`, the `pda`-rule `((P a EMP) (R EMP))` is generated
- The `pda` moves from `P` to `R` without modifying the stack

Pushdown Automata

- ```
  ;; ndfa → pda     Purpose: Convert the given ndfa to a pda
  (define (ndfa->pda M #:accepts [ins '()] #:rejects [notins '()])
  ```
  ```

Pushdown Automata

- ```
  ;; ndfa → pda     Purpose: Convert the given ndfa to a pda
  (define (ndfa->pda M #:accepts [ins '()] #:rejects [notins '()])
  ```

- ```
  ;; Sample pda
  ;; L = w | a not in w OR b not in w OR c not in w
  (define ALOM-PDA
    (ndfa->pda AT-LEAST-ONE-MISSING
               #:rejects '((a b c) (b b a c c) (c b b c a))
               #:accepts '(() (a) (b) (c) (a b) (b c) (c a) (b a b b a a))))
  ))
  ;; L = ϵ U aa* U ab*
  (define LNDFA-PDA (ndfa->pda LNDFA
                               #:rejects '((b) (b a b b) (a b b b a b))
                               #:accepts '(() (a) (a a a) (a b b b b))))

  (check-equal? (sm-testequiv? ALOM-PDA AT-LEAST-ONE-MISSING) #t)
  (check-equal? (sm-testequiv? LNDFA-PDA LNDFA) #t)
  ```

Pushdown Automata

- ```
  ;; ndfa → pda     Purpose: Convert the given ndfa to a pda
  (define (ndfa->pda M #:accepts [ins '()] #:rejects [notins '()])
  ```
- ```
    (let [(states (sm-states M))
          (sigma  (sm-sigma M))
          (start  (sm-start M))
          (finals (sm-finals M))
          (rules  (sm-rules M))]
      (make-ndpda states
                  sigma
                  '()
                  start
                  finals
  ```

- ```
  ;; Sample pda
  ;; L = w | a not in w OR b not in w OR c not in w
  (define ALOM-PDA
    (ndfa->pda AT-LEAST-ONE-MISSING
               #:rejects '((a b c) (b b a c c) (c b b c a))
               #:accepts '(() (a) (b) (c) (a b) (b c) (c a) (b a b b a a))))
  ))
  ;; L = ϵ U aa* U ab*
  (define LNDFA-PDA (ndfa->pda LNDFA
                               #:rejects '((b) (b a b b) (a b b b a b))
                               #:accepts '(() (a) (a a a) (a b b b b))))

  (check-equal? (sm-testequiv? ALOM-PDA AT-LEAST-ONE-MISSING) #t)
  (check-equal? (sm-testequiv? LNDFA-PDA LNDFA) #t)
  ```
  ```

Pushdown Automata

```
;; ndfa → pda     Purpose: Convert the given ndfa to a pda
(define (ndfa->pda M #:accepts [ins '()] #:rejects [notins '()])
  (let [(states (sm-states M))
        (sigma  (sm-sigma M))
        (start  (sm-start M))
        (finals (sm-finals M))
        (rules  (sm-rules M))]
    (make-ndpda states
                sigma
                '()
                start
                finals
                (map (λ (r) (list (list (first r) (second r) EMP)
                                  (list (third r) EMP)))
                     rules)
                #:rejects notins
                #:accepts ins)))
```

```
;; Sample pda
;; L = w | a not in w OR b not in w OR c not in w
(define ALOM-PDA
  (ndfa->pda AT-LEAST-ONE-MISSING
             #:rejects '((a b c) (b b a c c) (c b b c a))
             #:accepts '(() (a) (b) (c) (a b) (b c) (c a) (b a b b a a))))
))
;; L = ϵ U aa* U ab*
(define LNDFA-PDA (ndfa->pda LNDFA
                             #:rejects '((b) (b a b b) (a b b b a b))
                             #:accepts '(() (a) (a a a) (a b b b b))))

(check-equal? (sm-testequiv? ALOM-PDA AT-LEAST-ONE-MISSING) #t)
(check-equal? (sm-testequiv? LNDFA-PDA LNDFA) #t)
```

Pushdown Automata

- HOMEWORK: 11

Equivalence of pdas and cfgs

Theorem

The set of languages accepted by pdas is exactly the context-free languages.

-

Equivalence of pdas and cfgs

Theorem

The set of languages accepted by pdas is exactly the context-free languages.

-
- We must show:
  1. Given a cfg, G, show how to build a pda for L(G)
  2. Given a pda, P, show how to build a cfg for L(P)

Equivalence of pdas and cfgs

Lemma

L is a context-free language $\Rightarrow \exists$ *pda that accepts L*

- Design Idea

Equivalence of pdas and cfgs

Lemma

L is a context-free language $\Rightarrow \exists$ *pda that accepts L*

- Design Idea
- `P = (make-ndpda K Σ S F Δ)`, such that `L(P) = L(G)`

Equivalence of pdas and cfgs

Lemma

L is a context-free language $\Rightarrow \exists$ *pda that accepts L*

- Design Idea
- `P = (make-ndpda K Σ S F Δ)`, such that `L(P) = L(G)`
- Read the input from left to right: simulate the leftmost derivation of `w`

Equivalence of pdas and cfgs

Lemma

L is a context-free language $\Rightarrow \exists$ *pda that accepts L*

- Design Idea
- `P = (make-ndpda K Σ S F Δ)`, such that `L(P) = L(G)`
- Read the input from left to right: simulate the leftmost derivation of `w`
- To start, `P` pushes S_G onto the stack
- For each `cfg`-rule, `A` $\rightarrow$`x`, `P` pops `A` off the stack and pushes `x` onto the stack
- For each `a`$\in\Sigma_G$, `P` reads `a` from the tape and pops `a` off the stack

Equivalence of pdas and cfgs

Lemma

L is a context-free language $\Rightarrow \exists$ *pda that accepts L*

- Design Idea
- `P = (make-ndpda K Σ S F Δ)`, such that `L(P) = L(G)`
- Read the input from left to right: simulate the leftmost derivation of `w`
- To start, `P` pushes S_G onto the stack
- For each `cfg`-rule, `A` $\rightarrow$`x`, `P` pops `A` off the stack and pushes `x` onto the stack
- For each `a`$\in\Sigma_G$, `P` reads `a` from the tape and pops `a` off the stack
- Suggests that `P` only needs two states:
  - Starting state, `S`, in which `P` pushes S_G and moves to, `Q`
  - `P` remains in `Q`, which is the final state, for the rest of the computation

Equivalence of pdas and cfgs

- P is constructed as follows:

  `(make-ndpda '(S Q) Σ_G V_G∪Σ_G 'S '(Q) Δ)`

Equivalence of pdas and cfgs

- P is constructed as follows:

  (make-ndpda '(S Q) Σ_G $V_G \cup \Sigma_G$ 'S '(Q) Δ)

- There are three types of rule in Δ:

  1. ((S EMP (list S_G)) (Q EMP))
  2. ((Q EMP (list 'A)) (Q (symbol->fsmlos x))), for (A $\rightarrow$ x)$\in R_G$
  3. ((Q a (a)) (Q EMP)), for a$\in \Sigma_G$

Equivalence of pdas and cfgs

- P is constructed as follows:

  (make-ndpda '(S Q) Σ_G $V_G \cup \Sigma_G$ 'S '(Q) Δ)

- There are three types of rule in Δ:

  1. ((S EMP (list S_G)) (Q EMP))
  2. ((Q EMP (list 'A)) (Q (symbol->fsmlos x))), for (A $\rightarrow$ x)$\in R_G$
  3. ((Q a (a)) (Q EMP)), for a$\in \Sigma_G$

- We are careful to say nothing about the behavior of P if it is ever applied to a word not in L(G)

Equivalence of pdas and cfgs

- ```
  ;; cfg → pda     Purpose: Transform the given cfg into a pda
  (define (cfg2pda G #:rejects [rejs '()] #:accepts [accs '()])
  ```
  ```

Equivalence of pdas and cfgs

- ```
  ;; cfg → pda     Purpose: Transform the given cfg into a pda
  (define (cfg2pda G #:rejects [rejs '()] #:accepts [accs '()])
  ```

- ```
  ;; Tests
  (define a2nb2n-pda (cfg2pda a2nb2n
                              #:rejects '((b b) (a a b b a b))
                              #:accepts '(() (a b) (a a b b) (a a a b b b))))

  (define numb>numa-pda (cfg2pda numb>numa   rejects may cause an infinite derivation
                                 #:accepts '((b b b) (b b a) (a b b))))
  ```

Equivalence of pdas and cfgs

- ```
  ;; cfg → pda     Purpose: Transform the given cfg into a pda
  (define (cfg2pda G #:rejects [rejs '()] #:accepts [accs '()])
  ```
- ```
   (let [(nts (grammar-nts G))      (sigma (grammar-sigma G))
         (start (grammar-start G))  (rules (grammar-rules G))]
    (make-ndpda '(S Q)
                sigma
                (append nts sigma)
                'S
                (list 'Q)
  ```

- ```
  ;; Tests
  (define a2nb2n-pda (cfg2pda a2nb2n
                              #:rejects '((b b) (a a b b a b))
                              #:accepts '(() (a b) (a a b b) (a a a b b b))))

  (define numb>numa-pda (cfg2pda numb>numa   rejects may cause an infinite derivation
                                 #:accepts '((b b b) (b b a) (a b b))))
  ```
  ```

Equivalence of pdas and cfgs

- ```
  ;; cfg → pda     Purpose: Transform the given cfg into a pda
  (define (cfg2pda G #:rejects [rejs '()] #:accepts [accs '()])
  ```
- ```
   (let [(nts (grammar-nts G))      (sigma (grammar-sigma G))
         (start (grammar-start G))  (rules (grammar-rules G))]
    (make-ndpda '(S Q)
                sigma
                (append nts sigma)
                'S
                (list 'Q)
  ```
- ```
                (append
                 (list (list (list 'S EMP EMP) (list 'Q (list start))))
                 (map (λ (r)
                        (list (list 'Q EMP (list (first r)))
                              (list 'Q
                                    (if (eq? (third r) EMP)
                                        EMP
                                        (symbol->fsmlos (third r))))))
                      rules)
                 (map (λ (a) (list (list 'Q a (list a)) (list 'Q EMP)))
                      sigma)
                 #:rejects rejs
                 #:accepts accs))))
  ```
- ```
  ;; Tests
  (define a2nb2n-pda (cfg2pda a2nb2n
                              #:rejects '((b b) (a a b b a b))
                              #:accepts '(() (a b) (a a b b) (a a a b b b))))

  (define numb>numa-pda (cfg2pda numb>numa   rejects may cause an infinite derivation
                                 #:accepts '((b b b) (b b a) (a b b))))
  ```

Equivalence of pdas and cfgs

- Sometimes a constructed pda is capable of rejecting and other times it is incapable of rejecting

Equivalence of pdas and cfgs

- Sometimes a constructed pda is capable of rejecting and other times it is incapable of rejecting
- A pda produced by the constructor is guaranteed to accept if the given word is in its language but no such guarantee is made for rejection

Equivalence of pdas and cfgs

- Sometimes a constructed pda is capable of rejecting and other times it is incapable of rejecting
- A pda produced by the constructor is guaranteed to accept if the given word is in its language but no such guarantee is made for rejection
- When a machine can guarantee accepting any word in a given language, L, but cannot guarantee rejecting any word not in L, we say that the machine *semidecides* L
- The constructor produces pdas that semidecide the language of the given cfg.

Equivalence of pdas and cfgs

- For an intuitive understanding of why a pda may only semidecide a language, it is necessary to understand how nondeterminism is implemented

Equivalence of pdas and cfgs

- For an intuitive understanding of why a pda may only semidecide a language, it is necessary to understand how nondeterminism is implemented
- Breadth-first search for a computation to accept the given word
- The search space is a tree of derivations defined by the machine's rules

Equivalence of pdas and cfgs

- For an intuitive understanding of why a pda may only semidecide a language, it is necessary to understand how nondeterminism is implemented
- Breadth-first search for a computation to accept the given word
- The search space is a tree of derivations defined by the machine's rules
- If a computation ends at level k then breadth-first search is guaranteed to reach it. If the computation leads to accept then the machine accepts
- Otherwise, the breath-first process continues to extend the search tree to find a computation that leads to accept

Equivalence of pdas and cfgs

- For an intuitive understanding of why a pda may only semidecide a language, it is necessary to understand how nondeterminism is implemented
- Breadth-first search for a computation to accept the given word
- The search space is a tree of derivations defined by the machine's rules
- If a computation ends at level k then breadth-first search is guaranteed to reach it. If the computation leads to accept then the machine accepts
- Otherwise, the breath-first process continues to extend the search tree to find a computation that leads to accept
- If all paths in the search tree are finite and reject then the machine rejects
- If there is a path in the search tree that is not finite BFS may continue to search for a computation that leads to an accept forever
- Thus, the machine can accept but can never reject

Equivalence of pdas and cfgs

- To understand why a2nb2n-pda decides its language:

```
'(((P ϵ ϵ)   (Q (S)))
  ((Q ϵ (S)) (Q ϵ))
  ((Q ϵ (S)) (Q (a S b)))
  ((Q a (a)) (Q ϵ))
  ((Q b (b)) (Q ϵ)))
```

- After popping S, the machine must read input before popping another nonterminal or must only read input (when there are only bs in the stack)

Equivalence of pdas and cfgs

- To understand why a2nb2n-pda decides its language:

```
'(((P ϵ ϵ)   (Q (S)))
  ((Q ϵ (S)) (Q ϵ))
  ((Q ϵ (S)) (Q (a S b)))
  ((Q a (a)) (Q ϵ))
  ((Q b (b)) (Q ϵ)))
```

- After popping S, the machine must read input before popping another nonterminal or must only read input (when there are only bs in the stack)
- The input becomes empty and the stack is examined to decide if the word is rejected or accepted
- The machine cannot consume the whole input which means the computation ends in reject
- The machine can always say accept or reject, because all the paths in the search space are finite
- We say the machine *decides* its language.

Equivalence of pdas and cfgs

- To understand why a2nb2n-pda decides its language:

```
'(((P ϵ ϵ)   (Q (S)))
  ((Q ϵ (S)) (Q ϵ))
  ((Q ϵ (S)) (Q (a S b)))
  ((Q a (a)) (Q ϵ))
  ((Q b (b)) (Q ϵ)))
```

- After popping S, the machine must read input before popping another nonterminal or must only read input (when there are only bs in the stack)
- The input becomes empty and the stack is examined to decide if the word is rejected or accepted
- The machine cannot consume the whole input which means the computation ends in reject
- The machine can always say accept or reject, because all the paths in the search space are finite
- We say the machine *decides* its language.
- In contrast, the transition relation for numb>numa is:

```
'(((P ϵ ϵ) (Q (S)))
  ((Q ϵ (S)) (Q (b)))
  ((Q ϵ (S)) (Q (A b A)))
  ((Q ϵ (A)) (Q (A a A b A)))
  ((Q ϵ (A)) (Q (A b A a A)))
  ((Q ϵ (A)) (Q ϵ))
  ((Q ϵ (A)) (Q (b A)))
  ((Q a (a)) (Q ϵ))
  ((Q b (b)) (Q ϵ)))
```

- There are computations for which the stack always grows using the fourth or fifth rules without consuming any input
- This means that there are computations that are infinite
- A pda can only reject if all possible computations reject: impossible for computations that are infinite

Equivalence of pdas and cfgs

- To understand why a2nb2n-pda decides its language:

```
'(((P ϵ ϵ)   (Q (S)))
  ((Q ϵ (S)) (Q ϵ))
  ((Q ϵ (S)) (Q (a S b)))
  ((Q a (a)) (Q ϵ))
  ((Q b (b)) (Q ϵ)))
```

- After popping S, the machine must read input before popping another nonterminal or must only read input (when there are only bs in the stack)
- The input becomes empty and the stack is examined to decide if the word is rejected or accepted
- The machine cannot consume the whole input which means the computation ends in reject
- The machine can always say accept or reject, because all the paths in the search space are finite
- We say the machine *decides* its language.
- In contrast, the transition relation for numb>numa is:

```
'(((P ϵ ϵ) (Q (S)))
  ((Q ϵ (S)) (Q (b)))
  ((Q ϵ (S)) (Q (A b A)))
  ((Q ϵ (A)) (Q (A a A b A)))
  ((Q ϵ (A)) (Q (A b A a A)))
  ((Q ϵ (A)) (Q ϵ))
  ((Q ϵ (A)) (Q (b A)))
  ((Q a (a)) (Q ϵ))
  ((Q b (b)) (Q ϵ)))
```

- There are computations for which the stack always grows using the fourth or fifth rules without consuming any input
- This means that there are computations that are infinite
- A pda can only reject if all possible computations reject: impossible for computations that are infinite
- Accept if in Q with an empty stack and no more input to read, but it can never determine that all possible computations reject
- We say the machine *semidecides* its language.

Equivalence of pdas and cfgs

Let:

```
G = (make-cfg N Σ R S)      P = (cfg2pda G)
```

Lemma

$S \ {}^{L}\!\rightarrow_{G} \omega\alpha \Leftrightarrow (Q\ \omega\ S) \vdash^{*}_{M} (Q\ \epsilon\ \alpha)$, *where* $\alpha \in \{N(N \cup \Sigma)^{*} \cup \{\epsilon\}\} \wedge \omega \in \Sigma^{*}$

- A leftmost derivation of $\omega\alpha$ is either empty or starts with the leftmost nonterminal in $\omega\alpha$ is logically equivalent to `M` starting in `Q`, reading ω, and popping `S` to end in `Q` with α on the stack
- Observe that if the stack is not empty after consuming ω then the topmost element is the next nonterminal to substitute in the leftmost derivation

Equivalence of pdas and cfgs

Let:

```
G = (make-cfg N Σ R S)      P = (cfg2pda G)
```

Lemma

$S \overset{L}{\rightarrow}_G \omega\alpha \Leftrightarrow (Q\ \omega\ S) \vdash^*_M (Q\ \epsilon\ \alpha)$, *where* $\alpha \in \{N(N \cup \Sigma)^* \cup \{\epsilon\}\} \wedge \omega \in \Sigma^*$

- ($\Rightarrow$) The proof is by induction on $\mathtt{n}$ = the length of the leftmost derivation of ω.

Equivalence of pdas and cfgs

Let:

```
G = (make-cfg N Σ R S)      P = (cfg2pda G)
```

Lemma

$S \overset{L}{\rightarrow}_G \omega\alpha \Leftrightarrow (Q\ \omega\ S) \vdash^*_M (Q\ \epsilon\ \alpha)$, *where* $\alpha \in \{N(N \cup \Sigma)^* \cup \{\epsilon\}\} \wedge \omega \in \Sigma^*$

- ($\Rightarrow$) The proof is by induction on $\mathtt{n}$ = the length of the leftmost derivation of ω.
- Base case, n = 0
- This means that $\omega = \epsilon$ and $\alpha =$ S
- We have that (Q ω S) = (Q ϵ S) $\vdash^*_M$ (Q ϵ S) = (Q ϵ α).

Equivalence of pdas and cfgs

Let:

```
G = (make-cfg N Σ R S)      P = (cfg2pda G)
```

Lemma

$S \overset{L}{\rightarrow}_G \omega\alpha \Leftrightarrow (Q\ \omega\ S) \vdash^*_M (Q\ \epsilon\ \alpha)$, *where* $\alpha \in \{N(N \cup \Sigma)^* \cup \{\epsilon\}\} \wedge \omega \in \Sigma^*$

- ($\Rightarrow$) The proof is by induction on n = the length of the leftmost derivation of ω.
- Base case, n = 0
- This means that $\omega = \epsilon$ and $\alpha = S$
- We have that $(Q\ \omega\ S) = (Q\ \epsilon\ S) \vdash^*_M (Q\ \epsilon\ S) = (Q\ \epsilon\ \alpha)$.
- For the inductive step:
- Assume: $S \overset{L}{\rightarrow}_G \omega\alpha \Leftrightarrow (Q\ \omega\ S) \vdash^*_M (Q\ \epsilon\ \alpha)$, for n = k.
- Show: $S \overset{L}{\rightarrow}_G \omega\alpha \Leftrightarrow (Q\ \omega\ S) \vdash^*_M (Q\ \epsilon\ \alpha)$, for n = k + 1.

Equivalence of pdas and cfgs

Let:

```
G = (make-cfg N Σ R S)      P = (cfg2pda G)
```

Lemma

$S \overset{L}{\rightarrow}_G \omega\alpha \Leftrightarrow (Q\ \omega\ S) \vdash^*_M (Q\ \epsilon\ \alpha)$, *where* $\alpha \in \{N(N \cup \Sigma)^* \cup \{\epsilon\}\} \wedge \omega \in \Sigma^*$

- ($\Rightarrow$) The proof is by induction on n = the length of the leftmost derivation of ω.
- Base case, n = 0
- This means that $\omega = \epsilon$ and $\alpha = S$
- We have that $(Q\ \omega\ S) = (Q\ \epsilon\ S) \vdash^*_M (Q\ \epsilon\ S) = (Q\ \epsilon\ \alpha)$.
- For the inductive step:
- Assume: $S \overset{L}{\rightarrow}_G \omega\alpha \Leftrightarrow (Q\ \omega\ S) \vdash^*_M (Q\ \epsilon\ \alpha)$, for n = k.
- Show: $S \overset{L}{\rightarrow}_G \omega\alpha \Leftrightarrow (Q\ \omega\ S) \vdash^*_M (Q\ \epsilon\ \alpha)$, for n = k + 1.
- The derivation of length k+1 looks as follows:
- $S \rightarrow u_1 \rightarrow u_2 \rightarrow \ldots \rightarrow u_n \rightarrow u_{n+1} = \omega\alpha$

Equivalence of pdas and cfgs

Let:

```
G = (make-cfg N Σ R S)      P = (cfg2pda G)
```

Lemma

$S \overset{L}{\rightarrow}_G \omega\alpha \Leftrightarrow (Q\ \omega\ S) \vdash^*_M (Q\ \epsilon\ \alpha)$, *where* $\alpha \in \{N(N \cup \Sigma)^* \cup \{\epsilon\}\} \wedge \omega \in \Sigma^*$

- ($\Rightarrow$) The proof is by induction on n = the length of the leftmost derivation of ω.
- Base case, n = 0
- This means that $\omega = \epsilon$ and $\alpha = S$
- We have that $(Q\ \omega\ S) = (Q\ \epsilon\ S) \vdash^*_M (Q\ \epsilon\ S) = (Q\ \epsilon\ \alpha)$.
- For the inductive step:
- Assume: $S \overset{L}{\rightarrow}_G \omega\alpha \Leftrightarrow (Q\ \omega\ S) \vdash^*_M (Q\ \epsilon\ \alpha)$, for n = k.
- Show: $S \overset{L}{\rightarrow}_G \omega\alpha \Leftrightarrow (Q\ \omega\ S) \vdash^*_M (Q\ \epsilon\ \alpha)$, for n = k + 1.
- The derivation of length k+1 looks as follows:
- $S \rightarrow u_1 \rightarrow u_2 \rightarrow \ldots \rightarrow u_n \rightarrow u_{n+1} = \omega\alpha$
- Observe that $u_n = xA\beta$, where $x \in \Sigma^* \wedge A \in N \wedge \beta \in \{N \cup \Sigma\}^*$

Equivalence of pdas and cfgs

Let:

```
G = (make-cfg N Σ R S)      P = (cfg2pda G)
```

Lemma

$S \overset{L}{\rightarrow}_G \omega\alpha \Leftrightarrow (Q\ \omega\ S) \vdash^*_M (Q\ \epsilon\ \alpha)$, *where* $\alpha \in \{N(N \cup \Sigma)^* \cup \{\epsilon\}\} \wedge \omega \in \Sigma^*$

- ($\Rightarrow$) The proof is by induction on n = the length of the leftmost derivation of ω.
- Base case, n = 0
- This means that $\omega = \epsilon$ and $\alpha = S$
- We have that $(Q\ \omega\ S) = (Q\ \epsilon\ S) \vdash^*_M (Q\ \epsilon\ S) = (Q\ \epsilon\ \alpha)$.
- For the inductive step:
- Assume: $S \overset{L}{\rightarrow}_G \omega\alpha \Leftrightarrow (Q\ \omega\ S) \vdash^*_M (Q\ \epsilon\ \alpha)$, for n = k.
- Show: $S \overset{L}{\rightarrow}_G \omega\alpha \Leftrightarrow (Q\ \omega\ S) \vdash^*_M (Q\ \epsilon\ \alpha)$, for n = k + 1.
- The derivation of length k+1 looks as follows:
- $S \rightarrow u_1 \rightarrow u_2 \rightarrow \ldots \rightarrow u_n \rightarrow u_{n+1} = \omega\alpha$
- Observe that $u_n = xA\beta$, where $x \in \Sigma^* \wedge A \in N \wedge \beta \in \{N \cup \Sigma\}^*$
- The derivation step from u_n to u_{n+1} substitutes A using a rule: $A \rightarrow \theta$, where $\theta \in \{N \cup \Sigma\}^*$
- $u_{n+1} = x\theta\beta$.

Equivalence of pdas and cfgs

Let:

```
G = (make-cfg N Σ R S)      P = (cfg2pda G)
```

Lemma

$S \overset{L}{\rightarrow}_G \omega\alpha \Leftrightarrow (Q\ \omega\ S) \vdash^*_M (Q\ \epsilon\ \alpha)$, *where* $\alpha \in \{N(N \cup \Sigma)^* \cup \{\epsilon\}\} \wedge \omega \in \Sigma^*$

- ($\Rightarrow$) The proof is by induction on n = the length of the leftmost derivation of ω.
- Base case, n = 0
- This means that $\omega = \epsilon$ and $\alpha = S$
- We have that $(Q\ \omega\ S) = (Q\ \epsilon\ S) \vdash^*_M (Q\ \epsilon\ S) = (Q\ \epsilon\ \alpha)$.
- For the inductive step:
- Assume: $S \overset{L}{\rightarrow}_G \omega\alpha \Leftrightarrow (Q\ \omega\ S) \vdash^*_M (Q\ \epsilon\ \alpha)$, for n = k.
- Show: $S \overset{L}{\rightarrow}_G \omega\alpha \Leftrightarrow (Q\ \omega\ S) \vdash^*_M (Q\ \epsilon\ \alpha)$, for n = k + 1.
- The derivation of length k+1 looks as follows:
- $S \rightarrow u_1 \rightarrow u_2 \rightarrow \ldots \rightarrow u_n \rightarrow u_{n+1} = \omega\alpha$
- Observe that $u_n = xA\beta$, where $x \in \Sigma^* \wedge A \in N \wedge \beta \in \{N \cup \Sigma\}^*$
- The derivation step from u_n to u_{n+1} substitutes A using a rule: $A \rightarrow \theta$, where $\theta \in \{N \cup \Sigma\}^*$
- $u_{n+1} = x\theta\beta$.
- By inductive hypothesis, we have $(Q\ x\ S) \vdash^* (Q\ \epsilon\ A\beta)$

Equivalence of pdas and cfgs

Let:

```
G = (make-cfg N Σ R S)      P = (cfg2pda G)
```

Lemma

$S \overset{L}{\rightarrow}_G \omega\alpha \Leftrightarrow (Q\ \omega\ S) \vdash^*_M (Q\ \epsilon\ \alpha)$, *where* $\alpha \in \{N(N \cup \Sigma)^* \cup \{\epsilon\}\} \wedge \omega \in \Sigma^*$

- ($\Rightarrow$) The proof is by induction on n = the length of the leftmost derivation of ω.
- Base case, n = 0
- This means that $\omega = \epsilon$ and $\alpha = S$
- We have that $(Q\ \omega\ S) = (Q\ \epsilon\ S) \vdash^*_M (Q\ \epsilon\ S) = (Q\ \epsilon\ \alpha)$.
- For the inductive step:
- Assume: $S \overset{L}{\rightarrow}_G \omega\alpha \Leftrightarrow (Q\ \omega\ S) \vdash^*_M (Q\ \epsilon\ \alpha)$, for n = k.
- Show: $S \overset{L}{\rightarrow}_G \omega\alpha \Leftrightarrow (Q\ \omega\ S) \vdash^*_M (Q\ \epsilon\ \alpha)$, for n = k + 1.
- The derivation of length k+1 looks as follows:
- $S \rightarrow u_1 \rightarrow u_2 \rightarrow \ldots \rightarrow u_n \rightarrow u_{n+1} = \omega\alpha$
- Observe that $u_n = xA\beta$, where $x \in \Sigma^* \wedge A \in N \wedge \beta \in \{N \cup \Sigma\}^*$
- The derivation step from u_n to u_{n+1} substitutes A using a rule: $A \rightarrow \theta$, where $\theta \in \{N \cup \Sigma\}^*$
- $u_{n+1} = x\theta\beta$.
- By inductive hypothesis, we have $(Q\ x\ S) \vdash^* (Q\ \epsilon\ A\beta)$
- By construction of P, there is a type 2 rule such that $(Q\ \epsilon\ A\beta) \vdash (Q\ \epsilon\ \theta\beta)$
- The leading terminals of θ, y, are in ω and that α starts with, B, the leftmost nonterminal in θ and includes β
- That is, $\omega = xy$ and $\alpha = B\beta$, where $y \in \Sigma^*$ and $\beta \in N$

Equivalence of pdas and cfgs

Let:

```
G = (make-cfg N Σ R S)      P = (cfg2pda G)
```

Lemma

$S \overset{L}{\rightarrow}_G \omega\alpha \Leftrightarrow (Q\ \omega\ S) \vdash^*_M (Q\ \epsilon\ \alpha)$, *where* $\alpha \in \{N(N \cup \Sigma)^* \cup \{\epsilon\}\} \wedge \omega \in \Sigma^*$

- ($\Rightarrow$) The proof is by induction on n = the length of the leftmost derivation of ω.
- Base case, n = 0
- This means that $\omega = \epsilon$ and $\alpha = S$
- We have that $(Q\ \omega\ S) = (Q\ \epsilon\ S) \vdash^*_M (Q\ \epsilon\ S) = (Q\ \epsilon\ \alpha)$.
- For the inductive step:
- Assume: $S \overset{L}{\rightarrow}_G \omega\alpha \Leftrightarrow (Q\ \omega\ S) \vdash^*_M (Q\ \epsilon\ \alpha)$, for n = k.
- Show: $S \overset{L}{\rightarrow}_G \omega\alpha \Leftrightarrow (Q\ \omega\ S) \vdash^*_M (Q\ \epsilon\ \alpha)$, for n = k + 1.
- The derivation of length k+1 looks as follows:
- $S \rightarrow u_1 \rightarrow u_2 \rightarrow \ldots \rightarrow u_n \rightarrow u_{n+1} = \omega\alpha$
- Observe that $u_n = xA\beta$, where $x \in \Sigma^* \wedge A \in N \wedge \beta \in \{N \cup \Sigma\}^*$
- The derivation step from u_n to u_{n+1} substitutes A using a rule: $A \rightarrow \theta$, where $\theta \in \{N \cup \Sigma\}^*$
- $u_{n+1} = x\theta\beta$.
- By inductive hypothesis, we have $(Q\ x\ S) \vdash^* (Q\ \epsilon\ A\beta)$
- By construction of P, there is a type 2 rule such that $(Q\ \epsilon\ A\beta) \vdash (Q\ \epsilon\ \theta\beta)$
- The leading terminals of θ, y, are in ω and that α starts with, B, the leftmost nonterminal in θ and includes β
- That is, $\omega = xy$ and $\alpha = B\beta$, where $y \in \Sigma^*$ and $\beta \in N$
- This means that $y\alpha = \theta\beta$

Equivalence of pdas and cfgs

Let:

```
G = (make-cfg N Σ R S)      P = (cfg2pda G)
```

Lemma

$S \overset{L}{\rightarrow}_G \omega\alpha \Leftrightarrow (Q\ \omega\ S) \vdash^*_M (Q\ \epsilon\ \alpha)$, *where* $\alpha \in \{N(N \cup \Sigma)^* \cup \{\epsilon\}\} \wedge \omega \in \Sigma^*$

- ($\Rightarrow$) The proof is by induction on n = the length of the leftmost derivation of ω.
- Base case, n = 0
- This means that $\omega = \epsilon$ and $\alpha = S$
- We have that $(Q\ \omega\ S) = (Q\ \epsilon\ S) \vdash^*_M (Q\ \epsilon\ S) = (Q\ \epsilon\ \alpha)$.
- For the inductive step:
- Assume: $S \overset{L}{\rightarrow}_G \omega\alpha \Leftrightarrow (Q\ \omega\ S) \vdash^*_M (Q\ \epsilon\ \alpha)$, for n = k.
- Show: $S \overset{L}{\rightarrow}_G \omega\alpha \Leftrightarrow (Q\ \omega\ S) \vdash^*_M (Q\ \epsilon\ \alpha)$, for n = k + 1.
- The derivation of length k+1 looks as follows:
- $S \rightarrow u_1 \rightarrow u_2 \rightarrow \ldots \rightarrow u_n \rightarrow u_{n+1} = \omega\alpha$
- Observe that $u_n = xA\beta$, where $x \in \Sigma^* \wedge A \in N \wedge \beta \in \{N \cup \Sigma\}^*$
- The derivation step from u_n to u_{n+1} substitutes A using a rule: $A \rightarrow \theta$, where $\theta \in \{N \cup \Sigma\}^*$
- $u_{n+1} = x\theta\beta$.
- By inductive hypothesis, we have $(Q\ x\ S) \vdash^* (Q\ \epsilon\ A\beta)$
- By construction of P, there is a type 2 rule such that $(Q\ \epsilon\ A\beta) \vdash (Q\ \epsilon\ \theta\beta)$
- The leading terminals of θ, y, are in ω and that α starts with, B, the leftmost nonterminal in θ and includes β
- That is, $\omega = xy$ and $\alpha = B\beta$, where $y \in \Sigma^*$ and $\beta \in N$
- This means that $y\alpha = \theta\beta$
- By construction of P there are type 3 rules that consume y to yield the following computation: $(Q\ \omega\ S) = (Q\ xy\ S) \vdash^* (Q\ y\ \theta\beta) = (Q\ y\ y\alpha) \vdash^* (Q\ \epsilon\ \alpha)$.

Equivalence of pdas and cfgs

Let:

```
G = (make-cfg N Σ R S)      P = (cfg2pda G)
```

Lemma

$S \overset{L}{\rightarrow}_G \omega\alpha \Leftrightarrow (Q\ \omega\ S) \vdash^*_M (Q\ \epsilon\ \alpha)$, *where* $\alpha \in \{N(N \cup \Sigma)^* \cup \{\epsilon\}\} \wedge \omega \in \Sigma^*$

- ($\Leftarrow$) Assume (Q ω S) $\vdash^*$ (Q ϵ α). The proof is by induction on, n, the number of type 2 transitions used by P

Equivalence of pdas and cfgs

Let:

```
G = (make-cfg N Σ R S)      P = (cfg2pda G)
```

Lemma

$S \ {}^{L}\!\rightarrow_G \omega\alpha \Leftrightarrow (Q\ \omega\ S) \vdash^*_M (Q\ \epsilon\ \alpha)$, *where* $\alpha \in \{N(N \cup \Sigma)^* \cup \{\epsilon\}\} \wedge \omega \in \Sigma^*$

- ($\Leftarrow$) Assume (Q ω S) $\vdash^*$ (Q ϵ α). The proof is by induction on, n, the number of type 2 transitions used by P
- For the base case, n $= 0$
- (Q ω S) $\vdash^*$(Q ϵ α) means that $\omega = \epsilon$ and $\alpha =$ S.
- Thus, we have that S ${}^{L}\!\rightarrow \omega\alpha = \epsilon$S $=$ S.

Equivalence of pdas and cfgs

Let:

```
G = (make-cfg N Σ R S)      P = (cfg2pda G)
```

Lemma

$S \ {}^{L}\!\rightarrow_G \omega\alpha \Leftrightarrow (Q\ \omega\ S) \vdash^*_M (Q\ \epsilon\ \alpha)$, *where* $\alpha \in \{N(N \cup \Sigma)^* \cup \{\epsilon\}\} \wedge \omega \in \Sigma^*$

- ($\Leftarrow$) Assume (Q ω S) $\vdash^*$ (Q ϵ α). The proof is by induction on, n, the number of type 2 transitions used by P
- For the base case, n $= 0$
- (Q ω S) $\vdash^*$(Q ϵ α) means that $\omega = \epsilon$ and $\alpha =$ S.
- Thus, we have that S ${}^{L}\!\rightarrow \omega\alpha = \epsilon$S $=$ S.
- For the inductive step:
- Assume: (Q ω S) $\vdash^*_M$ (Q ϵ α) $\Rightarrow$ S ${}^{L}\!\rightarrow_G \omega\alpha$, for n $=$ k.
- Show: (Q ω S) $\vdash^*_M$ (Q ϵ α) $\Rightarrow$ S ${}^{L}\!\rightarrow_G \omega\alpha$, for n $=$ k $+ 1$.

Equivalence of pdas and cfgs

Let:

```
G = (make-cfg N Σ R S)      P = (cfg2pda G)
```

Lemma

$S \overset{L}{\rightarrow}_G \omega\alpha \Leftrightarrow (Q\ \omega\ S) \vdash^*_M (Q\ \epsilon\ \alpha)$, *where* $\alpha \in \{N(N \cup \Sigma)^* \cup \{\epsilon\}\} \wedge \omega \in \Sigma^*$

- ($\Leftarrow$) Assume (Q ω S) $\vdash^*$ (Q ϵ α). The proof is by induction on, n, the number of type 2 transitions used by P
- For the base case, n $= 0$
- (Q ω S) $\vdash^*$(Q ϵ α) means that $\omega = \epsilon$ and $\alpha =$ S.
- Thus, we have that S $\overset{L}{\rightarrow}$ $\omega\alpha = \epsilon$S $=$ S.
- For the inductive step:
- Assume: (Q ω S) $\vdash^*_M$ (Q ϵ α) $\Rightarrow$ S $\overset{L}{\rightarrow}_G$ $\omega\alpha$, for n $=$ k.
- Show: (Q ω S) $\vdash^*_M$ (Q ϵ α) $\Rightarrow$ S $\overset{L}{\rightarrow}_G$ $\omega\alpha$, for n $=$ k $+ 1$.
- Let $\omega =$ xy and (A $\rightarrow \theta$)$\in$R
- A computation using n $+ 1$ type 2 rules looks as follows:
  (Q ω S) $=$ (Q xy S) $\vdash^*$ (Q y Aβ) $\vdash$ (Q y $\theta\beta$) $\vdash^*$ (Q ϵ α)

Equivalence of pdas and cfgs

Let:

```
G = (make-cfg N Σ R S)      P = (cfg2pda G)
```

Lemma

$S \overset{L}{\rightarrow}_G \omega\alpha \Leftrightarrow (Q\ \omega\ S) \vdash^*_M (Q\ \epsilon\ \alpha)$, *where* $\alpha \in \{N(N \cup \Sigma)^* \cup \{\epsilon\}\} \wedge \omega \in \Sigma^*$

- ($\Leftarrow$) Assume (Q ω S) $\vdash^*$ (Q ϵ α). The proof is by induction on, n, the number of type 2 transitions used by P
- For the base case, n $= 0$
- (Q ω S) $\vdash^*$(Q ϵ α) means that $\omega = \epsilon$ and $\alpha =$ S.
- Thus, we have that S $\overset{L}{\rightarrow}$ $\omega\alpha = \epsilon$S $=$ S.
- For the inductive step:
- Assume: (Q ω S) $\vdash^*_M$ (Q ϵ α) $\Rightarrow$ S $\overset{L}{\rightarrow}_G$ $\omega\alpha$, for n $=$ k.
- Show: (Q ω S) $\vdash^*_M$ (Q ϵ α) $\Rightarrow$ S $\overset{L}{\rightarrow}_G$ $\omega\alpha$, for n $=$ k $+ 1$.
- Let $\omega =$ xy and (A $\rightarrow \theta$)$\in$R
- A computation using n $+ 1$ type 2 rules looks as follows:
  (Q ω S) $=$ (Q xy S) $\vdash^*$ (Q y Aβ) $\vdash$ (Q y $\theta\beta$) $\vdash^*$ (Q ϵ α)
- By inductive hypothesis, we have S $\overset{L}{\rightarrow}^*_G$ xAβ
- Using the rule for A above yields S $\overset{L}{\rightarrow}^*_G$ x$\theta\beta$

Equivalence of pdas and cfgs

Let:

```
G = (make-cfg N Σ R S)      P = (cfg2pda G)
```

Lemma

$S \overset{L}{\rightarrow}_G \omega\alpha \Leftrightarrow (Q\ \omega\ S) \vdash^*_M (Q\ \epsilon\ \alpha)$, *where* $\alpha \in \{N(N \cup \Sigma)^* \cup \{\epsilon\}\} \wedge \omega \in \Sigma^*$

- ($\Leftarrow$) Assume (Q ω S) $\vdash^*$ (Q ϵ α). The proof is by induction on, n, the number of type 2 transitions used by P
- For the base case, n $= 0$
- (Q ω S) $\vdash^*$(Q ϵ α) means that $\omega = \epsilon$ and $\alpha =$ S.
- Thus, we have that S $\overset{L}{\rightarrow}$ $\omega\alpha = \epsilon$S $=$ S.
- For the inductive step:
- Assume: (Q ω S) $\vdash^*_M$ (Q ϵ α) $\Rightarrow$ S $\overset{L}{\rightarrow}_G$ $\omega\alpha$, for n $=$ k.
- Show: (Q ω S) $\vdash^*_M$ (Q ϵ α) $\Rightarrow$ S $\overset{L}{\rightarrow}_G$ $\omega\alpha$, for n $=$ k $+ 1$.
- Let $\omega =$ xy and (A $\rightarrow \theta$)$\in$R
- A computation using n $+ 1$ type 2 rules looks as follows:
  (Q ω S) $=$ (Q xy S) $\vdash^*$ (Q y Aβ) $\vdash$ (Q y $\theta\beta$) $\vdash^*$ (Q ϵ α)
- By inductive hypothesis, we have S $\overset{L}{\rightarrow}^*_G$ xAβ
- Using the rule for A above yields S $\overset{L}{\rightarrow}^*_G$ x$\theta\beta$
- Observe that (Q y $\theta\beta$) $\vdash^*$ (Q ϵ α) only uses type 3 rules
- This means that y$\alpha = \theta\beta$
- Thus, we have S $\overset{L}{\rightarrow}^*_G$ x$\theta\beta$ $=$ xy$\alpha = \omega\alpha$

Equivalence of pdas and cfgs

- We need to prove that if `P = (make-ndpda K Σ Γ S F Δ)` semidecides `L` then `L` is a context-free language
- We shall build a `cfg`, `G`, such that `L(P)` = `L(G)`

Equivalence of pdas and cfgs

- We need to prove that if `P = (make-ndpda K Σ Γ S F Δ)` semidecides `L` then `L` is a context-free language
- We shall build a `cfg`, `G`, such that `L(P)` $=$ `L(G)`
- Useful to restrict the structure of `P`'s transition rules to have what we shall call a *simple* `pda`
- A `pda` is simple if all transition rules have the following structure:

  `((Q a` β`) (P` θ`))`, such that `Q`$\neq$`S`, $\beta \in \Gamma$, $\wedge$ $|\theta| \leq 2$
- Always pops the topmost stack element and pushes zero, one, or two elements onto the stack
- No restrictions on `S`, because whenever a `pda` starts the stack is empty

Equivalence of pdas and cfgs

- To transform P into a simple pda, P', the components of P' are defined as follows:

$$
\begin{array}{lcl}
K' & = & K \cup \{X\ Y\},\ \textit{such that}\ X, Y \notin K \\
\Sigma' & = & \Sigma \\
\Gamma' & = & \Gamma \cup \{Z\},\ \textit{such that}\ Z \notin \Gamma \\
S' & = & X \\
F' & = & (Y) \\
\Delta' & = & T(\Delta) \cup \{((X\ EMP\ EMP)(S\ (Z)))\} \cup \{\forall W \in F\ ((W\ EMP\ (Z))(Y\ EMP))\}
\end{array}
$$

Equivalence of pdas and cfgs

- To transform P into a simple pda, P', the components of P' are defined as follows:

$$
\begin{array}{lcl}
K' & = & K \cup \{X\ Y\},\ \textit{such that}\ X, Y \notin K \\
\Sigma' & = & \Sigma \\
\Gamma' & = & \Gamma \cup \{Z\},\ \textit{such that}\ Z \notin \Gamma \\
S' & = & X \\
F' & = & (Y) \\
\Delta' & = & T(\Delta) \cup \{((X\ EMP\ EMP)(S\ (Z)))\} \cup \{\forall W \in F\ ((W\ EMP\ (Z))(Y\ EMP))\}
\end{array}
$$

- The new states, X and Y: start and only final states
- New stack symbol, Z, is the stack bottom symbol: whenever P would have an empty stack, P' only has Z on the stack

Equivalence of pdas and cfgs

- To transform P into a simple pda, P′, the components of P′ are defined as follows:

$$
\begin{array}{lcl}
K' & = & K \cup \{X\ Y\},\ \textit{such that}\ X, Y \notin K \\
\Sigma' & = & \Sigma \\
\Gamma' & = & \Gamma \cup \{Z\},\ \textit{such that}\ Z \notin \Gamma \\
S' & = & X \\
F' & = & (Y) \\
\Delta' & = & T(\Delta) \cup \{((X\ EMP\ EMP)(S\ (Z)))\} \cup \{\forall W \in F\ ((W\ EMP\ (Z))(Y\ EMP))\}
\end{array}
$$

- The new states, X and Y: start and only final states
- New stack symbol, Z, is the stack bottom symbol: whenever P would have an empty stack, P′ only has Z on the stack
- P′ simulates P's computation

Equivalence of pdas and cfgs

- To transform P into a simple pda, P′, the components of P′ are defined as follows:

$$
\begin{array}{lcl}
K' & = & K \cup \{X\ Y\},\ \textit{such that}\ X,Y \notin K \\
\Sigma' & = & \Sigma \\
\Gamma' & = & \Gamma \cup \{Z\},\ \textit{such that}\ Z \notin \Gamma \\
S' & = & X \\
F' & = & (Y) \\
\Delta' & = & T(\Delta) \cup \{((X\ EMP\ EMP)(S\ (Z)))\} \cup \{\forall W \in F\ ((W\ EMP\ (Z))(Y\ EMP))\}
\end{array}
$$

- The new states, X and Y: start and only final states
- New stack symbol, Z, is the stack bottom symbol: whenever P would have an empty stack, P′ only has Z on the stack
- P′ simulates P's computation
- T replaces all transitions rules that violate simplicity with equivalent rules that satisfy simplicity
- T replaces rules, ((Q a β) (P θ)), that violate simplicity in three steps:
  1. *Replace* $|\beta| \geq 2$ *rules*
  2. *Replace* $|\beta| = 0$ *rules without new rules with* $|\beta| \geq 2$
  3. *Replace* $|\theta| > 2$ *rules without new rules with* $|\beta| \neq 1$

Equivalence of pdas and cfgs

- To transform P into a simple pda, P′, the components of P′ are defined as follows:

$$
\begin{array}{lcl}
K' & = & K \cup \{X\ Y\},\ \textit{such that}\ X,Y \notin K \\
\Sigma' & = & \Sigma \\
\Gamma' & = & \Gamma \cup \{Z\},\ \textit{such that}\ Z \notin \Gamma \\
S' & = & X \\
F' & = & (Y) \\
\Delta' & = & T(\Delta) \cup \{((X\ EMP\ EMP)(S\ (Z)))\} \cup \{\forall W \in F\ ((W\ EMP\ (Z))(Y\ EMP))\}
\end{array}
$$

- The new states, X and Y: start and only final states
- New stack symbol, Z, is the stack bottom symbol: whenever P would have an empty stack, P′ only has Z on the stack
- P′ simulates P's computation
- T replaces all transitions rules that violate simplicity with equivalent rules that satisfy simplicity
- T replaces rules, ((Q a β) (P θ)), that violate simplicity in three steps:
  1. *Replace* $|\beta| \geq 2$ *rules*
  2. *Replace* $|\beta| = 0$ *rules without new rules with* $|\beta| \geq 2$
  3. *Replace* $|\theta| > 2$ *rules without new rules with* $|\beta| \neq 1$
- *Implementation in the book*

Equivalence of pdas and cfgs

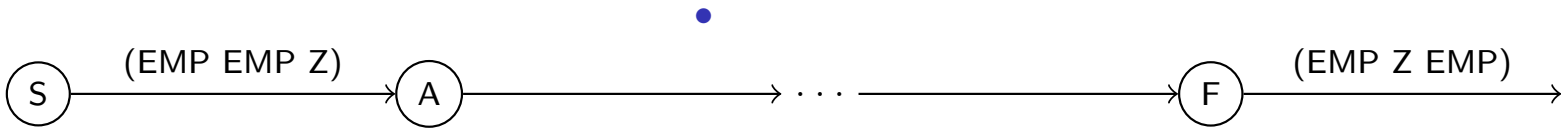
- Convert a simple pda to a cfg
- This is done by exploiting the properties of transition rules in a simple pda
- Unfortunately, the details get messy and the algorithm is not as intuitive as the algorithm to build a pda from a cfg

Equivalence of pdas and cfgs

- The transformation of a pda to a simple pda simplifies the construction of a cfg for L(P)
- The simplification stems from the fact that every pda-rule pops an element and pushes at most two elements

Equivalence of pdas and cfgs

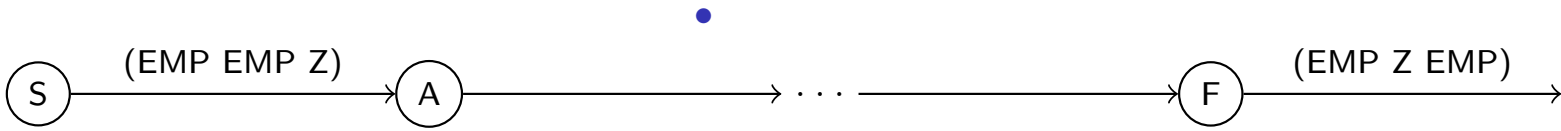
- The transformation of a pda to a simple pda simplifies the construction of a cfg for L(P)
- The simplification stems from the fact that every pda-rule pops an element and pushes at most two elements



- The middle part simulates the given pda
- The grammar that is constructed shall simulate the simple pda
- Generate all words that take the simple pda from A to F' by popping Z

Equivalence of pdas and cfgs

- The transformation of a pda to a simple pda simplifies the construction of a cfg for L(P)
- The simplification stems from the fact that every pda-rule pops an element and pushes at most two elements



- The middle part simulates the given pda
- The grammar that is constructed shall simulate the simple pda
- Generate all words that take the simple pda from A to F' by popping Z
- We may represent these words as a triple: (S Z F')
- This means that the starting nonterminal for the constructed grammar, S, must generate (S Z F'):

  S $\rightarrow$ (A Z F')

Equivalence of pdas and cfgs

- Triples need to be converted to symbols
- We shall talk about all words that take the simple pda from some state Q to some state R by popping θ: (Q θ R)
- All such triples represent the nonterminals of the grammar that is constructed and are formally defined as follows:

```
;; An list nonterminal, lnt, is a (list state symbol state).

;; Interpretation:
;;   All words that take the pda from the first state to
;;   the second state by popping the symbol off the stack.
```

Equivalence of pdas and cfgs

- Triples need to be converted to symbols
- We shall talk about all words that take the simple pda from some state Q to some state R by popping θ: (Q θ R)
- All such triples represent the nonterminals of the grammar that is constructed and are formally defined as follows:

```
;; An list nonterminal, lnt, is a (list state symbol state).

;; Interpretation:
;;   All words that take the pda from the first state to
;;   the second state by popping the symbol off the stack.
```

- We define a cfg-rule's left-hand side as follows:

```
;; A lhs is either:
;;  1. symbol
;;  2. lnt

;; Interpretation:
;;  The left-hand side of a cfg-rule is represented as either a
;;  a symbol or an lnt.
```

Equivalence of pdas and cfgs

- Triples need to be converted to symbols
- We shall talk about all words that take the simple pda from some state `Q` to some state `R` by popping θ: `(Q` θ `R)`
- All such triples represent the nonterminals of the grammar that is constructed and are formally defined as follows:

```
;; An list nonterminal, lnt, is a (list state symbol state).

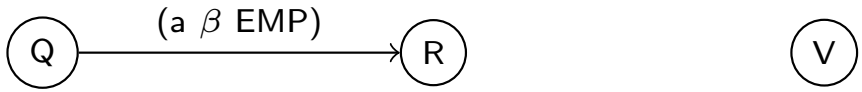
;; Interpretation:
;;   All words that take the pda from the first state to
;;   the second state by popping the symbol off the stack.
```

- We define a cfg-rule's left-hand side as follows:

```
;; A lhs is either:
;;  1. symbol
;;  2. lnt

;; Interpretation:
;;  The left-hand side of a cfg-rule is represented as either a
;;  a symbol or an lnt.
```

- How can the right-hand side of a cfg-rule be represented?
- Simple pda rules vary in what is pushed
- Rules that push 0 elements: `((Q a` β`) (R EMP))`
- After using this rule the machine may transition in zero or more steps to an arbitrary state `V`:



Equivalence of pdas and cfgs

- Triples need to be converted to symbols
- We shall talk about all words that take the simple pda from some state Q to some state R by popping θ: (Q θ R)
- All such triples represent the nonterminals of the grammar that is constructed and are formally defined as follows:

```
;; An list nonterminal, lnt, is a (list state symbol state).

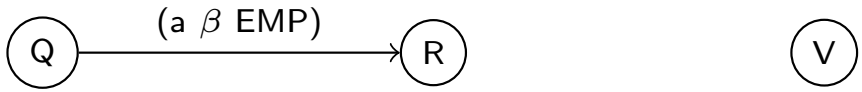
;; Interpretation:
;;   All words that take the pda from the first state to
;;   the second state by popping the symbol off the stack.
```

- We define a cfg-rule's left-hand side as follows:

```
;; A lhs is either:
;;  1. symbol
;;  2. lnt

;; Interpretation:
;;  The left-hand side of a cfg-rule is represented as either a
;;  a symbol or an lnt.
```

- How can the right-hand side of a cfg-rule be represented?
- Simple pda rules vary in what is pushed
- Rules that push 0 elements: ((Q a β) (R EMP))
- After using this rule the machine may transition in zero or more steps to an arbitrary state V:



- The grammar needs to generate all words that take the simple pda from Q to V by popping β and pushing nothing
- Such words are generated by (Q β V)
- To get to V, the machine reads a to reach R and then pops nothing to reach V in zero or more moves

Equivalence of pdas and cfgs

- Triples need to be converted to symbols
- We shall talk about all words that take the simple pda from some state Q to some state R by popping θ: (Q θ R)
- All such triples represent the nonterminals of the grammar that is constructed and are formally defined as follows:

```
;; An list nonterminal, lnt, is a (list state symbol state).

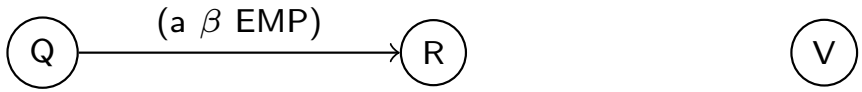
;; Interpretation:
;;   All words that take the pda from the first state to
;;   the second state by popping the symbol off the stack.
```

- We define a cfg-rule's left-hand side as follows:

```
;; A lhs is either:
;;  1. symbol
;;  2. lnt

;; Interpretation:
;;  The left-hand side of a cfg-rule is represented as either a
;;  a symbol or an lnt.
```

- How can the right-hand side of a cfg-rule be represented?
- Simple pda rules vary in what is pushed
- Rules that push 0 elements: ((Q a β) (R EMP))
- After using this rule the machine may transition in zero or more steps to an arbitrary state V:



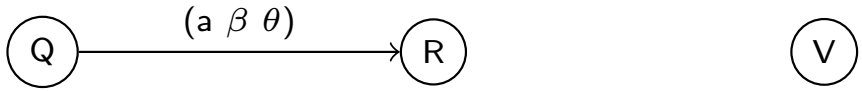
- The grammar needs to generate all words that take the simple pda from Q to V by popping β and pushing nothing
- Such words are generated by (Q β V)
- To get to V, the machine reads a to reach R and then pops nothing to reach V in zero or more moves
- The needed rule is:

  (Q β V) $\rightarrow$ a(R EMP V)

- The right-hand side of a cfg-rule may be represented by a symbol followed by an lnt.

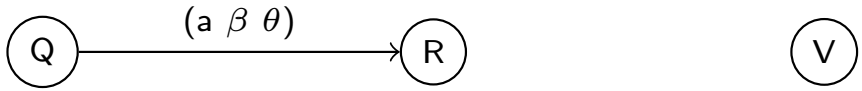
Equivalence of pdas and cfgs

- Consider the cfg-rule needed when a single element is pushed:



Equivalence of pdas and cfgs

- Consider the cfg-rule needed when a single element is pushed:



- The grammar needs to generate all words that take the simple pda from Q to V by popping β and pushing θ
- Similar to pushing nothing except that after reaching R θ must be popped
- The needed cfg-rule is:

$$\texttt{(Q } \beta \texttt{ V)} \rightarrow \texttt{a(R } \theta \texttt{ V)}$$

Equivalence of pdas and cfgs

- Consider the pda-rules that push two elements: `((Q a` β`) (R` $\theta\tau$`))`.
- After popping the first element the machine may be in an arbitrary state `W`. Graphically, consider the following:

Equivalence of pdas and cfgs

- Consider the pda-rules that push two elements: ((Q a β) (R $\theta\tau$)).
- After popping the first element the machine may be in an arbitrary state W. Graphically, consider the following:

- The grammar must be able to generate a word that starts with a followed by a word that takes the simple pda from R to W by popping θ and ending with a word that takes it from W to V by popping τ
- The needed cfg-rule may be represented as follows:

$$(\mathtt{Q}\ \beta\ \mathtt{V}) \rightarrow \mathtt{a}(\mathtt{R}\ \theta\ \mathtt{W})(\mathtt{W}\ \tau\ \mathtt{V})$$

Equivalence of pdas and cfgs

- Finally, we must consider the stopping conditions for word generation by the grammar
- The three types of production rules generated above always generate at least one `lnt`

Equivalence of pdas and cfgs

- Finally, we must consider the stopping conditions for word generation by the grammar
- The three types of production rules generated above always generate at least one `lnt`
- Once all the terminal symbols are generated, remaining nonterminals must be eliminated
- If simple `pda` is in an arbitrary state `Q` then it may remain in `Q` without popping or consuming any input:

$$\forall \texttt{Q} \in \texttt{(sm-states P')}\ \texttt{(Q EMP Q)} \rightarrow \texttt{EMP}$$

Equivalence of pdas and cfgs

- Finally, we must consider the stopping conditions for word generation by the grammar
- The three types of production rules generated above always generate at least one lnt
- Once all the terminal symbols are generated, remaining nonterminals must be eliminated
- If simple pda is in an arbitrary state Q then it may remain in Q without popping or consuming any input:

  $\forall$Q$\in$(sm-states P′) (Q EMP Q) $\rightarrow$ EMP
- The representation of the right-hand side of a cfg-rule has variety and may be defined as follows:

```
;; A rhs is either:
;;  1. symbol
;;  2. (list symbol lnt)
;;  3. (list symbol lnt lnt)
;; Interpretation:
;;  A cfg-rule right hand side is represented as either a
;;  symbol or a list with a symbol and either one or two lnt.
```

- We can now define the representation of a cfg-rule as follows:

```
;; A cfg-rl is a (list lhs ARROW rhs)
;; Interpretation:
;;   A cfg-rl represents a cfg-fule as a list with an lhs,
;;   an ARROW, and a rhs.
```

Equivalence of pdas and cfgs

- Finally, we must consider the stopping conditions for word generation by the grammar
- The three types of production rules generated above always generate at least one lnt
- Once all the terminal symbols are generated, remaining nonterminals must be eliminated
- If simple pda is in an arbitrary state Q then it may remain in Q without popping or consuming any input:

  $\forall$Q$\in$(sm-states P′) (Q EMP Q) $\rightarrow$ EMP
- The representation of the right-hand side of a cfg-rule has variety and may be defined as follows:

```
;; A rhs is either:
;;  1. symbol
;;  2. (list symbol lnt)
;;  3. (list symbol lnt lnt)
;; Interpretation:
;;  A cfg-rule right hand side is represented as either a
;;  symbol or a list with a symbol and either one or two lnt.
```

- We can now define the representation of a cfg-rule as follows:

```
;; A cfg-rl is a (list lhs ARROW rhs)
;; Interpretation:
;;   A cfg-rl represents a cfg-fule as a list with an lhs,
;;   an ARROW, and a rhs.
```

- Implementation in the textbook

Equivalence of pdas and cfgs

- HOMEWORK: 7

Properties of CFLs

- For every context-free `L` there is a `cfg` that generates its members and a `pda` that accepts its members
- We e shall study closure properties of context-free languages and how to prove that a language is not context-free

Properties of CFLs

- Context-free languages are closed under union

Properties of CFLs

- Context-free languages are closed under union
- Design Idea

Properties of CFLs

- Context-free languages are closed under union
- Design Idea
- If L_1 and L_2 are context-free languages then there are cfgs, G_1 and G_2, such that $L_1 = L(G_1)$ and $L_2 = L(G_2)$
- Without loss of generality, we assume that $(\texttt{grammar-nts}\ G_1) \cap (\texttt{grammar-nts}\ G_2) = \varnothing$

Properties of CFLs

- Context-free languages are closed under union
- Design Idea
- If L_1 and L_2 are context-free languages then there are cfgs, G_1 and G_2, such that $L_1 = L(G_1)$ and $L_2 = L(G_2)$
- Without loss of generality, we assume that $(\texttt{grammar-nts}\ G_1) \cap (\texttt{grammar-nts}\ G_2) = \varnothing$
- The grammar for $L_1 \cup L_2$ nondeterministically decides to simulate a derivation using G_1 or G_2.

Properties of CFLs

- Context-free languages are closed under union
- Design Idea
- If L_1 and L_2 are context-free languages then there are cfgs, G_1 and G_2, such that $L_1 = L(G_1)$ and $L_2 = L(G_2)$
- Without loss of generality, we assume that $(\texttt{grammar-nts}\ G_1) \cap (\texttt{grammar-nts}\ G_2) = \varnothing$
- The grammar for $L_1 \cup L_2$ nondeterministically decides to simulate a derivation using G_1 or G_2.
- A cfg for $L_1 \cup L_2$ is built using the following components:

$$
\begin{array}{lcl}
V & = & \textit{(grammar-nts } G_1\textit{)} \cup \textit{(grammar-nts } G_2\textit{)} \cup \{A\} \\
\Sigma & = & \textit{(grammar-sigma } G_1\textit{)} \cup \textit{(grammar-sigma } G_2\textit{)} \\
R & = & \textit{(grammar-rules } G_1\textit{)} \cup \textit{(grammar-rules } G_2\textit{)} \cup \\
 & & \{\textit{(S ARROW (grammar-start } G_1\textit{)) (S ARROW (grammar-start } G_2\textit{))}\} \\
S & = & \textit{A = (generate-symbol 'S (append (grammar-nts } G_1\textit{) (grammar-nts } G_2\textit{)))}
\end{array}
$$

Properties of CFLs

- ```
  ;; cfg cfg → cfg
  ;; Purpose: Construct a grammar for the union of the given grammars
  (define (cfg-union G1 G2 #:rejects [rejs '()] #:accepts [accs '()])
  ```
  ```

Properties of CFLs

- ```
  ;; cfg cfg → cfg
  ;; Purpose: Construct a grammar for the union of the given grammars
  (define (cfg-union G1 G2 #:rejects [rejs '()] #:accepts [accs '()])
  ```

- ```
  (define a2nb2nUnumb>numa (cfg-union a2nb2n
                                      numb>numa
                                      #:rejects '((b a) (b a a a b b))
                                      #:accepts '((a b) (a a b b) (b b) (b a a b b))))
  (define numb>numaUMULT3-as (cfg-union numb>numa
                                        MULT3-as
                                        #:rejects '((a a a a b b) (a a))
                                        #:accepts '((b b b) (a b a b a b))))
  ```

Properties of CFLs

- ```
  ;; cfg cfg → cfg
  ;; Purpose: Construct a grammar for the union of the given grammars
  (define (cfg-union G1 G2 #:rejects [rejs '()] #:accepts [accs '()])
  ```
- ```
    (let* [(H (grammar-rename-nts (grammar-nts G1) G2))
  ```

- ```
  (define a2nb2nUnumb>numa (cfg-union a2nb2n
                                      numb>numa
                                      #:rejects '((b a) (b a a a b b))
                                      #:accepts '((a b) (a a b b) (b b) (b a a b b))))
  (define numb>numaUMULT3-as (cfg-union numb>numa
                                        MULT3-as
                                        #:rejects '((a a a a b b) (a a))
                                        #:accepts '((b b b) (a b a b a b))))
  ```
  ```

Properties of CFLs

```
;; cfg cfg → cfg
;; Purpose: Construct a grammar for the union of the given grammars
(define (cfg-union G1 G2 #:rejects [rejs '()] #:accepts [accs '()])
  (let* [(H (grammar-rename-nts (grammar-nts G1) G2))
         (G-nts (grammar-nts G1))     (H-nts (grammar-nts H))
         (G-sigma (grammar-sigma G1)) (H-sigma (grammar-sigma H))
         (G-rules (grammar-rules G1)) (H-rules (grammar-rules H))
         (G-start (grammar-start G1)) (H-start (grammar-start H))
```

```
(define a2nb2nUnumb>numa (cfg-union a2nb2n
                                    numb>numa
                                    #:rejects '((b a) (b a a a b b))
                                    #:accepts '((a b) (a a b b) (b b) (b a a b b))))
(define numb>numaUMULT3-as (cfg-union numb>numa
                                      MULT3-as
                                      #:rejects '((a a a a b b) (a a))
                                      #:accepts '((b b b) (a b a b a b))))
```

Properties of CFLs

```
;; cfg cfg → cfg
;; Purpose: Construct a grammar for the union of the given grammars
(define (cfg-union G1 G2 #:rejects [rejs '()] #:accepts [accs '()])
  (let* [(H (grammar-rename-nts (grammar-nts G1) G2))
         (G-nts (grammar-nts G1))     (H-nts (grammar-nts H))
         (G-sigma (grammar-sigma G1)) (H-sigma (grammar-sigma H))
         (G-rules (grammar-rules G1)) (H-rules (grammar-rules H))
         (G-start (grammar-start G1)) (H-start (grammar-start H))
         (A (gen-nt (append G-nts H-nts)))]
```

```
(define a2nb2nUnumb>numa (cfg-union a2nb2n
                                    numb>numa
                                    #:rejects '((b a) (b a a a b b))
                                    #:accepts '((a b) (a a b b) (b b) (b a a b b))))
(define numb>numaUMULT3-as (cfg-union numb>numa
                                      MULT3-as
                                      #:rejects '((a a a a b b) (a a))
                                      #:accepts '((b b b) (a b a b a b))))
```

Properties of CFLs

```
;; cfg cfg → cfg
;; Purpose: Construct a grammar for the union of the given grammars
(define (cfg-union G1 G2 #:rejects [rejs '()] #:accepts [accs '()])
  (let* [(H (grammar-rename-nts (grammar-nts G1) G2))
         (G-nts (grammar-nts G1))     (H-nts (grammar-nts H))
         (G-sigma (grammar-sigma G1)) (H-sigma (grammar-sigma H))
         (G-rules (grammar-rules G1)) (H-rules (grammar-rules H))
         (G-start (grammar-start G1)) (H-start (grammar-start H))
         (A (gen-nt (append G-nts H-nts)))]
    (make-cfg (cons A (append G-nts H-nts))
              (remove-duplicates (append G-sigma H-sigma))
              (append (list (list A ARROW G-start)
                            (list A ARROW H-start))
                      G-rules
                      H-rules)
              A
              #:rejects rejs
              #:accepts accs)))

(define a2nb2nUnumb>numa (cfg-union a2nb2n
                                    numb>numa
                                    #:rejects '((b a) (b a a a b b))
                                    #:accepts '((a b) (a a b b) (b b) (b a a b b))))
(define numb>numaUMULT3-as (cfg-union numb>numa
                                      MULT3-as
                                      #:rejects '((a a a a b b) (a a))
                                      #:accepts '((b b b) (a b a b a b))))
```

Properties of CFLs

Theorem

Context-free languages are closed under union.

- Assume S $\rightarrow_G$ w

Properties of CFLs

Theorem

Context-free languages are closed under union.

- Assume S $\rightarrow_G$ w
- By construction of `G`, the derivation of `w` starts with `(grammar-start G)` and then either `(grammar-start G1)` is generated or `(grammar-start G2)` is generated

Properties of CFLs

Theorem

Context-free languages are closed under union.

- Assume S $\rightarrow_G$ w
- By construction of `G`, the derivation of `w` starts with `(grammar-start G)` and then either `(grammar-start G1)` is generated or `(grammar-start G2)` is generated
- If `(grammar-start G1)` is generated then `G` simulates `G1`
- If `(grammar-start G2)` is generated then `G` simulates `G2`

Properties of CFLs

Theorem

Context-free languages are closed under union.

- Assume S $\rightarrow_G$ w
- By construction of `G`, the derivation of `w` starts with `(grammar-start G)` and then either `(grammar-start G1)` is generated or `(grammar-start G2)` is generated
- If `(grammar-start G1)` is generated then `G` simulates `G1`
- If `(grammar-start G2)` is generated then `G` simulates `G2`
- Recall that `(grammar-nts G1)`$\cap$`(grammar-nts G2)` $= \varnothing$
- This means that either `G1` generates `w` or `G2` generates `w`
- Thus, `w` $\in$ `L(G1)`$\cup$`L(G2)`.

Properties of CFLs

- HOMEWORK: 1

Properties of CFLs

- Context-free languages are closed under concatenation.

Properties of CFLs

- Context-free languages are closed under concatenation.
- Design Idea
- There are cfgs, G_1 and G_2, such that $L_1 = L(G_1)$ and $L_2 = L(G_2)$

Properties of CFLs

- Context-free languages are closed under concatenation.
- Design Idea
- There are cfgs, G_1 and G_2, such that $L_1 = L(G_1)$ and $L_2 = L(G_2)$
- Without loss of generality, we assume that (grammar-nts G_1) $\cap$ (grammar-nts G_2) = $\varnothing$
- We shall build a grammar, G, for L_1L_2.

Properties of CFLs

- Context-free languages are closed under concatenation.
- Design Idea
- There are cfgs, G_1 and G_2, such that $L_1 = L(G_1)$ and $L_2 = L(G_2)$
- Without loss of generality, we assume that $(\texttt{grammar-nts}\ G_1) \cap (\texttt{grammar-nts}\ G_2) = \varnothing$
- We shall build a grammar, G, for L_1L_2.
- Every word in L_1L_2 may be written as $w = xy$, such that $x \in L_1$ and $y \in L_2$

Properties of CFLs

- Context-free languages are closed under concatenation.
- Design Idea
- There are cfgs, G_1 and G_2, such that $L_1 = L(G_1)$ and $L_2 = L(G_2)$
- Without loss of generality, we assume that (grammar-nts G_1) $\cap$ (grammar-nts G_2) $= \varnothing$
- We shall build a grammar, G, for L_1L_2.
- Every word in L_1L_2 may be written as $w = xy$, such that $x \in L_1$ and $y \in L_2$
- A cfg for L_1L_2 is built using the following components:

  V = (grammar-nts G_1) $\cup$ (grammar-nts G_2) $\cup$ {A}

  Σ = (grammar-sigma G_1) $\cup$ (grammar-sigma G_2)

  R = (grammar-rules G_1) $\cup$ (grammar-rules G_2) $\cup$ {S ARROW (grammar-start G_1)(grammar-start G_2)}

  S = A = (generate-symbol 'S (append (grammar-nts G_1) (grammar

Properties of CFLs

- ```
  ;; cfg cfg → cfg
  ;; Purpose: Build a cfg for the concatenation of the given cfgs
  (define (cfg-concat G1 G2 #:rejects [rejs '()] #:accepts [accs '()])
  ```
  ```

Properties of CFLs

- ```
  ;; cfg cfg → cfg
  ;; Purpose: Build a cfg for the concatenation of the given cfgs
  (define (cfg-concat G1 G2 #:rejects [rejs '()] #:accepts [accs '()])
  ```

- ```
  ;; Tests for cfg-concat
  (define a2nb2n-numb>numa (cfg-concat a2nb2n
                                       numb>numa
                                       #:rejects '((b a) (a a a) (a a b b))
                                       #:accepts '((b b b) (a b b) (a a b b b b b))))
  (define MULT3-as-a2nb2n (cfg-concat MULT3-as
                                      a2nb2n
                                      #:rejects '((a a) (b b b b a))
                                      #:accepts '((a a b b) (b b b b b b))))
  ```

Properties of CFLs

- ```
  ;; cfg cfg → cfg
  ;; Purpose: Build a cfg for the concatenation of the given cfgs
  (define (cfg-concat G1 G2 #:rejects [rejs '()] #:accepts [accs '()])
  ```
- ```
    (let* [(G2 (grammar-rename-nts (grammar-nts G1) G2))
  ```

- ```
  ;; Tests for cfg-concat
  (define a2nb2n-numb>numa (cfg-concat a2nb2n
                                       numb>numa
                                       #:rejects '((b a) (a a a) (a a b b))
                                       #:accepts '((b b b) (a b b) (a a b b b b b))))
  (define MULT3-as-a2nb2n (cfg-concat MULT3-as
                                      a2nb2n
                                      #:rejects '((a a) (b b b b a))
                                      #:accepts '((a a b b) (b b b b b b))))
  ```
  ```

Properties of CFLs

```
;; cfg cfg → cfg
;; Purpose: Build a cfg for the concatenation of the given cfgs
(define (cfg-concat G1 G2 #:rejects [rejs '()] #:accepts [accs '()])
  (let* [(G2 (grammar-rename-nts (grammar-nts G1) G2))
         (G1-nts   (grammar-nts G1))   (G2-nts (grammar-nts G2))
         (G1-sigma (grammar-sigma G1)) (G2-sigma (grammar-sigma G2))
         (G1-rules (grammar-rules G1)) (G2-rules (grammar-rules G2))
         (G1-start (grammar-start G1)) (G2-start (grammar-start G2))
```

```
;; Tests for cfg-concat
(define a2nb2n-numb>numa (cfg-concat a2nb2n
                                     numb>numa
                                     #:rejects '((b a) (a a a) (a a b b))
                                     #:accepts '((b b b) (a b b) (a a b b b b b))))
(define MULT3-as-a2nb2n (cfg-concat MULT3-as
                                    a2nb2n
                                    #:rejects '((a a) (b b b b a))
                                    #:accepts '((a a b b) (b b b b b b))))
```

Properties of CFLs

```
;; cfg cfg → cfg
;; Purpose: Build a cfg for the concatenation of the given cfgs
(define (cfg-concat G1 G2 #:rejects [rejs '()] #:accepts [accs '()])
  (let* [(G2 (grammar-rename-nts (grammar-nts G1) G2))
         (G1-nts   (grammar-nts G1))   (G2-nts (grammar-nts G2))
         (G1-sigma (grammar-sigma G1)) (G2-sigma (grammar-sigma G2))
         (G1-rules (grammar-rules G1)) (G2-rules (grammar-rules G2))
         (G1-start (grammar-start G1)) (G2-start (grammar-start G2))
         (A (gen-nt (append G1-nts G2-nts)))]
```

```
;; Tests for cfg-concat
(define a2nb2n-numb>numa (cfg-concat a2nb2n
                                     numb>numa
                                     #:rejects '((b a) (a a a) (a a b b))
                                     #:accepts '((b b b) (a b b) (a a b b b b b))))
(define MULT3-as-a2nb2n (cfg-concat MULT3-as
                                    a2nb2n
                                    #:rejects '((a a) (b b b b a))
                                    #:accepts '((a a b b) (b b b b b b))))
```

Properties of CFLs

```
;; cfg cfg → cfg
;; Purpose: Build a cfg for the concatenation of the given cfgs
(define (cfg-concat G1 G2 #:rejects [rejs '()] #:accepts [accs '()])
  (let* [(G2 (grammar-rename-nts (grammar-nts G1) G2))
         (G1-nts   (grammar-nts G1))   (G2-nts (grammar-nts G2))
         (G1-sigma (grammar-sigma G1)) (G2-sigma (grammar-sigma G2))
         (G1-rules (grammar-rules G1)) (G2-rules (grammar-rules G2))
         (G1-start (grammar-start G1)) (G2-start (grammar-start G2))
         (A (gen-nt (append G1-nts G2-nts)))]
    (make-cfg (cons A (append G1-nts G2-nts))
              (remove-duplicates (append G1-sigma G2-sigma))
              (append
                (list (list A ARROW (los->symbol (list G1-start G2-start))))
                G1-rules
                G2-rules)
              A
              #:rejects rejs
              #:accepts accs)))

;; Tests for cfg-concat
(define a2nb2n-numb>numa (cfg-concat a2nb2n
                                     numb>numa
                                     #:rejects '((b a) (a a a) (a a b b))
                                     #:accepts '((b b b) (a b b) (a a b b b b b))))
(define MULT3-as-a2nb2n (cfg-concat MULT3-as
                                    a2nb2n
                                    #:rejects '((a a) (b b b b a))
                                    #:accepts '((a a b b) (b b b b b b))))
```

Properties of CFLs

Theorem

Context-free languages are closed under concatenation.

- HOMEWORK: 2 (proof), 3

Properties of CFLs

- Context-free languages are closed under Kleene star

Properties of CFLs

- Context-free languages are closed under Kleene star
- Design idea
- Given that `L` is a context-free language, there exists a `G` = `(make-cfg N` Σ `R S)` such that `L` = `L(G)`

Properties of CFLs

- Context-free languages are closed under Kleene star
- Design idea
- Given that `L` is a context-free language, there exists a `G` = `(make-cfg N` Σ `R S)` such that `L` = `L(G)`
- A `cfg` for `L`* must generate 0 or more words in `L`

Properties of CFLs

- Context-free languages are closed under Kleene star
- Design idea
- Given that `L` is a context-free language, there exists a `G` = `(make-cfg N` Σ `R S)` such that `L` = `L(G)`
- A `cfg` for `L`* must generate 0 or more words in `L`
- Each word in `L` is generated by `S`
- A `cfg` for `L`* must generate 0 or more `S`s.

Properties of CFLs

- Context-free languages are closed under Kleene star
- Design idea
- Given that `L` is a context-free language, there exists a `G` = `(make-cfg N Σ R S)` such that `L` = `L(G)`
- A `cfg` for `L`* must generate 0 or more words in `L`
- Each word in `L` is generated by `S`
- A `cfg` for `L`* must generate 0 or more `S`s.
- A `cfg` for `L`* may be constructed using the following components:

$$
\begin{array}{lcl}
V & = & \textit{(grammar-nts G)} \cup \{\textit{A}\} \\
\Sigma & = & \textit{(grammar-sigma G)} \\
R & = & \textit{(grammar-rules G)} \cup \{\textit{(S ARROW EMP) (S ARROW SS}\} \\
S & = & \textit{A = (generate-symbol 'S (grammar-nts G))}
\end{array}
$$

Properties of CFLs

- QUIZ: 4-5 (due in a week)

Properties of CFLs

- Like regular languages, context-free languages are infinite and exhibit periodicity

Properties of CFLs

- Like regular languages, context-free languages are infinite and exhibit periodicity
- There are languages that are not context-free
- We shall prove a pumping theorem for context-free languages

Properties of CFLs

- To simplify our work, bound a parse tree's yield length

Properties of CFLs

- To simplify our work, bound a parse tree's yield length
- Define κ as the longest righthand side of any rule:

```
;; L = a^nb^n
(define a2nb2n (make-cfg '(S)
                         '(a b)
                         `((S ,ARROW ,EMP)
                           (S ,ARROW aSb))
                         'S))

;; L = w | w in (a b)* AND  w has more b than a
(define numb>numa (make-cfg '(S A)
                            '(a b)
                            `((S ,ARROW b)
                              (S ,ARROW AbA)
                              (A ,ARROW AaAbA)
                              (A ,ARROW AbAaA)
                              (A ,ARROW ,EMP)
                              (A ,ARROW bA))
                            'S))
```

- For a2nb2n, κ is 3 because its longest righthand side is aSb
- For numb>numa, κ is 5.

Properties of CFLs

- To simplify our work, bound a parse tree's yield length
- Define κ as the longest righthand side of any rule:

```
;; L = a^nb^n
(define a2nb2n (make-cfg '(S)
                         '(a b)
                         `((S ,ARROW ,EMP)
                           (S ,ARROW aSb))
                         'S))

;; L = w | w in (a b)* AND  w has more b than a
(define numb>numa (make-cfg '(S A)
                            '(a b)
                            `((S ,ARROW b)
                              (S ,ARROW AbA)
                              (A ,ARROW AaAbA)
                              (A ,ARROW AbAaA)
                              (A ,ARROW ,EMP)
                              (A ,ARROW bA))
                            'S))
```

- For `a2nb2n`, κ is 3 because its longest righthand side is `aSb`
- For `numb>numa`, κ is 5.
- Given a `cfg` `G`, the yield's length for any parse tree is bounded by the height of the tree and κ

Properties of CFLs

Lemma

T's yield's length $\leq \kappa^h$.

- The proof is by induction on h

Properties of CFLs

Lemma

T's yield's length $\leq \kappa^h$.

- The proof is by induction on h
- Base case: h $= 1$
- A parse tree of height 1 means that a single rule has been used
- The yield's length $\leq \kappa = \kappa^1 = \kappa^h$

Properties of CFLs

Lemma

T's yield's length $\leq \kappa^h$.

- The proof is by induction on h
- Base case: h $= 1$
- A parse tree of height 1 means that a single rule has been used
- The yield's length $\leq \kappa = \kappa^1 = \kappa^h$
- Inductive Step
- Assume: T's yield's length $\leq \kappa^n$, for h $=$ n
- Show: T's yield's length $\leq \kappa^{n+1}$, for h $=$ n $+ 1$

Properties of CFLs

Lemma

T's yield's length $\leq \kappa^h$.

- The proof is by induction on h
- Base case: h $= 1$
- A parse tree of height 1 means that a single rule has been used
- The yield's length $\leq \kappa = \kappa^1 = \kappa^h$
- Inductive Step
- Assume: T's yield's length $\leq \kappa^n$, for h $=$ n
- Show: T's yield's length $\leq \kappa^{n+1}$, for h $=$ n $+ 1$
- Consider the structure of a parse tree of height n $+ 1$
  - Root
  - At most κ subtrees of height n

Properties of CFLs

Lemma

T's yield's length $\leq \kappa^h$.

- The proof is by induction on h
- Base case: h $= 1$
- A parse tree of height 1 means that a single rule has been used
- The yield's length $\leq \kappa = \kappa^1 = \kappa^h$
- Inductive Step
- Assume: T's yield's length $\leq \kappa^n$, for h $=$ n
- Show: T's yield's length $\leq \kappa^{n+1}$, for h $=$ n $+ 1$
- Consider the structure of a parse tree of height n $+ 1$
  - Root
  - At most κ subtrees of height n
- Let T_i be any of these subtrees
- By inductive hypothesis, T_i's yield's length $\leq \kappa^n$
- If every T_i's yield's length is κ^n then T's yield's length is at most $\kappa * \kappa^n$
- Thus, T's yield's length $\leq \kappa^{n+1}$. $\square$

Properties of CFLs

Lemma

T's yield's length $\leq \kappa^h$.

- The proof is by induction on h
- Base case: h $= 1$
- A parse tree of height 1 means that a single rule has been used
- The yield's length $\leq \kappa = \kappa^1 = \kappa^h$
- Inductive Step
- Assume: T's yield's length $\leq \kappa^n$, for h $=$ n
- Show: T's yield's length $\leq \kappa^{n+1}$, for h $=$ n $+ 1$
- Consider the structure of a parse tree of height n $+ 1$
  - Root
  - At most κ subtrees of height n
- Let T_i be any of these subtrees
- By inductive hypothesis, T_i's yield's length $\leq \kappa^n$
- If every T_i's yield's length is κ^n then T's yield's length is at most $\kappa * \kappa^n$
- Thus, T's yield's length $\leq \kappa^{n+1}$. $\square$
- Let `w`$\in$`L(G)` such that $|$`w`$| > \kappa^h$
- `w`'s parse tree must have a path longer than `h`

Properties of CFLs

Theorem

If $w \in L(G) \wedge |w| > \kappa^{|V|}$ *then* $w = uvxyz$ *such that* $(v \neq \texttt{EMP} \vee y \neq \texttt{EMP})$, $\exists k\ |vxy| \leq k$, *and* $\forall n \geq 0$ $uv^n xy^n z \in L(G)$.

- Proof

Properties of CFLs

Theorem

If $w \in L(G) \wedge |w| > \kappa^{|V|}$ *then* $w = uvxyz$ *such that* $(v \neq EMP \vee y \neq EMP)$, $\exists k\ |vxy| \leq k$, *and* $\forall n \geq 0$ $uv^n xy^n z \in L(G)$.

- Proof
- Let T be the parse tree for w rooted at S with the smallest number of leaves

Properties of CFLs

Theorem

If $w \in L(G) \wedge |w| > \kappa^{|V|}$ *then* $w = uvxyz$ *such that* $(v \neq EMP \vee y \neq EMP)$, $\exists k\ |vxy| \leq k$, *and* $\forall n \geq 0$ $uv^n xy^n z \in L(G)$.

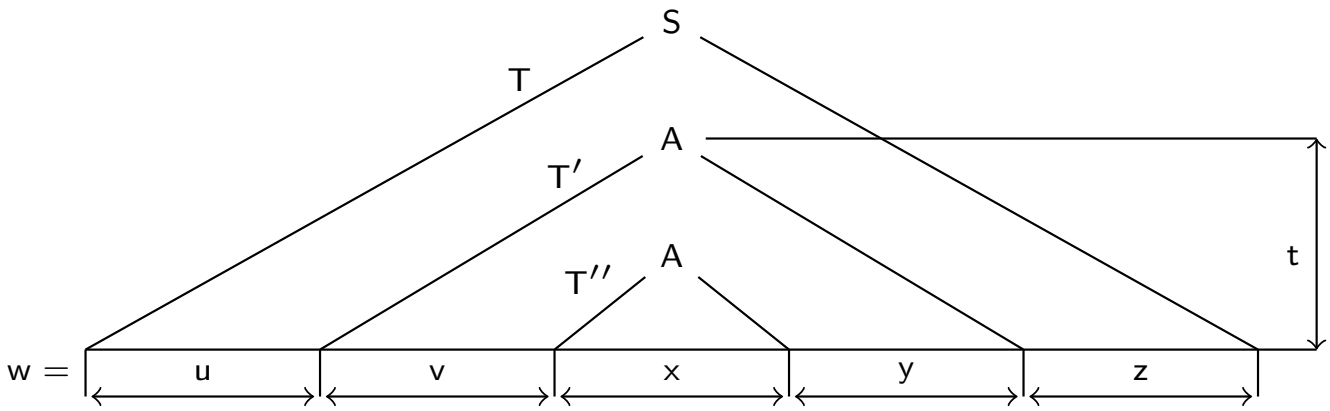
- Proof
- Let T be the parse tree for w rooted at S with the smallest number of leaves
- $|w| > \kappa^{|V|} \Rightarrow$ path in T of length at least $|V| + 1$
- $|V| + 2$ nodes of which the last must be a terminal symbol
- $|V| + 1$ nonterminals
- Must have a repeated nonterminal

Properties of CFLs

Theorem

If $w \in L(G) \wedge |w| > \kappa^{|V|}$ *then* $w = uvxyz$ *such that* $(v \neq EMP \vee y \neq EMP)$, $\exists k\ |vxy| \leq k$, *and* $\forall n \geq 0$ $uv^n xy^n z \in L(G)$.

- Proof
- Let T be the parse tree for w rooted at S with the smallest number of leaves
- $|w| > \kappa^{|V|} \Rightarrow$ path in T of length at least $|V| + 1$
- $|V| + 2$ nodes of which the last must be a terminal symbol
- $|V| + 1$ nonterminals
- Must have a repeated nonterminal



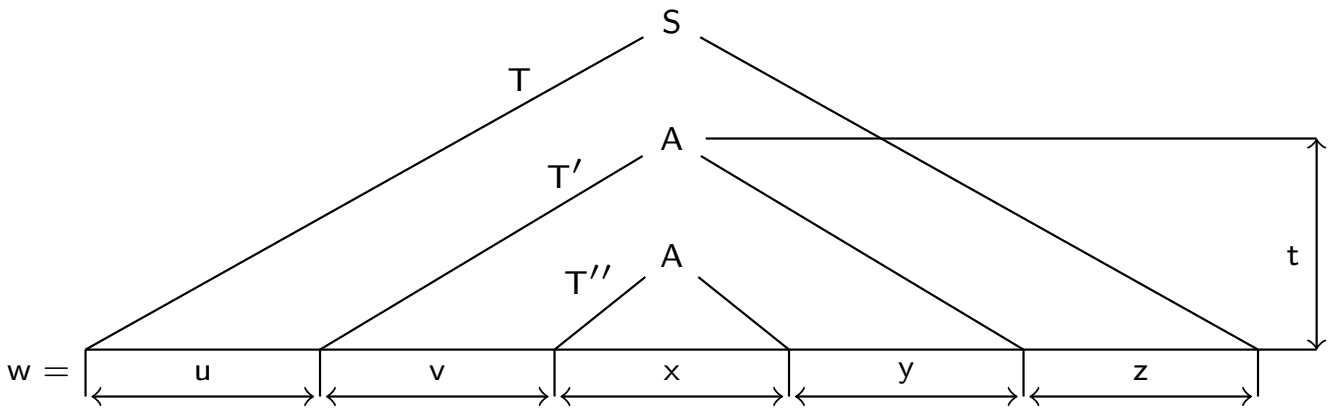
- Observe that the part of T' excluding T'' may be repeated an arbitrary number of times or may be excluded resulting in a word that is in L(G)
- $uv^n xy^n z \in L(G)$ for $n \geq 0$

Properties of CFLs

Theorem

If $w \in L(G) \wedge |w| > \kappa^{|V|}$ *then* $w = uvxyz$ *such that* $(v \neq EMP \vee y \neq EMP)$, $\exists k\ |vxy| \leq k$, *and* $\forall n \geq 0$ $uv^n xy^n z \in L(G)$.

- Proof
- Let T be the parse tree for w rooted at S with the smallest number of leaves
- $|w| > \kappa^{|V|} \Rightarrow$ path in T of length at least $|V| + 1$
- $|V| + 2$ nodes of which the last must be a terminal symbol
- $|V| + 1$ nonterminals
- Must have a repeated nonterminal



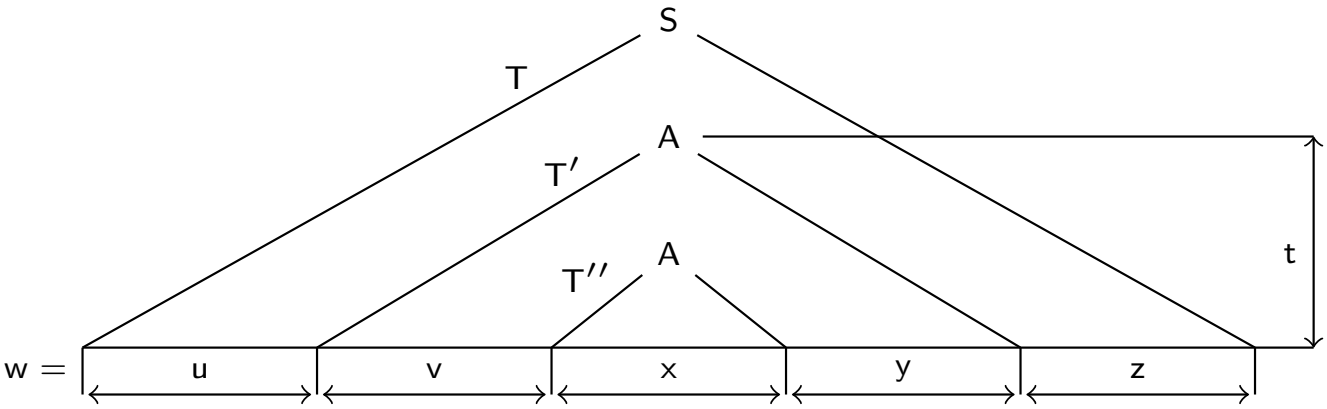
- Observe that the part of T' excluding T'' may be repeated an arbitrary number of times or may be excluded resulting in a word that is in L(G)
- $uv^n xy^n z \in L(G)$ for $n \geq 0$
- Let t be the height of T'
- The yield of T', vxy, has a length of at most κ^t
- This is the constant k that bounds the length of the subword that contains the parts that may be repeated an arbitrary number of times: $|vxy| \leq \kappa^t = k$

Properties of CFLs

Theorem

If $w \in L(G) \wedge |w| > \kappa^{|V|}$ *then* $w = uvxyz$ *such that* $(v \neq EMP \vee y \neq EMP)$, $\exists k\ |vxy| \leq k$, *and* $\forall n \geq 0$ $uv^n xy^n z \in L(G)$.

- Proof
- Let T be the parse tree for w rooted at S with the smallest number of leaves
- $|w| > \kappa^{|V|} \Rightarrow$ path in T of length at least $|V| + 1$
- $|V| + 2$ nodes of which the last must be a terminal symbol
- $|V| + 1$ nonterminals
- Must have a repeated nonterminal



- Observe that the part of T' excluding T'' may be repeated an arbitrary number of times or may be excluded resulting in a word that is in L(G)
- $uv^n xy^n z \in L(G)$ for $n \geq 0$
- Let t be the height of T'
- The yield of T', vxy, has a length of at most κ^t
- This is the constant k that bounds the length of the subword that contains the parts that may be repeated an arbitrary number of times: $|vxy| \leq \kappa^t = k$
- Finally, observe that if $vy = EMP$ then there is a parse tree with fewer leaves that generates w:

  $S \rightarrow^* uAz \rightarrow^* uxz = w$
- Smaller parse tree omits the derivation of A from A
- Contradicts our assumption that T is the parse tree for w with the smallest number of leaves
- Therefore, $vy \neq EMP \Rightarrow 0 < |vxy| \leq k \leq \kappa^{|V|}$ □

Properties of CFLs

Theorem

$L = \{a^n b^n c^n \mid n \geq 0\}$ is not context-free.

- Let w = $a^k b^k c^k$

Properties of CFLs

Theorem

$L = \{a^n b^n c^n \mid n \geq 0\}$ is not context-free.

- Let w = $a^k b^k c^k$
- We must determine the possible values `vxy` may take
- `vxy` may not contains as, bs, and cs because the length of `vxy` must be less than or equal to `k`
- `vxy` may contain only one letter variety or may contain two letter varieties by straddling the border between as and bs or between bs and cs

Properties of CFLs

Theorem

$L = \{a^n b^n c^n \mid n \geq 0\}$ is not context-free.

- Let w = $a^k b^k c^k$
- We must determine the possible values vxy may take
- vxy may not contains as, bs, and cs because the length of vxy must be less than or equal to k
- vxy may contain only one letter variety or may contain two letter varieties by straddling the border between as and bs or between bs and cs
- We prove that L is not context-free as follows:

vxy	Argument
a^+	Pump up once on v and y. The resulting word has more as than bs and than cs. Therefore, it is not in L.
b^+	Pump up once on v and y. The resulting word has more bs than as and than cs. Therefore, it is not in L.
c^+	Pump up once on v and y. The resulting word has more cs than as and than bs. Therefore, it is not in L.
a^+b^+	Pump up once on v and y. The resulting word has a b before an a. Therefore, it is not in L.
b^+c^+	Pump up once on v and y. The resulting word has a c before a b. Therefore, it is not in L.

- No matter where the vxy window is placed in w, a word that is not in w may be generated
- We may conclude that L is not context-free. □

Properties of CFLs

Theorem

$L = \{a^n \mid n$ is a square$\}$ is not context-free.

- Let w $= a^{k^2}$

Properties of CFLs

Theorem

$L = \{a^n \mid n$ is a square$\}$ is not context-free.

- Let $w = a^{k^2}$
- $vy = a^i$, where $i > 0$

Properties of CFLs

Theorem

$L = \{a^n \mid n$ is a square$\}$ is not context-free.

- Let $w = a^{k^2}$
- $vy = a^i$, where $i > 0$
- If we pump up once the resulting word is a^{K^2+i}

Properties of CFLs

Theorem

$L = \{a^n \mid n$ is a square$\}$ is not context-free.

- Let w $= a^{k^2}$
- vy $= a^i$, where $i > 0$
- If we pump up once the resulting word is a^{K^2+i}
- Observe that there are no words in L with a length greater than $|a^{K^2}|$ and less than $|a^{(k+1)^2}|$
- After a^{k^2} the next word in L has length $K^2 + 2K + 1$

Properties of CFLs

Theorem

$L = \{a^n \mid n$ is a square$\}$ is not context-free.

- Let w $= a^{k^2}$
- vy $= a^i$, where $i > 0$
- If we pump up once the resulting word is a^{K^2+i}
- Observe that there are no words in L with a length greater than $|a^{K^2}|$ and less than $|a^{(k+1)^2}|$
- After a^{k^2} the next word in L has length $K^2 + 2K + 1$
- In order for a^{k^2+i} to be in L, i must at a minimum be of length $2k + 1$
- Impossible because |vxy| must be less than or equal to k
- i cannot be greater than or equal to $2k + 1$

Properties of CFLs

Theorem

$L = \{a^n \mid n$ is a square$\}$ is not context-free.

- Let w $= a^{k^2}$
- vy $= a^i$, where $i > 0$
- If we pump up once the resulting word is a^{K^2+i}
- Observe that there are no words in L with a length greater than $|a^{K^2}|$ and less than $|a^{(k+1)^2}|$
- After a^{k^2} the next word in L has length $K^2 + 2K + 1$
- In order for a^{k^2+i} to be in L, i must at a minimum be of length $2k + 1$
- Impossible because |vxy| must be less than or equal to k
- i cannot be greater than or equal to $2k + 1$
- No matter where the vxy window is placed in w, a word that is not in w may be generated
- Therefore, we may conclude that L is not context-free □

Properties of CFLs

- HOMEWORK: 6, 7, 9, 10

Properties of CFLs

Theorem

Context-free languages are not closed under intersection.

- Proof

Properties of CFLs

Theorem

Context-free languages are not closed under intersection.

- Proof
- Let $L_1 = \{a^n b^n c^m \mid m,n \geq 0\}$
- $L_2 = \{a^m b^n c^n \mid m,n \geq 0\}$
- Both are context-free

Properties of CFLs

Theorem

Context-free languages are not closed under intersection.

- Proof
- Let $L_1 = \{a^n b^n c^m \mid m,n \geq 0\}$
- $L_2 = \{a^m b^n c^n \mid m,n \geq 0\}$
- Both are context-free
- Consider $L = L_1 \cap L_2$

Properties of CFLs

Theorem

Context-free languages are not closed under intersection.

- Proof
- Let $L_1 = \{a^n b^n c^m \mid m,n \geq 0\}$
- $L_2 = \{a^m b^n c^n \mid m,n \geq 0\}$
- Both are context-free
- Consider $L = L_1 \cap L_2$
- $L = \{a^n b^n c^n \mid n \geq 0\}$, which we know is not context-free
- Therefore, context-free languages are not closed under intersection.

Properties of CFLs

Theorem

Context-free languages are not closed under complement.

- Proof

Properties of CFLs

Theorem

Context-free languages are not closed under complement.

- Proof
- $L_1 \cap L_2 =$ `(complement ((complement` L_1`) ∪ (complement` L_2`)))`
- If context-free languages were closed under complement then they would also be closed under intersection, which we know they are not
- Therefore, context free languages are not closed under complement

Properties of CFLs

- Although context-free languages are not closed under intersection, there exist context-free languages that result in a context-free language when intersected

Properties of CFLs

- Although context-free languages are not closed under intersection, there exist context-free languages that result in a context-free language when intersected
- Recall that all regular languages are context-free
- The intersection of a context-free language and a regular language is a context-free language
- How can we prove this?

Properties of CFLs

- Although context-free languages are not closed under intersection, there exist context-free languages that result in a context-free language when intersected
- Recall that all regular languages are context-free
- The intersection of a context-free language and a regular language is a context-free language
- How can we prove this?
- Develop a new constructor for a context-free language: constructive proof

Properties of CFLs

- Design Idea

Properties of CFLs

- Design Idea
- Have a pda: $\texttt{M}_1$ = (make-ndpda $\texttt{K}_1$ Σ_1 Γ $\texttt{S}_1$ $\texttt{F}_1$ $\texttt{R}_1$)
- Have a dfa: $\texttt{M}_2$ = (make-dfa $\texttt{K}_2$ Σ_2 $\texttt{S}_2$ $\texttt{F}_2$ $\texttt{R}_2$)

Properties of CFLs

- Design Idea
- Have a pda: $\texttt{M}_1$ = (make-ndpda $\texttt{K}_1$ Σ_1 Γ $\texttt{S}_1$ $\texttt{F}_1$ $\texttt{R}_1$)
- Have a dfa: $\texttt{M}_2$ = (make-dfa $\texttt{K}_2$ Σ_2 $\texttt{S}_2$ $\texttt{F}_2$ $\texttt{R}_2$)
- Build a pda that simultaneously simulates $\texttt{M}_1$ and $\texttt{M}_2$
- Only accept if both $\texttt{M}_1$ and $\texttt{M}_2$ accept.

Properties of CFLs

- Design Idea
- Have a pda: M_1 = (make-ndpda K_1 Σ_1 Γ S_1 F_1 R_1)
- Have a dfa: M_2 = (make-dfa K_2 Σ_2 S_2 F_2 R_2)
- Build a pda that simultaneously simulates M_1 and M_2
- Only accept if both M_1 and M_2 accept.
- Basic idea: Each state of the new pda be a *super state* that represents a state in M_1 and a state in M_2
- Simulate all the transitions of M_1 and track the transitions made by M_2
- If M_1 has a rule that moves from P to Q on an a then for every state $Y \in M_2$ the new pda has a rule that moves from super state (P Y) to super state (Q T)
- If M_1 has a rule that moves from P to Q on EMP then for every state $Y \in M_2$ the new pda has a rule that moves from super state (P Y) to super state (Q Y) to simulate that M_2 does not change state

Properties of CFLs

- Design Idea
- Have a pda: M_1 = (make-ndpda K_1 Σ_1 Γ S_1 F_1 R_1)
- Have a dfa: M_2 = (make-dfa K_2 Σ_2 S_2 F_2 R_2)
- Build a pda that simultaneously simulates M_1 and M_2
- Only accept if both M_1 and M_2 accept.
- Basic idea: Each state of the new pda be a *super state* that represents a state in M_1 and a state in M_2
- Simulate all the transitions of M_1 and track the transitions made by M_2
- If M_1 has a rule that moves from P to Q on an a then for every state $Y \in M_2$ the new pda has a rule that moves from super state (P Y) to super state (Q T)
- If M_1 has a rule that moves from P to Q on EMP then for every state $Y \in M_2$ the new pda has a rule that moves from super state (P Y) to super state (Q Y) to simulate that M_2 does not change state
-
```
;; Push or pop elements, stacke, is either
;;  1. EMP
;;  2. (listof symbol)

;; A super state, ss, is a (list state state)

;; A pda super rule, ssrule, is:
;;  (list (list ss symbol stacke) (list ss stacke))
```

Properties of CFLs

- ```
  ;; pda dfa → pda  Purpose: Build intersection pda from given machines
  (define (pda-intersect-dfa a-pda a-dfa #:rejects [rejs '()] #:accepts [accs '()])
  ```
  ```

Properties of CFLs

- ```
  ;; pda dfa → pda  Purpose: Build intersection pda from given machines
  (define (pda-intersect-dfa a-pda a-dfa #:rejects [rejs '()] #:accepts [accs '()])
  ```

- ```
   (let* [(pda-sts (sm-states a-pda))  (dfa-sts (sm-states a-dfa))
          (pda-rls (sm-rules a-pda))   (dfa-rls (sm-rules a-dfa))
  ```

Properties of CFLs

- ```
  ;; pda dfa → pda  Purpose: Build intersection pda from given machines
  (define (pda-intersect-dfa a-pda a-dfa #:rejects [rejs '()] #:accepts [accs '()])
  ```

- ```
    (let* [(pda-sts (sm-states a-pda))  (dfa-sts (sm-states a-dfa))
           (pda-rls (sm-rules a-pda))   (dfa-rls (sm-rules a-dfa))
  ```
- ```
           (K (cartesian-product pda-sts dfa-sts))
           (start (list (sm-start a-pda) (sm-start a-dfa)))
           (F (cartesian-product (sm-finals a-pda) (sm-finals a-dfa)))
  ```
  ```

Properties of CFLs

- ```
  ;; pda dfa → pda  Purpose: Build intersection pda from given machines
  (define (pda-intersect-dfa a-pda a-dfa #:rejects [rejs '()] #:accepts [accs '()])
  ```

- ```
    (let* [(pda-sts (sm-states a-pda))  (dfa-sts (sm-states a-dfa))
           (pda-rls (sm-rules a-pda))   (dfa-rls (sm-rules a-dfa))
  ```
- ```
           (K (cartesian-product pda-sts dfa-sts))
           (start (list (sm-start a-pda) (sm-start a-dfa)))
           (F (cartesian-product (sm-finals a-pda) (sm-finals a-dfa)))
  ```
- ```
           (non-EMP-pda-rls (filter (λ (r) (not (eq? (second (first r)) EMP))) pda-rls))
           (EMP-pda-rls (filter (λ (r) (eq? (second (first r)) EMP)) pda-rls))
           (non-EMP-rls (make-nonEMP-rls non-EMP-pda-rls dfa-sts dfa-rls))
           (EMP-rls (make-EMP-rls EMP-pda-rls dfa-sts))
  ```

Properties of CFLs

```
;; pda dfa → pda  Purpose: Build intersection pda from given machines
(define (pda-intersect-dfa a-pda a-dfa #:rejects [rejs '()] #:accepts [accs '()])
```

```
(let* [(pda-sts (sm-states a-pda))  (dfa-sts (sm-states a-dfa))
       (pda-rls (sm-rules a-pda))   (dfa-rls (sm-rules a-dfa))
       (K (cartesian-product pda-sts dfa-sts))
       (start (list (sm-start a-pda) (sm-start a-dfa)))
       (F (cartesian-product (sm-finals a-pda) (sm-finals a-dfa)))
       (non-EMP-pda-rls (filter (λ (r) (not (eq? (second (first r)) EMP))) pda-rls))
       (EMP-pda-rls (filter (λ (r) (eq? (second (first r)) EMP)) pda-rls))
       (non-EMP-rls (make-nonEMP-rls non-EMP-pda-rls dfa-sts dfa-rls))
       (EMP-rls (make-EMP-rls EMP-pda-rls dfa-sts))
       (ss-tbl (make-ss-table K))
       (ss->state (λ (ss) (second (assoc ss ss-tbl))))]
```

Properties of CFLs

```
;; pda dfa → pda  Purpose: Build intersection pda from given machines
(define (pda-intersect-dfa a-pda a-dfa #:rejects [rejs '()] #:accepts [accs '()])
```

```
(let* [(pda-sts (sm-states a-pda))  (dfa-sts (sm-states a-dfa))
       (pda-rls (sm-rules a-pda))   (dfa-rls (sm-rules a-dfa))
       (K (cartesian-product pda-sts dfa-sts))
       (start (list (sm-start a-pda) (sm-start a-dfa)))
       (F (cartesian-product (sm-finals a-pda) (sm-finals a-dfa)))
       (non-EMP-pda-rls (filter (λ (r) (not (eq? (second (first r)) EMP))) pda-rls))
       (EMP-pda-rls (filter (λ (r) (eq? (second (first r)) EMP)) pda-rls))
       (non-EMP-rls (make-nonEMP-rls non-EMP-pda-rls dfa-sts dfa-rls))
       (EMP-rls (make-EMP-rls EMP-pda-rls dfa-sts))
       (ss-tbl (make-ss-table K))
       (ss->state (λ (ss) (second (assoc ss ss-tbl))))]
  (make-ndpda (map ss->state K)  (sm-sigma a-pda)  (sm-gamma a-pda)
              (ss->state start)  (map ss->state F)
              (convert-ssrules (append non-EMP-rls EMP-rls) ss->state)
              #:rejects rejs
              #:accepts accs)))
```

Properties of CFLs

```
;; pda dfa → pda  Purpose: Build intersection pda from given machines
(define (pda-intersect-dfa a-pda a-dfa #:rejects [rejs '()] #:accepts [accs '()])
  ;; (listof state) (listof state) → (listof ss)
  ;; Purpose: Create a list of super states
  (define (cartesian-product pda-sts dfa-sts) ...)
  ;; (listof pda-rule) (listof state) → (listof ssrule)
  ;; Purpose: Create ssrules for given empty transition pda-rules
  (define (make-EMP-rls EMP-pda-rls dfa-sts) ...)
  ;; (listof pda-rule) (listof state) (listof dfa-rls) → (listof ssrule)
  ;; Purpose: Create ssrules for given nonempty transition pda-rules
  (define (make-nonEMP-rls non-EMP-pda-rls dfa-sts dfa-rls) ...)
  ;; (listof ss) → (listof (list ss state))
  ;; Purpose: Create table associating given super states with a new state
  (define (make-ss-table K) ...)
  ;; (listof ssrule) (ss → state) → (listof pda-rule)
  ;; Purpose: Convert ssrules to pda-rules
  (define (convert-ssrules ss-rls ss->state) ...)
 (let* [(pda-sts (sm-states a-pda))  (dfa-sts (sm-states a-dfa))
        (pda-rls (sm-rules a-pda))   (dfa-rls (sm-rules a-dfa))
        (K (cartesian-product pda-sts dfa-sts))
        (start (list (sm-start a-pda) (sm-start a-dfa)))
        (F (cartesian-product (sm-finals a-pda) (sm-finals a-dfa)))
        (non-EMP-pda-rls (filter (λ (r) (not (eq? (second (first r)) EMP))) pda-rls))
        (EMP-pda-rls (filter (λ (r) (eq? (second (first r)) EMP)) pda-rls))
        (non-EMP-rls (make-nonEMP-rls non-EMP-pda-rls dfa-sts dfa-rls))
        (EMP-rls (make-EMP-rls EMP-pda-rls dfa-sts))
        (ss-tbl (make-ss-table K))
        (ss->state (λ (ss) (second (assoc ss ss-tbl))))]
   (make-ndpda (map ss->state K)  (sm-sigma a-pda)  (sm-gamma a-pda)
               (ss->state start)  (map ss->state F)
               (convert-ssrules (append non-EMP-rls EMP-rls) ss->state)
               #:rejects rejs
               #:accepts accs)))
```

Properties of CFLs

- ```
  ;; (listof state) (listof state) → (listof ss)
  ;; Purpose: Create a list of super states
  (define (cartesian-product pda-sts dfa-sts)
    (for*/list [(s1 pda-sts) (s2 dfa-sts)] (list s1 s2)))
  ```
  ```

Properties of CFLs

- ```
  ;; (listof state) (listof state) → (listof ss)
  ;; Purpose: Create a list of super states
  (define (cartesian-product pda-sts dfa-sts)
    (for*/list [(s1 pda-sts) (s2 dfa-sts)] (list s1 s2)))
  ```
- ```
  (define (make-EMP-rls EMP-pda-rls dfa-sts)
    (for*/list [(r EMP-pda-rls) (s dfa-sts)]
      (list (list (list (first (first r)) s)
                  EMP
                  (third (first r)))
            (list (list (first (second r)) s)
                  (second (second r))))))
  ```

Properties of CFLs

- ```
  ;; (listof state) (listof state) → (listof ss)
  ;; Purpose: Create a list of super states
  (define (cartesian-product pda-sts dfa-sts)
    (for*/list [(s1 pda-sts) (s2 dfa-sts)] (list s1 s2)))
  ```
- ```
  (define (make-EMP-rls EMP-pda-rls dfa-sts)
    (for*/list [(r EMP-pda-rls) (s dfa-sts)]
      (list (list (list (first (first r)) s)
                  EMP
                  (third (first r)))
            (list (list (first (second r)) s)
                  (second (second r))))))
  ```
- ```
  (define (make-nonEMP-rls non-EMP-pda-rls dfa-sts dfa-rls)
    (for*/list [(r non-EMP-pda-rls) (s dfa-sts)]
      (list (list
              (list (first (first r)) s)
              (second (first r))
              (third (first r)))
            (list
              (list
                (first (second r))
                (third
                (first
                  (filter (λ (rl)
                            (and (eq? (first rl) s)
                                 (eq? (second rl)
                                      (second (first r)))))
                          dfa-rls))))
              (second (second r))))))
  ```
  ```

Properties of CFLs

- ```
  (define (make-ss-table K)
    (foldl (λ (ss acc)
              (cons (list ss (gen-state (map second acc))) acc))
           '()
           K))

  (define (convert-ssrules ss-rls ss->state)
   (for*/list [(r ss-rls)]
    (list
     (list (ss->state (first (first r)))
           (second (first r))
           (third (first r)))
     (list (ss->state (first (second r)))
           (second (second r))))))
  ```
  ```

Properties of CFLs

- ```
  (define (make-ss-table K)
    (foldl (λ (ss acc)
              (cons (list ss (gen-state (map second acc))) acc))
           '()
           K))

  (define (convert-ssrules ss-rls ss->state)
   (for*/list [(r ss-rls)]
    (list
     (list (ss->state (first (first r)))
           (second (first r))
           (third (first r)))
     (list (ss->state (first (second r)))
           (second (second r))))))
  ```
- ```
  ;; Tests for pda-intersect-dfa
  (define a^nb^nIM (pda-intersect-dfa a^nb^n
                                      ab*
                                      #:rejects '(() (a b b) (a a b b))
                                      #:accepts '((a b)))) only word in the intersection
  (define a^nb^nINO-ABBA (pda-intersect-dfa a^nb^n
                                            NO-ABAA
                                            #:rejects '((b b) (a a b) (b b a a b b a b))
                                            #:accepts '(() (a a b b))))
  ```

Properties of CFLs

Theorem

The intersection of a context-free language and a regular language is context-free.

- Proof (sketch)

Properties of CFLs

Theorem

The intersection of a context-free language and a regular language is context-free.

- Proof (sketch)
- Let Q be a `pda`, let D be a `dfa`, and let M = (pda-intersect-dfa Q D)

Properties of CFLs

Theorem

The intersection of a context-free language and a regular language is context-free.

- Proof (sketch)
- Let Q be a `pda`, let D be a `dfa`, and let M = (pda-intersect-dfa Q D)
- By construction, M simulates the transitions of both Q and D
- M starts its simulations in a state that represents Q's and D's starting states
- Only accepts if both Q and D accept
- This means that all accepted strings must be in L(Q) and L(D)

Properties of CFLs

Theorem

The intersection of a context-free language and a regular language is context-free.

- Proof (sketch)
- Let Q be a `pda`, let D be a `dfa`, and let M = (pda-intersect-dfa Q D)
- By construction, M simulates the transitions of both Q and D
- M starts its simulations in a state that represents Q's and D's starting states
- Only accepts if both Q and D accept
- This means that all accepted strings must be in L(Q) and L(D)
- $L(M) = L(Q) \cap L(D)$. $\square$

Properties of CFLs

Theorem

$L = \{ w \in \{a\ b\}^* \mid$ w does not contain abaa and has equal number of as and bs.$\}$ is context-free.

- The previous theorem may be used to prove a language, L, is context-free

Properties of CFLs

Theorem

$L = \{ w \in \{a\ b\}^* \mid$ w does not contain abaa and has equal number of as and bs.$\}$ is context-free.

- The previous theorem may be used to prove a language, L, is context-free
- The language of all words that do not contain abaa is decided by NO-ABAA (from dfa chapter)
- L(NO-ABAA) is regular

Properties of CFLs

Theorem

$L = \{ w \in \{a\ b\}^* \mid$ w does not contain abaa and has equal number of as and bs.$\}$ is context-free.

- The previous theorem may be used to prove a language, L, is context-free
- The language of all words that do not contain abaa is decided by NO-ABAA (from dfa chapter)
- L(NO-ABAA) is regular
- The language, E, of all words that have an equal number of as and bs is generated by the following context-free grammar:

$$\begin{aligned} \texttt{S} &\rightarrow \texttt{EMP} \\ &\rightarrow \texttt{aSb} \\ &\rightarrow \texttt{bSa} \\ &\rightarrow \texttt{aSbSbSa} \\ &\rightarrow \texttt{bSaSaSb} \end{aligned}$$

- E is context-free

Properties of CFLs

Theorem

$L = \{ w \in \{a\ b\}^* \mid$ w does not contain abaa and has equal number of as and bs.$\}$ is context-free.

- The previous theorem may be used to prove a language, L, is context-free
- The language of all words that do not contain abaa is decided by NO-ABAA (from dfa chapter)
- L(NO-ABAA) is regular
- The language, E, of all words that have an equal number of as and bs is generated by the following context-free grammar:

$$\begin{aligned} \texttt{S} &\rightarrow \texttt{EMP} \\ &\rightarrow \texttt{aSb} \\ &\rightarrow \texttt{bSa} \\ &\rightarrow \texttt{aSbSbSa} \\ &\rightarrow \texttt{bSaSaSb} \end{aligned}$$

- E is context-free
- Observe that L = L(E) ∩ L(NO-ABAA)
- By previous theorem, we may conclude that L is context-free. □

Properties of CFLs

- HOMEWORK: 12–14

Deterministic pdas

- Now, we explore deterministic pdas in more detail

Deterministic pdas

- Now, we explore deterministic pdas in more detail
- From a given configuration, C_i, there is at most one transition that is applicable
- The pda never has a choice of what to do next

Deterministic pdas

- Now, we explore deterministic pdas in more detail
- From a given configuration, C_i, there is at most one transition that is applicable
- The pda never has a choice of what to do next
- Such a pda must satisfy the following constraints:
  1. There are no pair of transitions that offer a choice on which to use
  2. For all accepting states, Q, there are no transitions of this nature: `((Q EMP EMP) (P a))`. That is, the machine does not have to choose between continuing with the computation or accepting.

Deterministic pdas

- Now, we explore deterministic pdas in more detail
- From a given configuration, C_i, there is at most one transition that is applicable
- The pda never has a choice of what to do next
- Such a pda must satisfy the following constraints:
  1. There are no pair of transitions that offer a choice on which to use
  2. For all accepting states, Q, there are no transitions of this nature: `((Q EMP EMP) (P a))`. That is, the machine does not have to choose between continuing with the computation or accepting.
- Given a nondeterministic pda, can a deterministic pda that decides the same language be constructed?

Deterministic pdas

$L = wcw^R$

- The nondeterministic pda's we previously designed may be summarized as follows:
  1. Nondeterministically move from S, the starting state, to P
  2. In P, accumulate w on the stack in reversed order and move to Q upon reading a c without modifying the stack.
  3. In Q, match the elements of w^R with the elements on the stack and nondeterministically move to, F, the final state.

Deterministic pdas

$L = wcw^R$

- The nondeterministic pda's we previously designed may be summarized as follows:
  1. Nondeterministically move from S, the starting state, to P
  2. In P, accumulate w on the stack in reversed order and move to Q upon reading a c without modifying the stack.
  3. In Q, match the elements of w^R with the elements on the stack and nondeterministically move to, F, the final state.
- A deterministic pda for L must avoid nondeterministic transitions
- Observe that the nondeterministic pda operates in two phases: before reading c, it pushes w onto the stack and after reading c, it matches w^R with the elements on the stack
- The phases are clearly delineated around reading c

Deterministic pdas

$L = wcw^R$

- The nondeterministic pda's we previously designed may be summarized as follows:
  1. Nondeterministically move from S, the starting state, to P
  2. In P, accumulate w on the stack in reversed order and move to Q upon reading a c without modifying the stack.
  3. In Q, match the elements of w^R with the elements on the stack and nondeterministically move to, F, the final state.
- A deterministic pda for L must avoid nondeterministic transitions
- Observe that the nondeterministic pda operates in two phases: before reading c, it pushes w onto the stack and after reading c, it matches w^R with the elements on the stack
- The phases are clearly delineated around reading c
- A deterministic pda may decide L as follows:
  1. Before reading c, push read elements onto the stack
  2. After reading c, match read elements with elements onto the stack

Deterministic pdas

$L = wcw^R$

- Name is `dwcw^r`, input alphabet is {`a b c`}, and the stack alphabet is {`a b`}.

Deterministic pdas

$L = wcw^R$

- Name is `dwcw^r`, input alphabet is {`a b c`}, and the stack alphabet is {`a b`}.
- Sample tests:

```
(check-reject? dwcw^r '() '(a b b a) '(a a) '(a b b c b b a b))
(check-accept? dwcw^r '(a c a) '(b a c a b) '(b b a c a b b))
```

Deterministic pdas

$L = wcw^R$

- S: ci = {a b}* AND stack = ci^R, starting state

Deterministic pdas

$L = wcw^R$

- S: ci = $\{a\ b\}^*$ AND stack = ci^R, starting state
- F: ci = $xycy^R$ AND stack = x^R, final state

Deterministic pdas

L = wcw^R

- S: ci = $\{a\ b\}^*$ AND stack = ci^R, starting state
- F: ci = $xycy^R$ AND stack = x^R, final state
- ((S a ,EMP) (S (a))) ((S b ,EMP) (S (b))) ((S c ,EMP) (F ,EMP))

Deterministic pdas

L = wcw^R

- S: ci = $\{a\ b\}^*$ AND stack = ci^R, starting state
- F: ci = $xycy^R$ AND stack = x^R, final state
- ((S a ,EMP) (S (a))) ((S b ,EMP) (S (b))) ((S c ,EMP) (F ,EMP))
- ((F a (a)) (F ,EMP)) ((F b (b)) (F ,EMP))
- There are no nondeterministic choices that the machine must make

Deterministic pdas

$L = wcw^R$

- S: ci = {a b}* AND stack = ci^R, starting state
- F: ci = xycy^R AND stack = x^R, final state
- ((S a ,EMP) (S (a))) ((S b ,EMP) (S (b))) ((S c ,EMP) (F ,EMP))
- ((F a (a)) (F ,EMP)) ((F b (b)) (F ,EMP))
- There are no nondeterministic choices that the machine must make
-
```
;; L = wcw^R
;; State Documentation
;;  S: ci = a b* AND stack = ci^R
;;  F: ci = xycy^R AND stack = x^R
(define dwcw^r (make-ndpda '(S F)
                           '(a b c)
                           '(a b)
                           'S
                           '(F)
                           `(((S a ,EMP) (S (a)))
                             ((S b ,EMP) (S (b)))
                             ((S c ,EMP) (F ,EMP))
                             ((F a (a)) (F ,EMP))
                             ((F b (b)) (F ,EMP)))))

(check-reject? dwcw^r '() '(a b b a) '(a a) '(a b b c b b a b))
(check-accept? dwcw^r '(a c a) '(b a c a b) '(b b a c a b b))
```

Deterministic pdas

$L = wcw^R$

```
;; word stack → Boolean
(define (S-INV ci s)
  (and (andmap (λ (s) (or (eq? s 'a) (eq? s 'b))) ci)
       (equal? ci (reverse s))))

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

Deterministic pdas

$L = wcw^R$

- ```
;; word stack → Boolean
(define (S-INV ci s)
  (and (andmap (λ (s) (or (eq? s 'a) (eq? s 'b))) ci)
       (equal? ci (reverse s))))

(check-equal? (S-INV '(c) '()) #f)
(check-equal? (S-INV '(b a) '(b a)) #f)
(check-equal? (S-INV '() '()) #t)
(check-equal? (S-INV '(a b b) '(b b a)) #t)
```
- ```
;; word stack → Boolean
(define (F-INV ci s)
  (and (member 'c ci)
       (let* [(w (takef ci (λ (x) (not (eq? x 'c)))))
              (x (take ci (length s)))
              (y^R (drop ci (add1 (length w))))]
         (and (equal?
                ci
                (append x (reverse y^R) (list 'c) y^R))
              (equal? s (reverse x))))))

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

Deterministic pdas

`L = wcw^R` `M = dwcw^r` `v ∈ (sm-sigma M)*`
`s = the stack` `ci = consumed input`

Theorem

The state invariants hold when M accepts v.

- For the base case, when `M` starts ci = `'()` and s = `'()`. This means that ci only contains as and bs and that ci equals s reversed. Thus, S-INV holds.

Deterministic pdas

`L = wcw`^{R} `M = dwcw^r` `v` $\in$ `(sm-sigma M)`*

`s = the stack` `ci = consumed input`

Theorem

The state invariants hold when M accepts v.

- For the base case, when `M` starts ci $=$ `'()` and s $=$ `'()`. This means that ci only contains as and bs and that ci equals s reversed. Thus, S-INV holds.
- Proof invariants hold after each transition that consumes input:
- `((S a ,EMP) (S (a)))`: By inductive hypothesis S-INV holds. This means that ci$\in${a b}* and that ci $=$ s^R. After using this rule, the consumed input only contains as and bs and the consumed input is equal to the stack reversed. Thus, S-INV holds.

Deterministic pdas

`L = wcw`^{R} `M = dwcw^r` `v` $\in$ `(sm-sigma M)`*

`s = the stack` `ci = consumed input`

Theorem

The state invariants hold when M accepts v.

- For the base case, when `M` starts ci $=$ `'()` and s $=$ `'()`. This means that ci only contains as and bs and that ci equals s reversed. Thus, S-INV holds.
- Proof invariants hold after each transition that consumes input:
- `((S a ,EMP) (S (a)))`: By inductive hypothesis S-INV holds. This means that ci$\in${a b}* and that ci $=$ s^R. After using this rule, the consumed input only contains as and bs and the consumed input is equal to the stack reversed. Thus, S-INV holds.
- `((S b ,EMP) (S (b)))`: By inductive hypothesis S-INV holds. This means that ci$\in${a b}* and that ci $=$ s^R. After using this rule, the consumed input only contains as and bs and the consumed input is equal to the stack reversed. Thus, S-INV holds.

Deterministic pdas

`L = wcw`^{R} `M = dwcw^r` `v` $\in$ `(sm-sigma M)`*

`s = the stack` `ci = consumed input`

Theorem

The state invariants hold when M accepts v.

- For the base case, when `M` starts ci = `'()` and s = `'()`. This means that ci only contains as and bs and that ci equals s reversed. Thus, S-INV holds.
- Proof invariants hold after each transition that consumes input:
- `((S a ,EMP) (S (a)))`: By inductive hypothesis S-INV holds. This means that ci$\in${a b}* and that ci = s^R. After using this rule, the consumed input only contains as and bs and the consumed input is equal to the stack reversed. Thus, S-INV holds.
- `((S b ,EMP) (S (b)))`: By inductive hypothesis S-INV holds. This means that ci$\in${a b}* and that ci = s^R. After using this rule, the consumed input only contains as and bs and the consumed input is equal to the stack reversed. Thus, S-INV holds.
- `((S c ,EMP) (F ,EMP))`: By inductive hypothesis S-INV holds. This means that ci$\in${a b}* and that ci = s^R. After using this rule, ci=xycy^R, where x$\in${a b}* and y=`'()`, and stack=x^R. Thus, F-INV holds.

Deterministic pdas

`L = wcw`^{R} `M = dwcw^r` `v` $\in$ `(sm-sigma M)`*

`s = the stack` `ci = consumed input`

Theorem

The state invariants hold when M accepts v.

- For the base case, when `M` starts ci = `'()` and s = `'()`. This means that ci only contains as and bs and that ci equals s reversed. Thus, S-INV holds.
- Proof invariants hold after each transition that consumes input:
- `((S a ,EMP) (S (a)))`: By inductive hypothesis S-INV holds. This means that ci$\in${a b}* and that ci = s^R. After using this rule, the consumed input only contains as and bs and the consumed input is equal to the stack reversed. Thus, S-INV holds.
- `((S b ,EMP) (S (b)))`: By inductive hypothesis S-INV holds. This means that ci$\in${a b}* and that ci = s^R. After using this rule, the consumed input only contains as and bs and the consumed input is equal to the stack reversed. Thus, S-INV holds.
- `((S c ,EMP) (F ,EMP))`: By inductive hypothesis S-INV holds. This means that ci$\in${a b}* and that ci = s^R. After using this rule, ci=xycy^R, where x$\in${a b}* and y=`'()`, and stack=x^R. Thus, F-INV holds.
- `((F a (a)) (F ,EMP))`: By inductive hypothesis F-INV holds. This means that ci=xycy^R, where x,y$\in${a b}*, and stack=x^R. Using this rule means that x's last element is an a. That is, x=x`'`a. After using this rule, ci=x`'`aycy^{R}a=x`'`ayc(ay)^R and stack=x`'`^{R}. Thus, F-INV holds.

Deterministic pdas

`L = wcw^R` `M = dwcw^r` `v ∈ (sm-sigma M)*`
`s = the stack` `ci = consumed input`

Theorem

The state invariants hold when M accepts v.

- For the base case, when `M` starts ci = `'()` and s = `'()`. This means that ci only contains as and bs and that ci equals s reversed. Thus, S-INV holds.
- Proof invariants hold after each transition that consumes input:
- `((S a ,EMP) (S (a)))`: By inductive hypothesis S-INV holds. This means that ci$\in$`{a b}`* and that ci = s^R. After using this rule, the consumed input only contains as and bs and the consumed input is equal to the stack reversed. Thus, S-INV holds.
- `((S b ,EMP) (S (b)))`: By inductive hypothesis S-INV holds. This means that ci$\in$`{a b}`* and that ci = s^R. After using this rule, the consumed input only contains as and bs and the consumed input is equal to the stack reversed. Thus, S-INV holds.
- `((S c ,EMP) (F ,EMP))`: By inductive hypothesis S-INV holds. This means that ci$\in$`{a b}`* and that ci = s^R. After using this rule, ci=xycy^R, where x$\in$`{a b}`* and y=`'()`, and stack=x^R. Thus, F-INV holds.
- `((F a (a)) (F ,EMP))`: By inductive hypothesis F-INV holds. This means that ci=xycy^R, where x,y$\in$`{a b}`*, and stack=x^R. Using this rule means that x's last element is an a. That is, x=x`'a`. After using this rule, ci=x`'a`ycy^{R}a=x`'a`yc(ay)^R and stack=x`'`^{R}. Thus, F-INV holds.
- `((F b (b)) (F ,EMP))`: By inductive hypothesis F-INV holds. This means that ci=xycy^R, where x,y$\in$`{a b}`*, and stack=x^R. Using this rule means that x's last element is a b. That is, x=x`'b`. After using this rule, ci=x`'b`ycy^{R}b= x`'b`yc(by)^R and stack=x`'`^{R}. Thus, F-INV holds

Deterministic pdas

$L = wcw^R$

Lemma

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

-

Deterministic pdas

$L = wcw^R$

Lemma

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

-

Proof.

($\Rightarrow$) Assume w∈L. This means that $w = vcv^R$. Given that state invariants always hold, there is a computation that has M consume w and reach F with an empty stack. Therefore, w∈L(M).
($\Leftarrow$) Assume w∈L(M). This means that M halts in F, the only final state, with an empty stack having consumed w. Given that the state invariants always hold we may conclude that $w = vcv^R$. Therefore, w∈L. □ □

-

Deterministic pdas

`L = wcw^R`

Lemma

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

-

Proof.

($\Rightarrow$) Assume w∈L. This means that w = vcv^R. Given that state invariants always hold, there is a computation that has M consume w and reach F with an empty stack. Therefore, w∈L(M).
($\Leftarrow$) Assume w∈L(M). This means that M halts in F, the only final state, with an empty stack having consumed w. Given that the state invariants always hold we may conclude that w = vcv^R. Therefore, w∈L.
□ □

-

Lemma

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

-

Deterministic pdas

`L = wcw^R`

Lemma

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

-

Proof.

($\Rightarrow$) Assume w∈L. This means that w = vcv^R. Given that state invariants always hold, there is a computation that has M consume w and reach F with an empty stack. Therefore, w∈L(M).
($\Leftarrow$) Assume w∈L(M). This means that M halts in F, the only final state, with an empty stack having consumed w. Given that the state invariants always hold we may conclude that w = vcv^R. Therefore, w∈L. □ □

-

Lemma

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

-

Proof.

($\Rightarrow$) Assume w∉L. This means w ≠ v^{R}cv. Given that the state invariant predicates always hold, there is no computation that has M consume w and end in F with an empty stack. Therefore, w∉L(M).
($\Leftarrow$) Assume w∉L(M). This means that M does not halt in F with an empty stack having consumed w. Given that the state invariants always hold, this means that w ≠ vcv^R. Thus, w∉L. □ □

-

Deterministic pdas

$L = \{a^m b^n c^p \mid m \neq n \wedge m,n,p > 0\}$

- Intuitively: push all the as onto the stack, match bs, either pop excess as or read excess bs, and read cs
- Challenge: deterministically determine if excess as need to be popped or excess bs need to be read

Deterministic pdas

$L = \{a^m b^n c^p \mid m \neq n \wedge m,n,p > 0\}$

- Intuitively: push all the as onto the stack, match bs, either pop excess as or read excess bs, and read cs
- Challenge: deterministically determine if excess as need to be popped or excess bs need to be read
- If there is an excess of bs in the input word then all the as on the stack will be matched and the stack becomes empty
- This is a problem because the machine cannot nondeterministically decide to keep matching as on the stack or read excess bs

Deterministic pdas

$L = \{a^m b^n c^p \mid m \neq n \wedge m,n,p > 0\}$

- Intuitively: push all the as onto the stack, match bs, either pop excess as or read excess bs, and read cs
- Challenge: deterministically determine if excess as need to be popped or excess bs need to be read
- If there is an excess of bs in the input word then all the as on the stack will be matched and the stack becomes empty
- This is a problem because the machine cannot nondeterministically decide to keep matching as on the stack or read excess bs
- If the machine could sense that the stack is empty then the decision becomes deterministic
- If the stack is empty read excess bs
- Otherwise, match as as bs are read

Deterministic pdas

$L = \{a^m b^n c^p \mid m \neq n \wedge m,n,p > 0\}$

- Intuitively: push all the as onto the stack, match bs, either pop excess as or read excess bs, and read cs
- Challenge: deterministically determine if excess as need to be popped or excess bs need to be read
- If there is an excess of bs in the input word then all the as on the stack will be matched and the stack becomes empty
- This is a problem because the machine cannot nondeterministically decide to keep matching as on the stack or read excess bs
- If the machine could sense that the stack is empty then the decision becomes deterministic
- If the stack is empty read excess bs
- Otherwise, match as as bs are read
- Another potential problem is deterministically transitioning to the final state
- To prevent nondeterministic transitions of this nature, a special end of input symbol, z, not in the input alphabet of the machine, shall be used
- All input: wz

Deterministic pdas

$L = \{a^m b^n c^p \mid m \neq n \wedge m,n,p > 0\}$

- Intuitively: push all the as onto the stack, match bs, either pop excess as or read excess bs, and read cs
- Challenge: deterministically determine if excess as need to be popped or excess bs need to be read
- If there is an excess of bs in the input word then all the as on the stack will be matched and the stack becomes empty
- This is a problem because the machine cannot nondeterministically decide to keep matching as on the stack or read excess bs
- If the machine could sense that the stack is empty then the decision becomes deterministic
- If the stack is empty read excess bs
- Otherwise, match as as bs are read
- Another potential problem is deterministically transitioning to the final state
- To prevent nondeterministic transitions of this nature, a special end of input symbol, z, not in the input alphabet of the machine, shall be used
- All input: wz
- How can the machine sense that the stack is empty?

Deterministic pdas

$L = \{a^m b^n c^p \mid m \neq n \wedge m,n,p > 0\}$

- Intuitively: push all the as onto the stack, match bs, either pop excess as or read excess bs, and read cs
- Challenge: deterministically determine if excess as need to be popped or excess bs need to be read
- If there is an excess of bs in the input word then all the as on the stack will be matched and the stack becomes empty
- This is a problem because the machine cannot nondeterministically decide to keep matching as on the stack or read excess bs
- If the machine could sense that the stack is empty then the decision becomes deterministic
- If the stack is empty read excess bs
- Otherwise, match as as bs are read
- Another potential problem is deterministically transitioning to the final state
- To prevent nondeterministic transitions of this nature, a special end of input symbol, z, not in the input alphabet of the machine, shall be used
- All input: wz
- How can the machine sense that the stack is empty?
- Use a technique developed to construct simple pdas: the machine shall use a bottom of the stack symbol
- For our purposes, y serves as the bottom of the stack symbol.

Deterministic pdas

$L = \{a^m b^n c^p \mid m \neq n \wedge m,n,p > 0\}$

- Intuitively: push all the as onto the stack, match bs, either pop excess as or read excess bs, and read cs
- Challenge: deterministically determine if excess as need to be popped or excess bs need to be read
- If there is an excess of bs in the input word then all the as on the stack will be matched and the stack becomes empty
- This is a problem because the machine cannot nondeterministically decide to keep matching as on the stack or read excess bs
- If the machine could sense that the stack is empty then the decision becomes deterministic
- If the stack is empty read excess bs
- Otherwise, match as as bs are read
- Another potential problem is deterministically transitioning to the final state
- To prevent nondeterministic transitions of this nature, a special end of input symbol, z, not in the input alphabet of the machine, shall be used
- All input: wz
- How can the machine sense that the stack is empty?
- Use a technique developed to construct simple pdas: the machine shall use a bottom of the stack symbol
- For our purposes, y serves as the bottom of the stack symbol.

- The machine's phases may be outlined as follows:
  1. When started, if an a is read then (a y) is pushed onto the stack.
  2. All the remaining as read are pushed onto the stack.
  3. Match a read b by popping an a
  4. Decide if there are more as or more bs:
     1. If a c is read and there is an a on the stack then there are more as than bs. The machine transitions to a state to read the remaining cs and pop the remaining as. Upon reading z, the machine pops y and moves to the final state.
     2. If a b is read and y is at the top of the stack then there are morebs than as. The machine transitions to read the bs with changing the stack. It then transitions to read the cs without altering the stack. Upon reading z, the machine pops y and moves to the final state

Deterministic pdas

$L = \{a^m b^n c^p \mid m \neq n \wedge m,n,p > 0\}$

```
;; State Documentation
;;  S: ci = stack = '(), starting state
;;  A: ci = a^+ and stack = a^+y
;;  B: ci = a^ib^j and stack = a^(i-j)y and 0 < i <= j
;;  C: ci = a^ib^jc^k and stack = a^(i-j-k)y and i > j and i,j,k > 0
;;  D: ci = a^ib^j and stack = y and j > i and i,j > 0
;;  E: ci = a^ib^jc^+ and stack = y, j != i and i,j > 0
;;  G: ci = a^ib^jc^+z and stack = a*y and i != j and i,j > 0
;;  F: ci = a^ib^jc^+z and stack = '() and i != j, and i,j > 0, final state
```

Deterministic pdas

$L = \{a^m b^n c^p \mid m \neq n \wedge m,n,p > 0\}$

```
#lang fsm

(define ambnco (make-ndpda '(S A B C D E F G)
                           '(a b c z)
                           '(a y)
                           'S
                           '(F)
```

Deterministic pdas

$L = \{a^m b^n c^p \mid m \neq n \wedge m,n,p > 0\}$

- ```
  #lang fsm

  (define ambnco (make-ndpda '(S A B C D E F G)
                             '(a b c z)
                             '(a y)
                             'S
                             '(F)
  ```

- ```
  (check-reject? ambnco '(z) '(a b b c c) '(a a b b c c z) '(a b a z) '(a b b z))
  (check-accept? ambnco '(a a a b b c c z) '(a a a b c z) '(a a a b b b b c c c z))
  ```

Deterministic pdas

$L = \{a^m b^n c^p \mid m \neq n \wedge m,n,p > 0\}$

- ```
  #lang fsm

  (define ambnco (make-ndpda '(S A B C D E F G)
                             '(a b c z)
                             '(a y)
                             'S
                             '(F)
  ```
- ```
                             `(((S a ,EMP) (A (a y)))
  ```

- ```
  (check-reject? ambnco '(z) '(a b b c c) '(a a b b c c z) '(a b a z) '(a b b z))
  (check-accept? ambnco '(a a a b b c c z) '(a a a b c z) '(a a a b b b b c c c z))
  ```
  ```

Deterministic pdas

$L = \{a^m b^n c^p \mid m \neq n \wedge m,n,p > 0\}$

- ```
  #lang fsm

  (define ambnco (make-ndpda '(S A B C D E F G)
                             '(a b c z)
                             '(a y)
                             'S
                             '(F)
  ```
- ```
                             `(((S a ,EMP) (A (a y)))
  ```
- ```
                               ((A a ,EMP) (A (a)))
                               ((A b (a)) (B ,EMP))
  ```

- ```
  (check-reject? ambnco '(z) '(a b b c c) '(a a b b c c z) '(a b a z) '(a b b z))
  (check-accept? ambnco '(a a a b b c c z) '(a a a b c z) '(a a a b b b b c c c z))
  ```

Deterministic pdas

$L = \{a^m b^n c^p \mid m \neq n \wedge m,n,p > 0\}$

```
#lang fsm

(define ambnco (make-ndpda '(S A B C D E F G)
                           '(a b c z)
                           '(a y)
                           'S
                           '(F)
                           `(((S a ,EMP) (A (a y)))
                             ((A a ,EMP) (A (a)))
                             ((A b (a)) (B ,EMP))
                             ((B b (a)) (B ,EMP))
                             ((B c (a)) (C ,EMP))
                             ((B b (y)) (D (y)))
                             ((C c (a)) (C ,EMP))
                             ((C b (y)) (D (y)))
                             ((D b (y)) (D ,EMP))
                             ((D c (z)) (E (z)))
                             ((E c (z)) (E ,EMP))
                             ((E c (a)) (F (a)))
                             ((F c (a)) (F ,EMP))
                             ((F c (y)) (G (y)))
                             ((G c (y)) (G ,EMP))
                             ((G c (z)) (G ,EMP)))))

(check-reject? ambnco '(z) '(a b b c c) '(a a b b c c z) '(a b a z) '(a b b z))
(check-accept? ambnco '(a a a b b c c z) '(a a a b c z) '(a a a b b b b c c c z))
```

Deterministic pdas

$L = \{a^m b^n c^p \mid m \neq n \wedge m,n,p > 0\}$

```
#lang fsm

(define ambnco (make-ndpda '(S A B C D E F G)
                           '(a b c z)
                           '(a y)
                           'S
                           '(F)
                           `(((S a ,EMP) (A (a y)))
                             ((A a ,EMP) (A (a)))
                             ((A b (a)) (B ,EMP))
                             ((B b (a)) (B ,EMP))
                             ((B c (a)) (C ,EMP))
                             ((C c (a)) (C ,EMP))
                             ((C z ,EMP) (G ,EMP))
                             ((C c (y)) (E (y)))
                             ((B b (y)) (D (y)))
```

```
(check-reject? ambnco '(z) '(a b b c c) '(a a b b c c z) '(a b a z) '(a b b z))
(check-accept? ambnco '(a a a b b c c z) '(a a a b c z) '(a a a b b b b c c c z))
```

Deterministic pdas

$L = \{a^m b^n c^p \mid m \neq n \wedge m,n,p > 0\}$

```
#lang fsm

(define ambnco (make-ndpda '(S A B C D E F G)
                           '(a b c z)
                           '(a y)
                           'S
                           '(F)
                           `(((S a ,EMP) (A (a y)))
                             ((A a ,EMP) (A (a)))
                             ((A b (a)) (B ,EMP))
                             ((B b (a)) (B ,EMP))
                             ((B c (a)) (C ,EMP))
                             ((C c (a)) (C ,EMP))
                             ((C z ,EMP) (G ,EMP))
                             ((C c (y)) (E (y)))
                             ((B b (y)) (D (y)))
                             ((D b (y)) (D (y)))
                             ((D c (y)) (E (y)))
```

```
(check-reject? ambnco '(z) '(a b b c c) '(a a b b c c z) '(a b a z) '(a b b z))
(check-accept? ambnco '(a a a b b c c z) '(a a a b c z) '(a a a b b b b c c c z))
```

Deterministic pdas

$L = \{a^m b^n c^p \mid m \neq n \wedge m,n,p > 0\}$

```
#lang fsm

(define ambnco (make-ndpda '(S A B C D E F G)
                           '(a b c z)
                           '(a y)
                           'S
                           '(F)
                           `(((S a ,EMP) (A (a y)))
                             ((A a ,EMP) (A (a)))
                             ((A b (a)) (B ,EMP))
                             ((B b (a)) (B ,EMP))
                             ((B c (a)) (C ,EMP))
                             ((C c (a)) (C ,EMP))
                             ((C z ,EMP) (G ,EMP))
                             ((C c (y)) (E (y)))
                             ((B b (y)) (D (y)))
                             ((D b (y)) (D (y)))
                             ((D c (y)) (E (y)))
                             ((E c (y)) (E (y)))
                             ((E z (y)) (F ,EMP)))))

(check-reject? ambnco '(z) '(a b b c c) '(a a b b c c z) '(a b a z) '(a b b z))
(check-accept? ambnco '(a a a b b c c z) '(a a a b c z) '(a a a b b b b c c c z))
```

Deterministic pdas

$L = \{a^m b^n c^p \mid m \neq n \wedge m,n,p > 0\}$

```
#lang fsm

(define ambnco (make-ndpda '(S A B C D E F G)
                           '(a b c z)
                           '(a y)
                           'S
                           '(F)
                           `(((S a ,EMP) (A (a y)))
                             ((A a ,EMP) (A (a)))
                             ((A b (a)) (B ,EMP))
                             ((B b (a)) (B ,EMP))
                             ((B c (a)) (C ,EMP))
                             ((C c (a)) (C ,EMP))
                             ((C z ,EMP) (G ,EMP))
                             ((C c (y)) (E (y)))
                             ((B b (y)) (D (y)))
                             ((D b (y)) (D (y)))
                             ((D c (y)) (E (y)))
                             ((E c (y)) (E (y)))
                             ((E z (y)) (F ,EMP))
                             ((G ,EMP (a)) (G ,EMP))
                             ((G ,EMP (y)) (F ,EMP)))))
(check-reject? ambnco '(z) '(a b b c c) '(a a b b c c z) '(a b a z) '(a b b z))
(check-accept? ambnco '(a a a b b c c z) '(a a a b c z) '(a a a b b b b c c c z))
```

Deterministic pdas

L = $\{a^m b^n c^p \mid m \neq n \wedge m,n,p > 0\}$ ci = consumed input P = ambncp

Theorem

The state invariants hold when P is applied to w.

- When P starts, S-INV holds because ci = '() and the stack = '(). This establishes the base case.

Deterministic pdas

L = $\{a^m b^n c^p \mid m \neq n \wedge m,n,p > 0\}$ ci = consumed input P = ambncp

Theorem

The state invariants hold when P is applied to w.

- When P starts, S-INV holds because ci = '() and the stack = '(). This establishes the base case.
- Proof that invariants hold after each nonempty transition:
- ((S a ,EMP) (A (a y))): By inductive hypothesis, S-INV holds. This means ci = stack = '(). Using this transition adds an a to ci and pushes y and then a onto the stack. Therefore, after using this rule, ci = (a^+) and stack = $(a^+$ y). Thus, A-INV holds.

Deterministic pdas

L = $\{a^m b^n c^p \mid m \neq n \wedge m,n,p > 0\}$ ci = consumed input P = ambncp

Theorem

The state invariants hold when P is applied to w.

- When P starts, S-INV holds because ci = '() and the stack = '(). This establishes the base case.
- Proof that invariants hold after each nonempty transition:
- ((S a ,EMP) (A (a y))): By inductive hypothesis, S-INV holds. This means ci = stack = '(). Using this transition adds an a to ci and pushes y and then a onto the stack. Therefore, after using this rule, ci = (a^+) and stack = (a^+ y). Thus, A-INV holds.
- ((A a ,EMP) (A (a))): By inductive hypothesis, A-INV holds. This means ci = (a^+) and stack = (a^+ y). Using this transition adds an a to ci and pushes an a onto the stack. Therefore, after using this rule, ci = (a^+) and stack = (a^+ y). Thus, A-INV holds.

Deterministic pdas

L = $\{a^m b^n c^p \mid m \neq n \wedge m,n,p > 0\}$ ci = consumed input P = ambncp

Theorem

The state invariants hold when P is applied to w.

- When P starts, S-INV holds because ci = '() and the stack = '(). This establishes the base case.
- Proof that invariants hold after each nonempty transition:
- ((S a ,EMP) (A (a y))): By inductive hypothesis, S-INV holds. This means ci = stack = '(). Using this transition adds an a to ci and pushes y and then a onto the stack. Therefore, after using this rule, ci = (a^+) and stack = $(a^+$ y). Thus, A-INV holds.
- ((A a ,EMP) (A (a))): By inductive hypothesis, A-INV holds. This means ci = (a^+) and stack = $(a^+$ y). Using this transition adds an a to ci and pushes an a onto the stack. Therefore, after using this rule, ci = (a^+) and stack = $(a^+$ y). Thus, A-INV holds.
- ((A b (a)) (B ,EMP)): By inductive hypothesis, A-INV holds. This means ci = $(a^+) = (a^i)$ and stack = $(a^+$ y). Using this transition adds a b to ci and pops an a off the stack. Therefore, after using this rule, ci = $(a^i\ b^1) = (a^i\ b^j)$, stack = $(a^{i-1}$ y) = $(a^{i-j}$ y) and $0 < j \leq i$. Thus, B-INV holds.

Deterministic pdas

L = $\{a^m b^n c^p \mid m \neq n \wedge m,n,p > 0\}$ ci = consumed input P = ambncp

Theorem

The state invariants hold when P is applied to w.

- When P starts, S-INV holds because ci = '() and the stack = '(). This establishes the base case.
- Proof that invariants hold after each nonempty transition:
- ((S a ,EMP) (A (a y))): By inductive hypothesis, S-INV holds. This means ci = stack = '(). Using this transition adds an a to ci and pushes y and then a onto the stack. Therefore, after using this rule, ci = (a^+) and stack = $(a^+$ y). Thus, A-INV holds.
- ((A a ,EMP) (A (a))): By inductive hypothesis, A-INV holds. This means ci = (a^+) and stack = $(a^+$ y). Using this transition adds an a to ci and pushes an a onto the stack. Therefore, after using this rule, ci = (a^+) and stack = $(a^+$ y). Thus, A-INV holds.
- ((A b (a)) (B ,EMP)): By inductive hypothesis, A-INV holds. This means ci = $(a^+) = (a^i)$ and stack = $(a^+$ y). Using this transition adds a b to ci and pops an a off the stack. Therefore, after using this rule, ci = $(a^i\ b^1) = (a^i\ b^j)$, stack = $(a^{i-1}$ y) = $(a^{i-j}$ y) and $0 < j \leq i$. Thus, B-INV holds.
- ((B b (a)) (B ,EMP)): By inductive hypothesis, B-INV holds. This means ci = $(a^i\ b^j)$, stack = $(a^{i-j}$ y) and $0 < j \leq i$. Given that there is an a on the stack, j must be strictly less than i. That is, $j < i$. Using this transition adds a b to ci and pops an a off the stack. Therefore, after using this rule, ci = $(a^i\ b^{j+1})$, stack = $(a^{i-(j+1)}$ y), $0 < j+1$, and $j+1 \leq i$. Without loss of generality, we may think of j+1 as a new value for j. Thus, B-INV holds.

Deterministic pdas

L = $\{a^m b^n c^p \mid m \neq n \wedge m,n,p > 0\}$ ci = consumed input P = ambncp

Theorem

The state invariants hold when P is applied to w.

- When P starts, S-INV holds because ci = '() and the stack = '(). This establishes the base case.
- Proof that invariants hold after each nonempty transition:
- ((S a ,EMP) (A (a y))): By inductive hypothesis, S-INV holds. This means ci = stack = '(). Using this transition adds an a to ci and pushes y and then a onto the stack. Therefore, after using this rule, ci = (a^+) and stack = $(a^+$ y). Thus, A-INV holds.
- ((A a ,EMP) (A (a))): By inductive hypothesis, A-INV holds. This means ci = (a^+) and stack = $(a^+$ y). Using this transition adds an a to ci and pushes an a onto the stack. Therefore, after using this rule, ci = (a^+) and stack = $(a^+$ y). Thus, A-INV holds.
- ((A b (a)) (B ,EMP)): By inductive hypothesis, A-INV holds. This means ci = $(a^+) = (a^i)$ and stack = $(a^+$ y). Using this transition adds a b to ci and pops an a off the stack. Therefore, after using this rule, ci = $(a^i\ b^1) = (a^i\ b^j)$, stack = $(a^{i-1}$ y) = $(a^{i-j}$ y) and $0 < j \leq i$. Thus, B-INV holds.
- ((B b (a)) (B ,EMP)): By inductive hypothesis, B-INV holds. This means ci = $(a^i\ b^j)$, stack = $(a^{i-j}$ y) and $0 < j \leq i$. Given that there is an a on the stack, j must be strictly less than i. That is, $j < i$. Using this transition adds a b to ci and pops an a off the stack. Therefore, after using this rule, ci = $(a^i\ b^{j+1})$, stack = $(a^{i-(j+1)}$ y), $0 < j+1$, and $j+1 \leq i$. Without loss of generality, we may think of j+1 as a new value for j. Thus, B-INV holds.
- ((B c (a)) (C ,EMP)): By inductive hypothesis, B-INV holds. This means ci = $(a^i\ b^j)$, stack = $(a^{i-j}$ y), and $0 < j \leq i$. Given that there is an a on the stack, $j < i$. Using this transition , adds a c to ci and pops an a. Therefore, after using this rule, ci = $(a^i\ b^j\ c^1) = (a^i\ b^j\ c^k)$, stack = $(a^{i-j-1}$ y) = $(a^{i-j-k}$ y), $i > j$, and i,j,k > 0. Thus, C-INV holds.

Deterministic pdas

L = $\{a^m b^n c^p \mid m \neq n \wedge m,n,p > 0\}$ ci = consumed input P = ambncp

Theorem

The state invariants hold when P is applied to w.

- When P starts, S-INV holds because ci = '() and the stack = '(). This establishes the base case.
- Proof that invariants hold after each nonempty transition:
- ((S a ,EMP) (A (a y))): By inductive hypothesis, S-INV holds. This means ci = stack = '(). Using this transition adds an a to ci and pushes y and then a onto the stack. Therefore, after using this rule, ci = (a^+) and stack = $(a^+$ y). Thus, A-INV holds.
- ((A a ,EMP) (A (a))): By inductive hypothesis, A-INV holds. This means ci = (a^+) and stack = $(a^+$ y). Using this transition adds an a to ci and pushes an a onto the stack. Therefore, after using this rule, ci = (a^+) and stack = $(a^+$ y). Thus, A-INV holds.
- ((A b (a)) (B ,EMP)): By inductive hypothesis, A-INV holds. This means ci = $(a^+) = (a^i)$ and stack = $(a^+$ y). Using this transition adds a b to ci and pops an a off the stack. Therefore, after using this rule, ci = $(a^i\ b^1) = (a^i\ b^j)$, stack = $(a^{i-1}$ y) = $(a^{i-j}$ y) and $0 < j \leq i$. Thus, B-INV holds.
- ((B b (a)) (B ,EMP)): By inductive hypothesis, B-INV holds. This means ci = $(a^i\ b^j)$, stack = $(a^{i-j}$ y) and $0 < j \leq i$. Given that there is an a on the stack, j must be strictly less than i. That is, $j < i$. Using this transition adds a b to ci and pops an a off the stack. Therefore, after using this rule, ci = $(a^i\ b^{j+1})$, stack = $(a^{i-(j+1)}$ y), $0 < j+1$, and $j+1 \leq i$. Without loss of generality, we may think of j+1 as a new value for j. Thus, B-INV holds.
- ((B c (a)) (C ,EMP)): By inductive hypothesis, B-INV holds. This means ci = $(a^i\ b^j)$, stack = $(a^{i-j}$ y), and $0 < j \leq i$. Given that there is an a on the stack, $j < i$. Using this transition , adds a c to ci and pops an a. Therefore, after using this rule, ci = $(a^i\ b^j\ c^1) = (a^i\ b^j\ c^k)$, stack = $(a^{i-j-1}$ y) = $(a^{i-j-k}$ y), $i > j$, and i,j,k > 0. Thus, C-INV holds.
- ((B b (y)) (D (y))): By inductive hypothesis, B-INV holds. This means ci = $(a^i\ b^j)$, stack = $(a^{i-j}$ y), and $0 < j \leq i$. Observe that if the stack does not contain as then j = i. Using this transition adds a b to ci without modifying the stack. Therefore, ci = $(a^i\ b^{j+1})$ and stack = (y) and $j+1 > i$ and $i,j+1 > 0$. Without loss of generality, we may think of j+1 as a new value for j. Therefore, D-INV holds.

Deterministic pdas

L = $\{a^m b^n c^p \mid m \neq n \wedge m,n,p > 0\}$

- ((C c (a)) (C ,EMP)): By inductive hypothesis, C-INV holds. This means ci = (a^i b^j c^k), stack = (a^{i-j-k} y), i > j, and i,j,k > 0. Using this transition adds a c to ci and pops an a from the stack. Therefore, ci = (a^i b^j c^{k+1}), stack = ($a^{i-j-(k+1)}$ y), i > j, and i,j,k+1 > 0. Without loss of generality, we may think of k+1 as a new value for k. Therefore, C-INV holds.

Deterministic pdas

L = $\{a^m b^n c^p \mid m \neq n \wedge m,n,p > 0\}$

- ((C c (a)) (C ,EMP)): By inductive hypothesis, C-INV holds. This means ci = (a^i b^j c^k), stack = (a^{i-j-k} y), i > j, and i,j,k > 0. Using this transition adds a c to ci and pops an a from the stack. Therefore, ci = (a^i b^j c^{k+1}), stack = ($a^{i-j-(k+1)}$ y), i > j, and i,j,k+1 > 0. Without loss of generality, we may think of k+1 as a new value for k. Therefore, C-INV holds.
- ((C z ,EMP) (G ,EMP)): By inductive hypothesis, C-INV holds. This means ci = (a^i b^j c^k), stack = (a^{i-j-k} y), i > j, and i,j,k > 0. Observe that i≠j and that $c^k \in c^+$. Using this transition adds z to ci without changing the stack. Therefore, ci = (a^i b^j c^+ z), stack = (a^* y), i ≠ j, and i,j > 0. Thus, G-INV holds.

Deterministic pdas

L = $\{a^m b^n c^p \mid m \neq n \wedge m,n,p > 0\}$

- `((C c (a)) (C ,EMP))`: By inductive hypothesis, `C-INV` holds. This means ci = (a^i b^j c^k), stack = (a^{i-j-k} y), i > j, and i,j,k > 0. Using this transition adds a c to ci and pops an a from the stack. Therefore, ci = (a^i b^j c^{k+1}), stack = ($a^{i-j-(k+1)}$ y), i > j, and i,j,k+1 > 0. Without loss of generality, we may think of k+1 as a new value for k. Therefore, C-INV holds.
- `((C z ,EMP) (G ,EMP))`: By inductive hypothesis, `C-INV` holds. This means ci = (a^i b^j c^k), stack = (a^{i-j-k} y), i > j, and i,j,k > 0. Observe that i≠j and that $c^k \in c^+$. Using this transition adds z to ci without changing the stack. Therefore, ci = (a^i b^j c^+ z), stack = (a^* y), i ≠ j, and i,j > 0. Thus, G-INV holds.
- `((C c (y)) (E (y)))`: By inductive hypothesis, `C-INV` holds. This means ci = (a^i b^j c^k), stack = (a^{i-j-k} y), i > j, and i,j,k > 0. Note that $c^k \in c^+$. Using this transition means that a c is added to ci without changing the stack. Therefore, after using this transition, ci = (a^i b^j c^+), stack = (y), j ≠ i, and i,j > 0. Thus, E-INV holds.

Deterministic pdas

L = $\{a^m b^n c^p \mid m \neq n \wedge m,n,p > 0\}$

- ((C c (a)) (C ,EMP)): By inductive hypothesis, C-INV holds. This means ci = (a^i b^j c^k), stack = (a^{i-j-k} y), i > j, and i,j,k > 0. Using this transition adds a c to ci and pops an a from the stack. Therefore, ci = (a^i b^j c^{k+1}), stack = ($a^{i-j-(k+1)}$ y), i > j, and i,j,k+1 > 0. Without loss of generality, we may think of k+1 as a new value for k. Therefore, C-INV holds.
- ((C z ,EMP) (G ,EMP)): By inductive hypothesis, C-INV holds. This means ci = (a^i b^j c^k), stack = (a^{i-j-k} y), i > j, and i,j,k > 0. Observe that i≠j and that $c^k \in c^+$. Using this transition adds z to ci without changing the stack. Therefore, ci = (a^i b^j c^+ z), stack = (a^* y), i ≠ j, and i,j > 0. Thus, G-INV holds.
- ((C c (y)) (E (y))): By inductive hypothesis, C-INV holds. This means ci = (a^i b^j c^k), stack = (a^{i-j-k} y), i > j, and i,j,k > 0. Note that $c^k \in c^+$. Using this transition means that a c is added to ci without changing the stack. Therefore, after using this transition, ci = (a^i b^j c^+), stack = (y), j ≠ i, and i,j > 0. Thus, E-INV holds.
- ((D b (y)) (D (y))): By inductive hypothesis, D-INV holds. This means ci = (a^i b^j), stack = (y), j > i, and i,j > 0. Using this transition adds a b to ci without changing the stack. Therefore, after using this transition, ci = (a^i b^{j+1}), stack = (y), j+1 > i, i,j+1 > 0. Without loss of generality, we may think of j+1 as a new value for j. Thus, D-INV holds.

Deterministic pdas

L = $\{a^m b^n c^p \mid m \neq n \wedge m,n,p > 0\}$

- ((C c (a)) (C ,EMP)): By inductive hypothesis, C-INV holds. This means ci = (a^i b^j c^k), stack = (a^{i-j-k} y), i > j, and i,j,k > 0. Using this transition adds a c to ci and pops an a from the stack. Therefore, ci = (a^i b^j c^{k+1}), stack = ($a^{i-j-(k+1)}$ y), i > j, and i,j,k+1 > 0. Without loss of generality, we may think of k+1 as a new value for k. Therefore, C-INV holds.
- ((C z ,EMP) (G ,EMP)): By inductive hypothesis, C-INV holds. This means ci = (a^i b^j c^k), stack = (a^{i-j-k} y), i > j, and i,j,k > 0. Observe that i≠j and that $c^k \in c^+$. Using this transition adds z to ci without changing the stack. Therefore, ci = (a^i b^j c^+ z), stack = (a^* y), i ≠ j, and i,j > 0. Thus, G-INV holds.
- ((C c (y)) (E (y))): By inductive hypothesis, C-INV holds. This means ci = (a^i b^j c^k), stack = (a^{i-j-k} y), i > j, and i,j,k > 0. Note that $c^k \in c^+$. Using this transition means that a c is added to ci without changing the stack. Therefore, after using this transition, ci = (a^i b^j c^+), stack = (y), j ≠ i, and i,j > 0. Thus, E-INV holds.
- ((D b (y)) (D (y))): By inductive hypothesis, D-INV holds. This means ci = (a^i b^j), stack = (y), j > i, and i,j > 0. Using this transition adds a b to ci without changing the stack. Therefore, after using this transition, ci = (a^i b^{j+1}), stack = (y), j+1 > i, i,j+1 > 0. Without loss of generality, we may think of j+1 as a new value for j. Thus, D-INV holds.
- ((D c (y)) (E (y))): By inductive hypothesis, D-INV holds. This means ci = (a^i b^j), stack = (y), j > i, and i,j > 0. Using this transition adds a c to ci without changing the stack. Therefore, after using this transition, ci = (a^i b^j c^+), stack = (y), j ≠ i, i,j > 0. Thus, E-INV holds.

Deterministic pdas

L = $\{a^m b^n c^p \mid m \neq n \wedge m,n,p > 0\}$

- ((C c (a)) (C ,EMP)): By inductive hypothesis, C-INV holds. This means ci = (a^i b^j c^k), stack = (a^{i-j-k} y), i > j, and i,j,k > 0. Using this transition adds a c to ci and pops an a from the stack. Therefore, ci = (a^i b^j c^{k+1}), stack = ($a^{i-j-(k+1)}$ y), i > j, and i,j,k+1 > 0. Without loss of generality, we may think of k+1 as a new value for k. Therefore, C-INV holds.
- ((C z ,EMP) (G ,EMP)): By inductive hypothesis, C-INV holds. This means ci = (a^i b^j c^k), stack = (a^{i-j-k} y), i > j, and i,j,k > 0. Observe that i≠j and that $c^k \in c^+$. Using this transition adds z to ci without changing the stack. Therefore, ci = (a^i b^j c^+ z), stack = (a^* y), i ≠ j, and i,j > 0. Thus, G-INV holds.
- ((C c (y)) (E (y))): By inductive hypothesis, C-INV holds. This means ci = (a^i b^j c^k), stack = (a^{i-j-k} y), i > j, and i,j,k > 0. Note that $c^k \in c^+$. Using this transition means that a c is added to ci without changing the stack. Therefore, after using this transition, ci = (a^i b^j c^+), stack = (y), j ≠ i, and i,j > 0. Thus, E-INV holds.
- ((D b (y)) (D (y))): By inductive hypothesis, D-INV holds. This means ci = (a^i b^j), stack = (y), j > i, and i,j > 0. Using this transition adds a b to ci without changing the stack. Therefore, after using this transition, ci = (a^i b^{j+1}), stack = (y), j+1 > i, i,j+1 > 0. Without loss of generality, we may think of j+1 as a new value for j. Thus, D-INV holds.
- ((D c (y)) (E (y))): By inductive hypothesis, D-INV holds. This means ci = (a^i b^j), stack = (y), j > i, and i,j > 0. Using this transition adds a c to ci without changing the stack. Therefore, after using this transition, ci = (a^i b^j c^+), stack = (y), j ≠ i, i,j > 0. Thus, E-INV holds.
- ((E c (y)) (E (y))): By inductive hypothesis, E-INV holds. This means ci = (a^i b^j c^+), stack = (y), j ≠ i, and i,j > 0. Using this transition adds a c to ci without modifying the stack. Therefore, after using this transition, ci = (a^i b^j c^+), stack = (y), j ≠ i, and i,j > 0. Thus, E-INV holds.

Deterministic pdas

L = $\{a^m b^n c^p \mid m \neq n \wedge m,n,p > 0\}$

- ((C c (a)) (C ,EMP)): By inductive hypothesis, C-INV holds. This means ci = (a^i b^j c^k), stack = (a^{i-j-k} y), i > j, and i,j,k > 0. Using this transition adds a c to ci and pops an a from the stack. Therefore, ci = (a^i b^j c^{k+1}), stack = ($a^{i-j-(k+1)}$ y), i > j, and i,j,k+1 > 0. Without loss of generality, we may think of k+1 as a new value for k. Therefore, C-INV holds.
- ((C z ,EMP) (G ,EMP)): By inductive hypothesis, C-INV holds. This means ci = (a^i b^j c^k), stack = (a^{i-j-k} y), i > j, and i,j,k > 0. Observe that i≠j and that $c^k \in c^+$. Using this transition adds z to ci without changing the stack. Therefore, ci = (a^i b^j c^+ z), stack = (a^* y), i ≠ j, and i,j > 0. Thus, G-INV holds.
- ((C c (y)) (E (y))): By inductive hypothesis, C-INV holds. This means ci = (a^i b^j c^k), stack = (a^{i-j-k} y), i > j, and i,j,k > 0. Note that $c^k \in c^+$. Using this transition means that a c is added to ci without changing the stack. Therefore, after using this transition, ci = (a^i b^j c^+), stack = (y), j ≠ i, and i,j > 0. Thus, E-INV holds.
- ((D b (y)) (D (y))): By inductive hypothesis, D-INV holds. This means ci = (a^i b^j), stack = (y), j > i, and i,j > 0. Using this transition adds a b to ci without changing the stack. Therefore, after using this transition, ci = (a^i b^{j+1}), stack = (y), j+1 > i, i,j+1 > 0. Without loss of generality, we may think of j+1 as a new value for j. Thus, D-INV holds.
- ((D c (y)) (E (y))): By inductive hypothesis, D-INV holds. This means ci = (a^i b^j), stack = (y), j > i, and i,j > 0. Using this transition adds a c to ci without changing the stack. Therefore, after using this transition, ci = (a^i b^j c^+), stack = (y), j ≠ i, i,j > 0. Thus, E-INV holds.
- ((E c (y)) (E (y))): By inductive hypothesis, E-INV holds. This means ci = (a^i b^j c^+), stack = (y), j ≠ i, and i,j > 0. Using this transition adds a c to ci without modifying the stack. Therefore, after using this transition, ci = (a^i b^j c^+), stack = (y), j ≠ i, and i,j > 0. Thus, E-INV holds.
- ((E z (y)) (F ,EMP)): By inductive hypothesis, E-INV holds. This means ci = (a^i b^j c^+), stack = (y), j ≠ i, and i,j > 0. Using this transition adds z to ci and pops y from the stack. Therefore, after using this transition, ci = (a^i b^j c^+ z), stack = '(), i ≠ j, and i,j > 0. Thus, F-INV holds.

Deterministic pdas

L = $\{a^m b^n c^p \mid m \neq n \wedge m,n,p > 0\}$

- ((G ,EMP (a)) (G ,EMP)): By inductive hypothesis, G-INV holds. This means ci = (a^i b^j c^+ z), stack = (a^* y), i $\neq$ j, and i,j $>$ 0. The use of this transition informs us that stack = (a^+ y) (i.e., the stack is not empty and contains at least one a). This transition pops an a from the stack without reading any input. Therefore, after using this transition, ci = (a^i b^j c^+ z), stack = (a^* y), i $\neq$ j, and i,j $>$ 0. Thus, G-INV holds.

Deterministic pdas

L = $\{a^m b^n c^p \mid m \neq n \wedge m,n,p > 0\}$

- ((G ,EMP (a)) (G ,EMP)): By inductive hypothesis, G-INV holds. This means ci = (a^i b^j c^+ z), stack = (a^* y), i $\neq$ j, and i,j $>$ 0. The use of this transition informs us that stack = (a^+ y) (i.e., the stack is not empty and contains at least one a). This transition pops an a from the stack without reading any input. Therefore, after using this transition, ci = (a^i b^j c^+ z), stack = (a^* y), i $\neq$ j, and i,j $>$ 0. Thus, G-INV holds.
- ((G ,EMP (y)) (F ,EMP)): By inductive hypothesis, G-INV holds. This means ci = (a^i b^j c^+ z), stack = (a^* y), i $\neq$ j, and i,j $>$ 0. The use of this transition informs us that stack = (y) (i.e., the stack does not contain any as). This transition pops y off the stack without consuming any input. Therefore, after using this transition, ci = (a^i b^j c^+ z), stack = '(), i $\neq$ j, and i,j $>$ 0. Thus, F-INV holds.
- This establishes the inductive step and concludes the proof.

Deterministic pdas

$L = \{a^m b^n c^p \mid m \neq n \wedge m,n,p > 0\}$

Lemma

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

-

Deterministic pdas

$L = \{a^m b^n c^p \mid m \neq n \wedge m,n,p > 0\}$

Lemma

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

-

Proof.

($\Rightarrow$) Assume w∈L. This means that $w = a^m b^n c^p$, where $m \neq n$ and $m,n,p > 0$. Given that state invariants always hold, there is a computation that has P consume w reaching F with an empty stack. Therefore, w∈L(P).
($\Leftarrow$) Assume w∈L(P). This means that M halts in F, the only final state, with an empty stack having consumed w. Given that the state invariants always hold we may conclude that $w = a^m b^n c^p$, where $m \neq n$ and $m,n,p > 0$. Therefore, w∈L. □

-

Deterministic pdas

$L = \{a^m b^n c^p \mid m \neq n \wedge m,n,p > 0\}$

Lemma

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

-

Proof.

($\Rightarrow$) Assume w∈L. This means that $w = a^m b^n c^p$, where $m \neq n$ and $m,n,p > 0$. Given that state invariants always hold, there is a computation that has P consume w reaching F with an empty stack. Therefore, w∈L(P).
($\Leftarrow$) Assume w∈L(P). This means that M halts in F, the only final state, with an empty stack having consumed w. Given that the state invariants always hold we may conclude that $w = a^m b^n c^p$, where $m \neq n$ and $m,n,p > 0$. Therefore, w∈L. □

-

Lemma

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

-

Deterministic pdas

$L = \{a^m b^n c^p \mid m \neq n \wedge m,n,p > 0\}$

Lemma

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

Proof.

($\Rightarrow$) Assume $w \in L$. This means that $w = a^m b^n c^p$, where $m \neq n$ and $m,n,p > 0$. Given that state invariants always hold, there is a computation that has P consume w reaching F with an empty stack. Therefore, $w \in L(P)$.
($\Leftarrow$) Assume $w \in L(P)$. This means that M halts in F, the only final state, with an empty stack having consumed w. Given that the state invariants always hold we may conclude that $w = a^m b^n c^p$, where $m \neq n$ and $m,n,p > 0$. Therefore, $w \in L$. □

Lemma

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

Proof.

($\Rightarrow$) Assume $w \notin L$. This means that $w \neq a^m b^n c^p$, $m \neq n$, or $m,n,p \not> 0$. Given that state predicate invariants always hold, P cannot consume w and end in F with an empty stack. Therefore, $w \notin L(P)$.
($\Leftarrow$) Assume $w \notin L(P)$. This means that P cannot transition into F with an empty stack having consumed w. Given that the state predicate invariants always hold, $w \neq a^m b^n c^p$, $m \neq n$, or $m,n,p \not> 0$. Therefore, $w \notin L$. □

Deterministic pdas

- HOMEWORK: 5–7

Deterministic pdas

- Are all CFLs deterministic CFLs?

Deterministic pdas

- Are all CFLs deterministic CFLs?
- Change the meaning of accept for, `M`, a deterministic `pda`
- A language,`L`, is deterministic context-free if `Lz = L(M)`
- If `M` accepts `Lz` then a nondeterministic `pda`, `M'`, may be constructed that nondeterministically "senses" the end of the input and transitions to states that consume no more input

Deterministic pdas

- Are all CFLs deterministic CFLs?
- Change the meaning of accept for, `M`, a deterministic `pda`
- A language,`L`, is deterministic context-free if `Lz = L(M)`
- If `M` accepts `Lz` then a nondeterministic `pda`, `M'`, may be constructed that nondeterministically "senses" the end of the input and transitions to states that consume no more input
- We shall build on two important concepts
- `M` is a simple `pda` with a bottom-of-the-stack symbol

Deterministic pdas

- Are all CFLs deterministic CFLs?
- Change the meaning of accept for, `M`, a deterministic `pda`
- A language,`L`, is deterministic context-free if `Lz = L(M)`
- If `M` accepts `Lz` then a nondeterministic `pda`, `M'`, may be constructed that nondeterministically "senses" the end of the input and transitions to states that consume no more input
- We shall build on two important concepts
- `M` is a simple `pda` with a bottom-of-the-stack symbol
- Second, we define a configuration, `C = (Q w s)`, as a *dead end* if in zero or more steps it leads to a configuration, `E = (R w' s')`, in which no input has been consumed (i.e., `w = w'`) or the stack has not shrunk (i.e., $|s'| \geq |s|$)
- If a `pda` does not have a dead-end configuration then it will eventually read all its input

Deterministic pdas

- Are all CFLs deterministic CFLs?
- Change the meaning of accept for, `M`, a deterministic `pda`
- A language,`L`, is deterministic context-free if `Lz = L(M)`
- If `M` accepts `Lz` then a nondeterministic `pda`, `M'`, may be constructed that nondeterministically "senses" the end of the input and transitions to states that consume no more input
- We shall build on two important concepts
- `M` is a simple `pda` with a bottom-of-the-stack symbol
- Second, we define a configuration, `C = (Q w s)`, as a *dead end* if in zero or more steps it leads to a configuration, `E = (R w' s')`, in which no input has been consumed (i.e., `w = w'`) or the stack has not shrunk (i.e., $|s'| \geq |s|$)
- If a `pda` does not have a dead-end configuration then it will eventually read all its input
- If `M` is simple then determining if a configuration is a dead end only depends on the current state, the next element to read, and the top stack symbol
- Let `D` be the set of dead-end configurations. Observe that `D` is finite.

Deterministic pdas

- Recall that context-free languages are not closed under complement
- To establish that not all context-free languages are deterministic, we shall prove that deterministic context-free languages are closed under complement.

Deterministic pdas

- Recall that context-free languages are not closed under complement
- To establish that not all context-free languages are deterministic, we shall prove that deterministic context-free languages are closed under complement.

Theorem

Deterministic context-free languages are closed under complement.

-

Deterministic pdas

- Recall that context-free languages are not closed under complement
- To establish that not all context-free languages are deterministic, we shall prove that deterministic context-free languages are closed under complement.

Theorem

Deterministic context-free languages are closed under complement.

-
- Assume L is a context-free language such that Lz is accepted by a deterministic simple pda M = (make-ndpda K Σ Γ S F R)
- Intuitively, we would like to build a pda for L's complement by simply switching the role of M's accepting and rejecting states
- Unfortunately, this does not work because pdas may reject without reading all the input: M is in a configuration for which no transition rules apply or M is in a dead-end configuration

Deterministic pdas

- Recall that context-free languages are not closed under complement
- To establish that not all context-free languages are deterministic, we shall prove that deterministic context-free languages are closed under complement.

Theorem

Deterministic context-free languages are closed under complement.

-
- Assume L is a context-free language such that Lz is accepted by a deterministic simple pda M = (make-ndpda K Σ Γ S F R)
- Intuitively, we would like to build a pda for L's complement by simply switching the role of M's accepting and rejecting states
- Unfortunately, this does not work because pdas may reject without reading all the input: M is in a configuration for which no transition rules apply or M is in a dead-end configuration
- We shall convert M into, M', an equivalent deterministic pda without dead-end configurations

Deterministic pdas

- Recall that context-free languages are not closed under complement
- To establish that not all context-free languages are deterministic, we shall prove that deterministic context-free languages are closed under complement.

Theorem

Deterministic context-free languages are closed under complement.

-
- Assume L is a context-free language such that Lz is accepted by a deterministic simple pda M = (make-ndpda K Σ Γ S F R)
- Intuitively, we would like to build a pda for L's complement by simply switching the role of M's accepting and rejecting states
- Unfortunately, this does not work because pdas may reject without reading all the input: M is in a configuration for which no transition rules apply or M is in a dead-end configuration
- We shall convert M into, M', an equivalent deterministic pda without dead-end configurations
- M' is constructed as follows:

  1. ∀ (Q a x)∈D, remove rules of the form ((Q a x) (H y)) in M and add ((Q a x) (U EMP)), where U is a new non-accepting state.
  2. ∀ a∈Σ, add the rule ((U a EMP) (U EMP))
  3. ∀ b∈Γ, add the rules ((U z EMP) (V EMP)) and ((V EMP b) (V EMP)), where V is a new non-accepting state.

- M' is deterministic given that M is deterministic and the added rules do not make M' nondeterministic
- M' rejects by moving to a non-accepting state whenever M does not read the entire input
- M' has no dead-end configurations
- Thus, M' always reads the entire input

Deterministic pdas

- Recall that context-free languages are not closed under complement
- To establish that not all context-free languages are deterministic, we shall prove that deterministic context-free languages are closed under complement.

Theorem

Deterministic context-free languages are closed under complement.

-
- Assume L is a context-free language such that Lz is accepted by a deterministic simple pda M = (make-ndpda K Σ Γ S F R)
- Intuitively, we would like to build a pda for L's complement by simply switching the role of M's accepting and rejecting states
- Unfortunately, this does not work because pdas may reject without reading all the input: M is in a configuration for which no transition rules apply or M is in a dead-end configuration
- We shall convert M into, M', an equivalent deterministic pda without dead-end configurations
- M' is constructed as follows:

  1. ∀ (Q a x)∈D, remove rules of the form ((Q a x) (H y)) in M and add ((Q a x) (U EMP)), where U is a new non-accepting state.
  2. ∀ a∈Σ, add the rule ((U a EMP) (U EMP))
  3. ∀ b∈Γ, add the rules ((U z EMP) (V EMP)) and ((V EMP b) (V EMP)), where V is a new non-accepting state.

- M' is deterministic given that M is deterministic and the added rules do not make M' nondeterministic
- M' rejects by moving to a non-accepting state whenever M does not read the entire input
- M' has no dead-end configurations
- Thus, M' always reads the entire input
- Given that M' reads the entire input, reversing the role of accepting and non-accepting states results in a machine that accepts (Σ^*-L)z = L's complement □

Deterministic pdas

- This a rather stunning result
- It allows us to prove that there are context-free languages that are not deterministic

Deterministic pdas

- This a rather stunning result
- It allows us to prove that there are context-free languages that are not deterministic
- $L = \{a^i b^j c^k \mid i \neq j \vee i \neq k\}$

Deterministic pdas

- This a rather stunning result
- It allows us to prove that there are context-free languages that are not deterministic
- $L = \{a^i b^j c^k \mid i \neq j \vee i \neq k\}$
- Assume L is a deterministic context-free language

Deterministic pdas

- This a rather stunning result
- It allows us to prove that there are context-free languages that are not deterministic
- $L = \{a^i b^j c^k \mid i \neq j \vee i \neq k\}$
- Assume L is a deterministic context-free language
- The theorem informs us that L's complement, $\bar{L}$, is also a deterministic context-free language

Deterministic pdas

- This a rather stunning result
- It allows us to prove that there are context-free languages that are not deterministic
- $L = \{a^i b^j c^k \mid i \neq j \vee i \neq k\}$
- Assume L is a deterministic context-free language
- The theorem informs us that L's complement, $\bar{L}$, is also a deterministic context-free language
- Recall that the intersection of a context-free language and a regular language is a context-free language

Deterministic pdas

- This a rather stunning result
- It allows us to prove that there are context-free languages that are not deterministic
- $L = \{a^i b^j c^k \mid i \neq j \vee i \neq k\}$
- Assume L is a deterministic context-free language
- The theorem informs us that L's complement, $\bar{L}$, is also a deterministic context-free language
- Recall that the intersection of a context-free language and a regular language is a context-free language
- Observe that $\bar{L} \cap a^* b^* c^* = a^n b^n c^n$

Deterministic pdas

- This a rather stunning result
- It allows us to prove that there are context-free languages that are not deterministic
- $L = \{a^i b^j c^k \mid i \neq j \vee i \neq k\}$
- Assume L is a deterministic context-free language
- The theorem informs us that L's complement, $\bar{L}$, is also a deterministic context-free language
- Recall that the intersection of a context-free language and a regular language is a context-free language
- Observe that $\bar{L} \cap a^* b^* c^* = a^n b^n c^n$
- Our assumption is wrong and L is not a deterministic context-free language.

Deterministic pdas

- This a rather stunning result
- It allows us to prove that there are context-free languages that are not deterministic
- $L = \{a^i b^j c^k \mid i \neq j \vee i \neq k\}$
- Assume L is a deterministic context-free language
- The theorem informs us that L's complement, $\bar{L}$, is also a deterministic context-free language
- Recall that the intersection of a context-free language and a regular language is a context-free language
- Observe that $\bar{L} \cap a^* b^* c^* = a^n b^n c^n$
- Our assumption is wrong and L is not a deterministic context-free language.
- The deterministic context-free languages are a proper subset of the context-free languages

Deterministic pdas

- This a rather stunning result
- It allows us to prove that there are context-free languages that are not deterministic
- $L = \{a^i b^j c^k \mid i \neq j \vee i \neq k\}$
- Assume L is a deterministic context-free language
- The theorem informs us that L's complement, $\bar{L}$, is also a deterministic context-free language
- Recall that the intersection of a context-free language and a regular language is a context-free language
- Observe that $\bar{L} \cap a^* b^* c^* = a^n b^n c^n$
- Our assumption is wrong and L is not a deterministic context-free language.
- The deterministic context-free languages are a proper subset of the context-free languages
- **In the context of *pda*s, nondeterminism is more powerful than determinism**

Deterministic pdas

- This a rather stunning result
- It allows us to prove that there are context-free languages that are not deterministic
- $L = \{a^i b^j c^k \mid i \neq j \vee i \neq k\}$
- Assume L is a deterministic context-free language
- The theorem informs us that L's complement, $\bar{L}$, is also a deterministic context-free language
- Recall that the intersection of a context-free language and a regular language is a context-free language
- Observe that $\bar{L} \cap a^* b^* c^* = a^n b^n c^n$
- Our assumption is wrong and L is not a deterministic context-free language.
- The deterministic context-free languages are a proper subset of the context-free languages
- **In the context of *pda*s, nondeterminism is more powerful than determinism**
- This makes software development more difficult: parsers decide if a program is valid

Deterministic pdas

Theorem

Deterministic context-free languages are not closed under union.

- Assume deterministic context-free languages are closed under union

Deterministic pdas

Theorem

Deterministic context-free languages are not closed under union.

- Assume deterministic context-free languages are closed under union
- Consider the following deterministic context-free languages:
  - $L = \{a^m b^n c^p \mid m \neq n \wedge m,n,p > 0\}$
  - $S = \{a^m b^n c^p \mid n \neq p \wedge m,n,p > 0\}$

Deterministic pdas

Theorem

Deterministic context-free languages are not closed under union.

- Assume deterministic context-free languages are closed under union
- Consider the following deterministic context-free languages:
  - $L = \{a^m b^n c^p \mid m \neq n \wedge m,n,p > 0\}$
  - $S = \{a^m b^n c^p \mid n \neq p \wedge m,n,p > 0\}$
- Let $LUS = L \cup S$
- Given our assumption, LUS is a deterministic context-free language

Deterministic pdas

Theorem

Deterministic context-free languages are not closed under union.

- Assume deterministic context-free languages are closed under union
- Consider the following deterministic context-free languages:
  - $L = \{a^m b^n c^p \mid m \neq n \wedge m,n,p > 0\}$
  - $S = \{a^m b^n c^p \mid n \neq p \wedge m,n,p > 0\}$
- Let $LUS = L \cup S$
- Given our assumption, LUS is a deterministic context-free language
- Deterministic context-free languages are closed under complement
- Thus, the complement of LSU, NOT-LSU, is also a deterministic context-free language

Deterministic pdas

Theorem

Deterministic context-free languages are not closed under union.

- Assume deterministic context-free languages are closed under union
- Consider the following deterministic context-free languages:
  - $L = \{a^m b^n c^p \mid m \neq n \wedge m,n,p > 0\}$
  - $S = \{a^m b^n c^p \mid n \neq p \wedge m,n,p > 0\}$
- Let $LUS = L \cup S$
- Given our assumption, LUS is a deterministic context-free language
- Deterministic context-free languages are closed under complement
- Thus, the complement of LSU, NOT-LSU, is also a deterministic context-free language
- NOT-LSU is the union of the following two languages:
  - $\{a^m b^n c^p \mid m = n = p \wedge m,n,p > 0\}$
  - $\{w \mid w \in \{a\ b\ c\}^* \wedge$ w has letters out of order $\vee$ w is missing at least one element in $\{a\ b\ c\}\}$

Deterministic pdas

Theorem

Deterministic context-free languages are not closed under union.

- Assume deterministic context-free languages are closed under union
- Consider the following deterministic context-free languages:
  - $L = \{a^m b^n c^p \mid m \neq n \wedge m,n,p > 0\}$
  - $S = \{a^m b^n c^p \mid n \neq p \wedge m,n,p > 0\}$
- Let $LUS = L \cup S$
- Given our assumption, LUS is a deterministic context-free language
- Deterministic context-free languages are closed under complement
- Thus, the complement of LSU, NOT-LSU, is also a deterministic context-free language
- NOT-LSU is the union of the following two languages:
  - $\{a^m b^n c^p \mid m = n = p \wedge m,n,p > 0\}$
  - $\{w \mid w \in \{a\ b\ c\}^* \wedge$ w has letters out of order $\vee$ w is missing at least one element in $\{a\ b\ c\}\}$
- The intersection of a context-free language and a regular language is a context-free language

Deterministic pdas

Theorem

Deterministic context-free languages are not closed under union.

- Assume deterministic context-free languages are closed under union
- Consider the following deterministic context-free languages:
  - $L = \{a^m b^n c^p \mid m \neq n \wedge m,n,p > 0\}$
  - $S = \{a^m b^n c^p \mid n \neq p \wedge m,n,p > 0\}$
- Let $LUS = L \cup S$
- Given our assumption, LUS is a deterministic context-free language
- Deterministic context-free languages are closed under complement
- Thus, the complement of LSU, NOT-LSU, is also a deterministic context-free language
- NOT-LSU is the union of the following two languages:
  - $\{a^m b^n c^p \mid m = n = p \wedge m,n,p > 0\}$
  - $\{w \mid w \in \{a\ b\ c\}^* \wedge$ w has letters out of order $\vee$ w is missing at least one element in $\{a\ b\ c\}\}$
- The intersection of a context-free language and a regular language is a context-free language
- NOT-LSU $\cap\ a^*b^*c^* = a^n b^n c^n$, where $n>0$
- This is not a context-free language
- Our assumption is wrong and deterministic context-free languages are not closed under union

Deterministic pdas

Theorem

Deterministic context-free languages are not closed under intersection.

- Assume deterministic context-free languages are closed under intersection

Deterministic pdas

Theorem

Deterministic context-free languages are not closed under intersection.

- Assume deterministic context-free languages are closed under intersection
- Consider the following languages:
  - $L = \{a^m b^n c^p \mid m = n \wedge m,n,p \geq 0\}$
  - $S = \{a^m b^n c^p \mid n = p \wedge m,n,p \geq 0\}$

Deterministic pdas

Theorem

Deterministic context-free languages are not closed under intersection.

- Assume deterministic context-free languages are closed under intersection
- Consider the following languages:
  - $L = \{a^m b^n c^p | \ m = n \wedge m,n,p \geq 0\}$
  - $S = \{a^m b^n c^p | \ n = p \wedge m,n,p \geq 0\}$
- The intersection of L and S is $\{a^n b^n c^n | \ n \geq 0\}$
- This language is not context-free and, therefore, deterministic context-free languages are not closed under intersection

Deterministic pdas

- HOMEWORK: 8, 10
- QUIZ: 9 (due in 1 week)