Computational Models - Lecture 4

- Push Down Automata (PDA)

Computational Models - Lecture 4

- Push Down Automata (PDA)
- Nondeterminism adds power to PDAs

Computational Models - Lecture 4

- Push Down Automata (PDA)
- Nondeterminism adds power to PDAs
- Equivalence of CFGs and PDAs

Computational Models - Lecture 4

- Push Down Automata (PDA)
- Nondeterminism adds power to PDAs
- Equivalence of CFGs and PDAs
- Pumping Lemma for context free languages

Computational Models - Lecture 4

- Push Down Automata (PDA)
- Nondeterminism adds power to PDAs
- Equivalence of CFGs and PDAs
- Pumping Lemma for context free languages

Computational Models - Lecture 4

- Push Down Automata (PDA)
- Nondeterminism adds power to PDAs
- Equivalence of CFGs and PDAs
- Pumping Lemma for context free languages

- Sipser's book, 2.2 & 2.3

- even if you can’t hear me

- even if you can’t hear me
- There will be a mid-term exam on Friday, Dec. 3

- even if you can't hear me
- There will be a mid-term exam on Friday, Dec. 3
- Material includes first five lectures

- even if you can't hear me
- There will be a mid-term exam on Friday, Dec. 3
  - Material includes first five lectures
  - Can use 4 pages, double sided, any size font you can read, and also your heads (optionally)

String Generators and String Acceptors

- Regular expressions are string generators – they tell us how to generate all strings in a language L

String Generators and String Acceptors

- Regular expressions are string generators – they tell us how to generate all strings in a language L
- Finite Automata (DFA, NFA) are string acceptors – they tell us if a specific string w is in L

String Generators and String Acceptors

- Regular expressions are string generators – they tell us how to generate all strings in a language L
- Finite Automata (DFA, NFA) are string acceptors – they tell us if a specific string w is in L
- CFGs are string generators

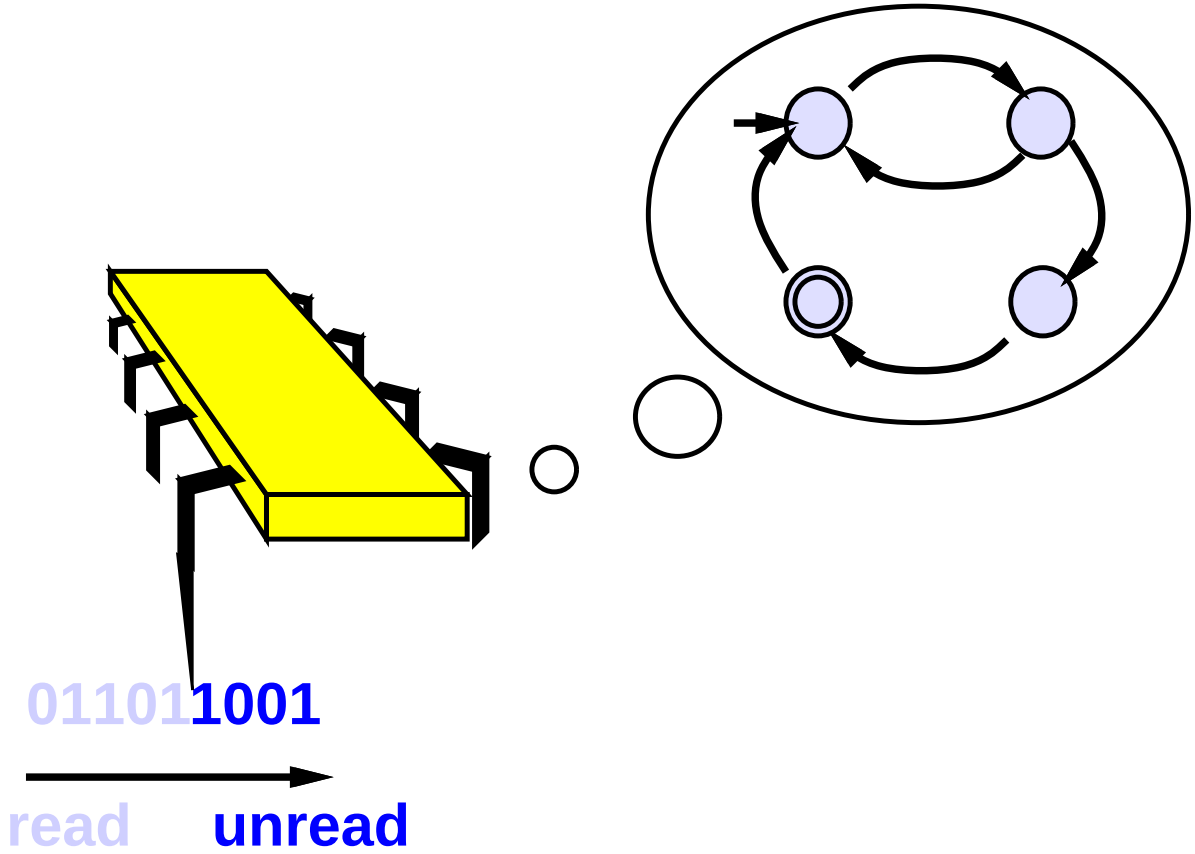
String Generators and String Acceptors

- Regular expressions are string generators – they tell us how to generate all strings in a language L
- Finite Automata (DFA, NFA) are string acceptors – they tell us if a specific string w is in L
- CFGs are string generators
- Are there string acceptors for Context-Free languages?

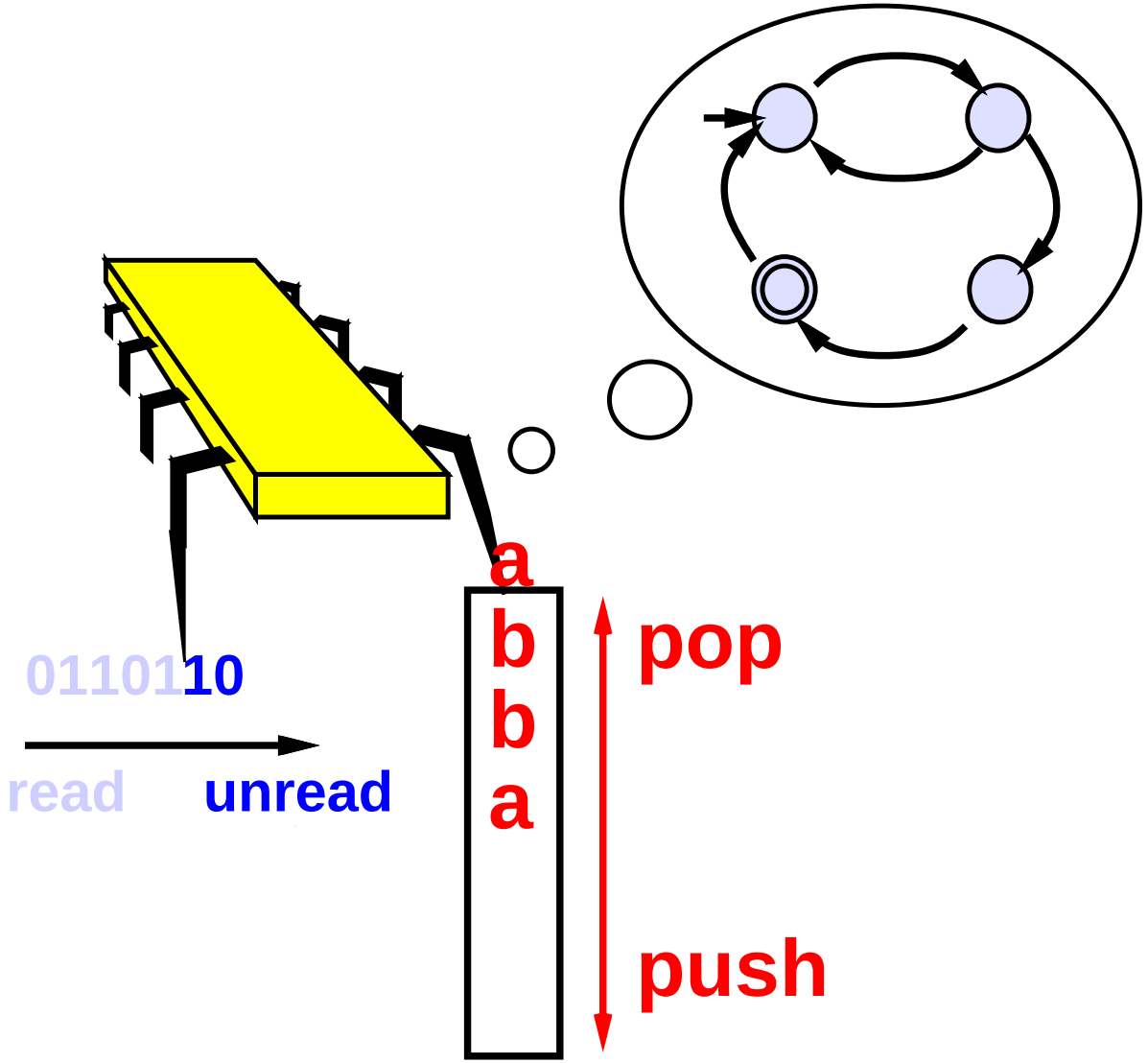
String Generators and String Acceptors

- Regular expressions are string generators – they tell us how to generate all strings in a language L
- Finite Automata (DFA, NFA) are string acceptors – they tell us if a specific string w is in L
- CFGs are string generators
- Are there string acceptors for Context-Free languages?
- YES! Push-down automata

A Finite Automaton



A PushDown Automaton



An Example

Recall that the language $\{0^n 1^n \mid n \geq 0\}$ is not regular. Consider the following PDA:

- Read input symbols

An Example

Recall that the language $\{0^n 1^n \mid n \geq 0\}$ is not regular. Consider the following PDA:

- Read input symbols
- For each 0, push it on the stack

An Example

Recall that the language $\{0^n 1^n \mid n \geq 0\}$ is not regular. Consider the following PDA:

- Read input symbols
- For each 0, push it on the stack
- As soon as a 1 is seen, pop a 0 for each 1 read

An Example

Recall that the language $\{0^n 1^n \mid n \geq 0\}$ is not regular. Consider the following PDA:

- Read input symbols
- For each 0, push it on the stack
- As soon as a 1 is seen, pop a 0 for each 1 read
- Accept if stack is empty when last symbol read

An Example

Recall that the language $\{0^n 1^n \mid n \geq 0\}$ is not regular. Consider the following PDA:

- Read input symbols
- For each 0, push it on the stack
- As soon as a 1 is seen, pop a 0 for each 1 read
- Accept if stack is empty when last symbol read
- Reject if

An Example

Recall that the language $\{0^n 1^n \mid n \geq 0\}$ is not regular. Consider the following PDA:

- Read input symbols
- For each 0, push it on the stack
- As soon as a 1 is seen, pop a 0 for each 1 read
- Accept if stack is empty when last symbol read
- Reject if
  - Stack is non-empty when last symbol read

An Example

Recall that the language $\{0^n 1^n \mid n \geq 0\}$ is not regular. Consider the following PDA:

- Read input symbols
- For each 0, push it on the stack
- As soon as a 1 is seen, pop a 0 for each 1 read
- Accept if stack is empty when last symbol read
- Reject if
  - Stack is non-empty when last symbol read
  - Stack is empty but input symbol(s) still exist,

An Example

Recall that the language $\{0^n 1^n \mid n \geq 0\}$ is not regular. Consider the following PDA:

- Read input symbols
- For each 0, push it on the stack
- As soon as a 1 is seen, pop a 0 for each 1 read
- Accept if stack is empty when last symbol read
- Reject if
  - Stack is non-empty when last symbol read
  - Stack is empty but input symbol(s) still exist,
  - 0 is read after 1.

PDA Configuration

- A Configuration of a Push-Down Automata is a triplet

PDA Configuration

- A Configuration of a Push-Down Automata is a triplet
- (<state>, <remaining input string>, <stack>)

PDA Configuration

- A Configuration of a Push-Down Automata is a triplet
- (<state>, <remaining input string>, <stack>)

PDA Configuration

- A Configuration of a Push-Down Automata is a triplet
- (<state>, <remaining input string>, <stack>)

- A configuration is like a snapshot of PDA progress

PDA Configuration

- A Configuration of a Push-Down Automata is a triplet
- (<state>, <remaining input string>, <stack>)

- A configuration is like a snapshot of PDA progress
- A PDA computation is a sequence of successive configurations, starting from start configuration

PDA Configuration

- A Configuration of a Push-Down Automata is a triplet
- (<state>, <remaining input string>, <stack>)

- A configuration is like a snapshot of PDA progress
- A PDA computation is a sequence of successive configurations, starting from start configuration
- The string describing the stack has top of the stack on the left (technical item)

Comparing PDA and Finite Automata

- PDA may be deterministic or non-deterministic.

Comparing PDA and Finite Automata

- PDA may be deterministic or non-deterministic.
- Unlike finite automata, non-determinism adds power.

Comparing PDA and Finite Automata

- PDA may be deterministic or non-deterministic.
- Unlike finite automata, non-determinism adds power.
- There are some languages accepted only by non-deterministic PDAs.

Comparing PDA and Finite Automata

- PDA may be deterministic or non-deterministic.
- Unlike finite automata, non-determinism adds power.
- There are some languages accepted only by non-deterministic PDAs.

Transition function δ looks different than DFA or NFA cases, reflecting stack functionality.

The Transition Function

Denote input alphabet by Σ and stack alphabet by Γ.

- the domain of the transition function δ is

The Transition Function

Denote input alphabet by Σ and stack alphabet by Γ.

- the domain of the transition function δ is
  - current state: Q

The Transition Function

Denote input alphabet by Σ and stack alphabet by Γ.

- the domain of the transition function δ is
  - current state: Q
  - next input, if any: Σ_ε (=$\Sigma \cup \{\varepsilon\}$)

The Transition Function

Denote input alphabet by Σ and stack alphabet by Γ.

- the domain of the transition function δ is
  - current state: Q
  - next input, if any: Σ_ε (=$\Sigma \cup \{\varepsilon\}$)
  - stack symbol popped, if any: Γ_ε(=$\Gamma \cup \{\varepsilon\}$)

The Transition Function

Denote input alphabet by Σ and stack alphabet by Γ.

- the domain of the transition function δ is
  - current state: Q
  - next input, if any: Σ_ε (=$\Sigma \cup \{\varepsilon\}$)
  - stack symbol popped, if any: Γ_ε(=$\Gamma \cup \{\varepsilon\}$)
- and its range is

The Transition Function

Denote input alphabet by Σ and stack alphabet by Γ.

- the domain of the transition function δ is
  - current state: Q
  - next input, if any: Σ_ε (=$\Sigma \cup \{\varepsilon\}$)
  - stack symbol popped, if any: Γ_ε(=$\Gamma \cup \{\varepsilon\}$)
- and its range is
  - new state: Q

The Transition Function

Denote input alphabet by Σ and stack alphabet by Γ.

- the domain of the transition function δ is
  - current state: Q
  - next input, if any: Σ_ε (=$\Sigma \cup \{\varepsilon\}$)
  - stack symbol popped, if any: Γ_ε(=$\Gamma \cup \{\varepsilon\}$)
- and its range is
  - new state: Q
  - stack symbol pushed, if any: Γ_ε

The Transition Function

Denote input alphabet by Σ and stack alphabet by Γ.

- the domain of the transition function δ is
  - current state: Q
  - next input, if any: Σ_ε (=$\Sigma \cup \{\varepsilon\}$)
  - stack symbol popped, if any: Γ_ε(=$\Gamma \cup \{\varepsilon\}$)
- and its range is
  - new state: Q
  - stack symbol pushed, if any: Γ_ε
  - non-determinism: $\mathcal{P}$(two components above)

The Transition Function

Denote input alphabet by Σ and stack alphabet by Γ.

- the domain of the transition function δ is
  - current state: Q
  - next input, if any: Σ_ε (=$\Sigma \cup \{\varepsilon\}$)
  - stack symbol popped, if any: Γ_ε(=$\Gamma \cup \{\varepsilon\}$)
- and its range is
  - new state: Q
  - stack symbol pushed, if any: Γ_ε
  - non-determinism: $\mathcal{P}$(two components above)
- $\delta : Q \times \Sigma_\varepsilon \times \Gamma_\varepsilon \rightarrow \mathcal{P}(Q \times \Gamma_\varepsilon)$

Formal Definitions

A pushdown automaton (PDA) is a 6-tuple $(Q, \Sigma, \Gamma, \delta, q_0, F)$, where

- Q is a finite set called the *states*,

Formal Definitions

A pushdown automaton (PDA) is a 6-tuple $(Q, \Sigma, \Gamma, \delta, q_0, F)$, where

- Q is a finite set called the *states*,
- Σ is a finite set called the *input alphabet*,

Formal Definitions

A pushdown automaton (PDA) is a 6-tuple $(Q, \Sigma, \Gamma, \delta, q_0, F)$, where

- Q is a finite set called the *states*,
- Σ is a finite set called the *input alphabet*,
- Γ is a finite set called the *stack alphabet*,

Formal Definitions

A pushdown automaton (PDA) is a 6-tuple $(Q, \Sigma, \Gamma, \delta, q_0, F)$, where

- Q is a finite set called the *states*,
- Σ is a finite set called the *input alphabet*,
- Γ is a finite set called the *stack alphabet*,
- $\delta : Q \times \Sigma_\varepsilon \times \Gamma_\varepsilon \to \mathcal{P}(Q \times \Gamma_\varepsilon)$ is the *transition function*,

Formal Definitions

A pushdown automaton (PDA) is a 6-tuple $(Q, \Sigma, \Gamma, \delta, q_0, F)$, where

- Q is a finite set called the *states*,
- Σ is a finite set called the *input alphabet*,
- Γ is a finite set called the *stack alphabet*,
- $\delta : Q \times \Sigma_\varepsilon \times \Gamma_\varepsilon \to \mathcal{P}(Q \times \Gamma_\varepsilon)$ is the *transition function*,
- $q_0 \in Q$ is the *start state*, and

Formal Definitions

A pushdown automaton (PDA) is a 6-tuple $(Q, \Sigma, \Gamma, \delta, q_0, F)$, where

- Q is a finite set called the *states*,
- Σ is a finite set called the *input alphabet*,
- Γ is a finite set called the *stack alphabet*,
- $\delta : Q \times \Sigma_\varepsilon \times \Gamma_\varepsilon \to \mathcal{P}(Q \times \Gamma_\varepsilon)$ is the *transition function*,
- $q_0 \in Q$ is the *start state*, and
- $F \subseteq Q$ is the set of *accept states*.

Conventions

- It will be convenient to be able to know when the stack is empty, but there is no built-in mechanism to do that.

Conventions

- It will be convenient to be able to know when the stack is empty, but there is no built-in mechanism to do that.
  - Solution:

Conventions

- It will be convenient to be able to know when the stack is empty, but there is no built-in mechanism to do that.
  - Solution:
  - Start by pushing $ onto stack.

Conventions

- It will be convenient to be able to know when the stack is empty, but there is no built-in mechanism to do that.
  - Solution:
  - Start by pushing $ onto stack.
  - When you see it again, stack is empty.

Conventions

- It will be convenient to be able to know when the stack is empty, but there is no built-in mechanism to do that.
  - Solution:
  - Start by pushing $ onto stack.
  - When you see it again, stack is empty.
- Question: When is input string exhausted?

Conventions

- It will be convenient to be able to know when the stack is empty, but there is no built-in mechanism to do that.
  - Solution:
  - Start by pushing $ onto stack.
  - When you see it again, stack is empty.
- Question: When is input string exhausted?
  - It doesn't matter. No modifications needed!

Conventions

- It will be convenient to be able to know when the stack is empty, but there is no built-in mechanism to do that.
  - Solution:
  - Start by pushing $ onto stack.
  - When you see it again, stack is empty.
- Question: When is input string exhausted?
  - It doesn't matter. No modifications needed!
  - 'Caus accepting state is effective only if input string was exhausted!

Notation

- Transition $a, b \rightarrow c$ means

Notation

- Transition $a, b \rightarrow c$ means
  - read a from input

Notation

- Transition $a, b \rightarrow c$ means
  - read a from input
  - pop b from stack

Notation

- Transition $a, b \rightarrow c$ means
  - read a from input
  - pop b from stack
  - push c onto stack

Notation

- Transition $a, b \rightarrow c$ means
  - read a from input
  - pop b from stack
  - push c onto stack
- Meaning of ε transitions:

Notation

- Transition $a, b \rightarrow c$ means
  - read a from input
  - pop b from stack
  - push c onto stack
- Meaning of ε transitions:
  - if $a = \varepsilon$, don't read inputs

Notation

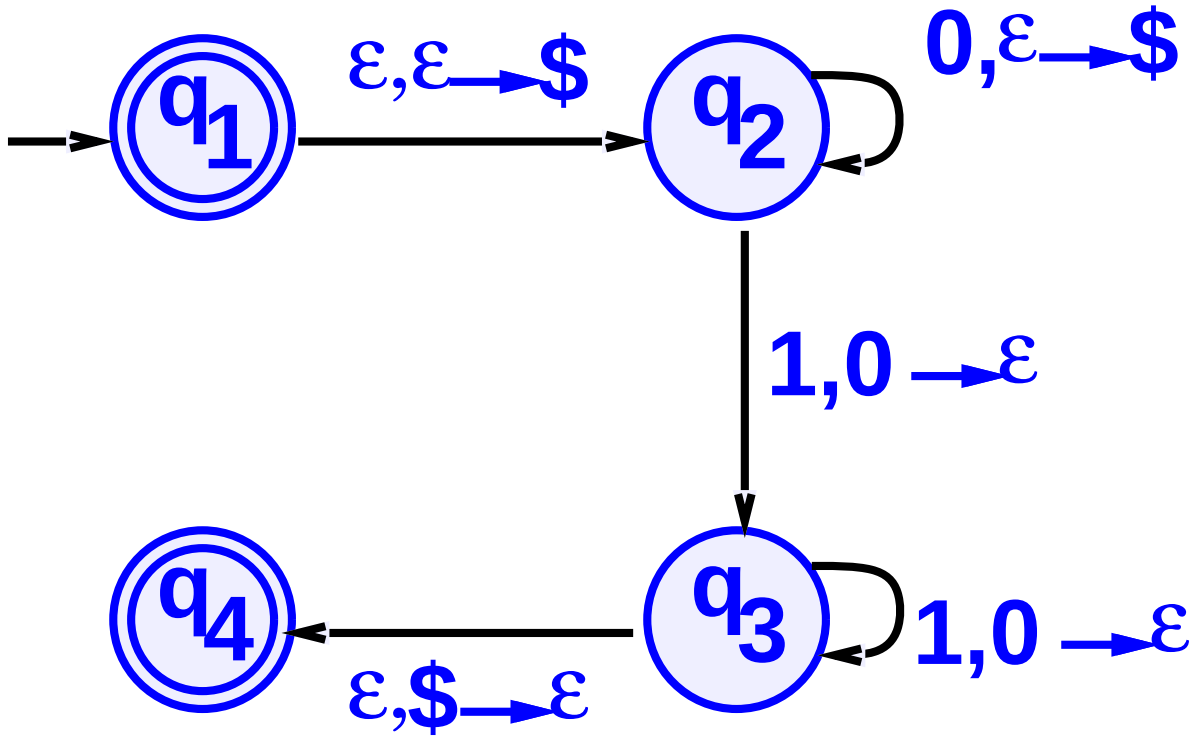
- Transition $a, b \rightarrow c$ means
  - read a from input
  - pop b from stack
  - push c onto stack
- Meaning of ε transitions:
  - if $a = \varepsilon$, don't read inputs
  - if $b = \varepsilon$, don't pop any symbols

Notation

- Transition $a, b \rightarrow c$ means
  - read a from input
  - pop b from stack
  - push c onto stack
- Meaning of ε transitions:
  - if $a = \varepsilon$, don't read inputs
  - if $b = \varepsilon$, don't pop any symbols
  - if $c = \varepsilon$, don't push any symbols

Example

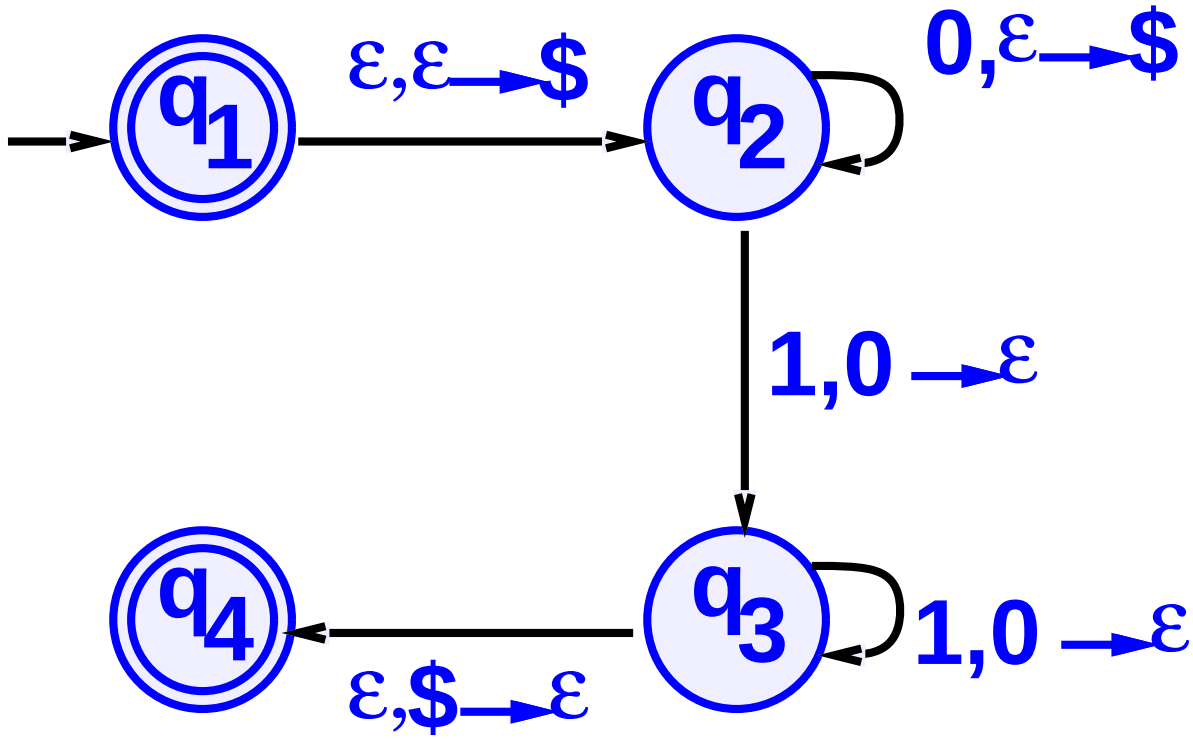
The PDA



accepts $\{0^n 1^n | n \geq 1\}$.

Example

The PDA



accepts $\{0^n 1^n | n \geq 1\}$.

(does it also accept the empty string?)

Another Example

A PDA that accepts

$$\{a^i b^j c^k | i, j, k > 0 \text{ and } i = j \text{ or } i = k\}$$

Informally:

- read and push a's

Another Example

A PDA that accepts

$$\{a^i b^j c^k | i, j, k > 0 \text{ and } i = j \text{ or } i = k\}$$

Informally:

- read and push a's
- either pop and match with b's

Another Example

A PDA that accepts

$$\{a^i b^j c^k | i, j, k > 0 \text{ and } i = j \text{ or } i = k\}$$

Informally:

- read and push a's
- either pop and match with b's
- or else pop and match with c's

Another Example

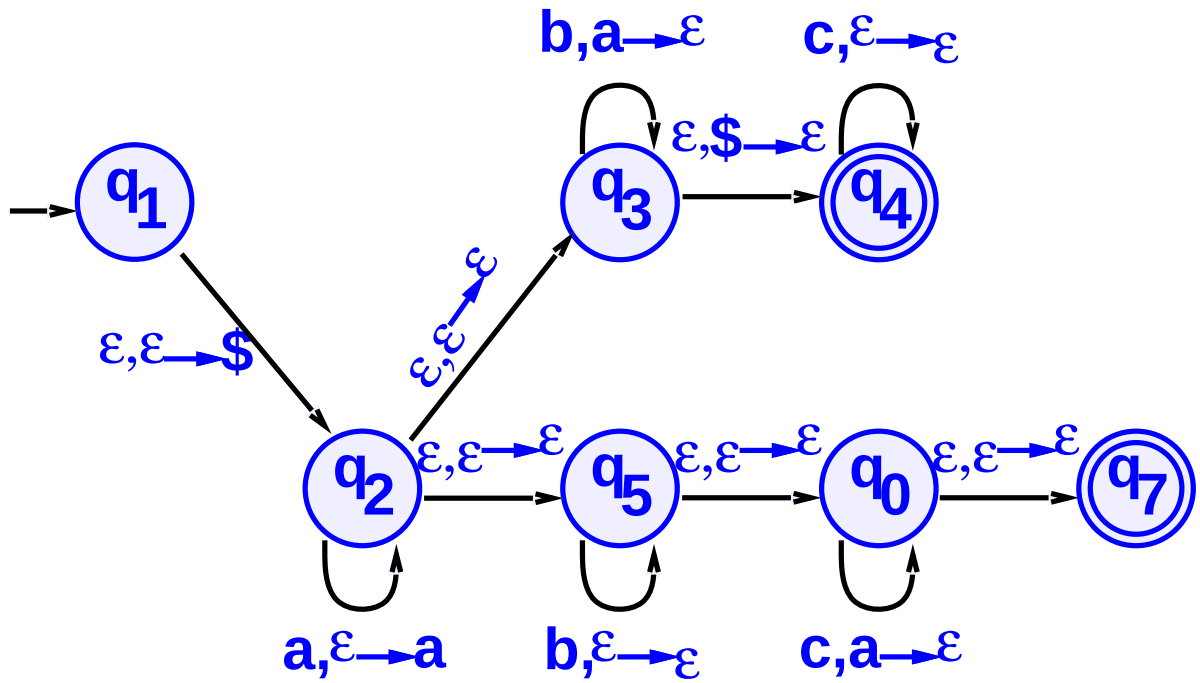
A PDA that accepts

$$\{a^i b^j c^k | i, j, k > 0 \text{ and } i = j \text{ or } i = k\}$$

Informally:

- read and push a's
- either pop and match with b's
- or else pop and match with c's
- a non-deterministic choice!

Another Example (cont.)



A PDA that accepts

$$\{a^i b^j c^k | i, j, k > 0 \text{ and } i = j \text{ or } i = k\}$$

Another Example (cont.)

A PDA that accepts

$$\{a^i b^j c^k | i, j, k > 0 \text{ and } i = j \text{ or } i = k\}$$

- Note: non-determinism is essential here!

Another Example (cont.)

A PDA that accepts

$$\{a^i b^j c^k | i, j, k > 0 \text{ and } i = j \text{ or } i = k\}$$

- Note: non-determinism is essential here!
- Unlike finite automata, non-determinism does add power.

Another Example (cont.)

A PDA that accepts

$$\{a^i b^j c^k | i, j, k > 0 \text{ and } i = j \text{ or } i = k\}$$

- Note: non-determinism is essential here!
- Unlike finite automata, non-determinism does add power.
- Let us try to think how a proof could go,

Another Example (cont.)

A PDA that accepts

$$\{a^i b^j c^k | i, j, k > 0 \text{ and } i = j \text{ or } i = k\}$$

- Note: non-determinism is essential here!
- Unlike finite automata, non-determinism does add power.
- Let us try to think how a proof could go,
- $\vdots$

Another Example (cont.)

A PDA that accepts

$$\{a^i b^j c^k | i, j, k > 0 \text{ and } i = j \text{ or } i = k\}$$

- Note: non-determinism is essential here!
- Unlike finite automata, non-determinism does add power.
- Let us try to think how a proof could go,
- ⋮
- and realize it does not seem trivial or immediate.

NonDeterministm Essential for PDAs

Theorem: Let M be a PDA that accepts

$$L = \{x^n y^n | n \geq 0\} \cup \left\{x^n y^{2n} \mid n \geq 0\right\} \ .$$

Then M is non-deterministic.

Proof: (www.cs.may.ie/∼jpower/Courses/parsing/node38.html)

Suppose, by way of contradiction, M is deterministic.

- Create two copies of this PDA, denoteded M_1 and M_2.

NonDeterministm Essential for PDAs

Theorem: Let M be a PDA that accepts

$$L = \{x^n y^n | n \geq 0\} \cup \left\{x^n y^{2n} \mid n \geq 0\right\} \ .$$

Then M is non-deterministic.

Proof: (www.cs.may.ie/∼jpower/Courses/parsing/node38.html)

Suppose, by way of contradiction, M is deterministic.

- Create two copies of this PDA, denoteded M_1 and M_2.
- Say two states "cousins" if they are copies of the same state in the original PDA.

NonDeterministm Essential for PDAs

Theorem: Let M be a PDA that accepts

$$L = \{x^n y^n | n \geq 0\} \cup \left\{x^n y^{2n} \mid n \geq 0\right\} \ .$$

Then M is non-deterministic.

Proof: (www.cs.may.ie/∼jpower/Courses/parsing/node38.html)

Suppose, by way of contradiction, M is deterministic.

- Create two copies of this PDA, denoteded M_1 and M_2.
- Say two states "cousins" if they are copies of the same state in the original PDA.
- We now modify these copies to make them into one PDA, M_0, over the alphabet $\{x, y, z\}$.

NonDeterministm Essential (cont.)

- States of the new M_0 are those of M_1 union M_2.

NonDeterministm Essential (cont.)

- States of the new M_0 are those of M_1 union M_2.
- Start state of the new M_0 is the start state of M_1.

NonDeterministm Essential (cont.)

- States of the new M_0 are those of M_1 union M_2.
- Start state of the new M_0 is the start state of M_1.
- The accepting states of the new M_0 are the accepting states of M_2.

NonDeterministm Essential (cont.)

- States of the new M_0 are those of M_1 union M_2.
- Start state of the new M_0 is the start state of M_1.
- The accepting states of the new M_0 are the accepting states of M_2.
- For both M_1 and M_2, add a new z transition for every existing y transition.

NonDeterministm Essential (cont.)

- States of the new M_0 are those of M_1 union M_2.
- Start state of the new M_0 is the start state of M_1.
- The accepting states of the new M_0 are the accepting states of M_2.
- For both M_1 and M_2, add a new z transition for every existing y transition.
- Now change any transition (x, y, or z) originating from an red accept state in M_1 so it goes to its "cousin" destination in M_2.

NonDeterministm Essential (cont.)

- What language M_0 recognizes? Consider its actions on an input of the form $x^k y^k z^k$ for some fixed k.

NonDeterministm Essential (cont.)

- What language M_0 recognizes? Consider its actions on an input of the form $x^k y^k z^k$ for some fixed k.
- Initially M_0 moves from the start state to an accept state of M_1 while consuming $x^k y^k$.

NonDeterministm Essential (cont.)

- What language M_0 recognizes? Consider its actions on an input of the form $x^k y^k z^k$ for some fixed k.
- Initially M_0 moves from the start state to an accept state of M_1 while consuming $x^k y^k$.
- Since no prefix of $x^k y^k$ is in L, M_0 does not go through any accept state earlier.

NonDeterministm Essential (cont.)

- What language M_0 recognizes? Consider its actions on an input of the form $x^k y^k z^k$ for some fixed k.
- Initially M_0 moves from the start state to an accept state of M_1 while consuming $x^k y^k$.
- Since no prefix of $x^k y^k$ is in L, M_0 does not go through any accept state earlier.
- Because M_0 is deterministic, it could reach no other state while consuming $x^k y^k$.

NonDeterministm Essential (cont.)

- What language M_0 recognizes? Consider its actions on an input of the form $x^k y^k z^k$ for some fixed k.
- Initially M_0 moves from the start state to an accept state of M_1 while consuming $x^k y^k$.
- Since no prefix of $x^k y^k$ is in L, M_0 does not go through any accept state earlier.
- Because M_0 is deterministic, it could reach no other state while consuming $x^k y^k$.
- Because M_1 accepts L, it would go on to accept another y^k block. Therefore, by construction, M_0 will move on z^k to its "M_2" half, and accept.

Conclusion of Proof

- We can conclude (after a few more arguments) that M_0 accepts the language $\{x^n y^n z^n | n \geq 0\}$. Contradiction. ♣

Conclusion of Proof

- We can conclude (after a few more arguments) that M_0 accepts the language $\{x^n y^n z^n | n \geq 0\}$. Contradiction. ♣

Conclusion of Proof

- We can conclude (after a few more arguments) that M_0 accepts the language $\{x^n y^n z^n | n \geq 0\}$. Contradiction. ♣

- Contradiction? What contradiction? What the $%&# does this contradict?

Conclusion of Proof

- We can conclude (after a few more arguments) that M_0 accepts the language $\{x^n y^n z^n | n \geq 0\}$. Contradiction. ♣

- Contradiction? What contradiction? What the $%&# does this contradict?

Conclusion of Proof

- We can conclude (after a few more arguments) that M_0 accepts the language $\{x^n y^n z^n | n \geq 0\}$. Contradiction. ♣

- Contradiction? What contradiction? What the $%&# does this contradict?

- And while in the mode of thinking about the proof, where would such argument fail if the original M were non-deterministic?

And Yet Another PDA Example

A palindrome has the form $ww^{\mathcal{R}}$.

- "Madam I'm Adam"

And Yet Another PDA Example

A palindrome has the form $ww^{\mathcal{R}}$.

- "Madam I'm Adam"
- "Dennis and Edna sinned"

And Yet Another PDA Example

A palindrome has the form $ww^{\mathcal{R}}$.

- "Madam I'm Adam"
- "Dennis and Edna sinned"
- "Red rum, sir, is murder"

And Yet Another PDA Example

A palindrome has the form $ww^{\mathcal{R}}$.

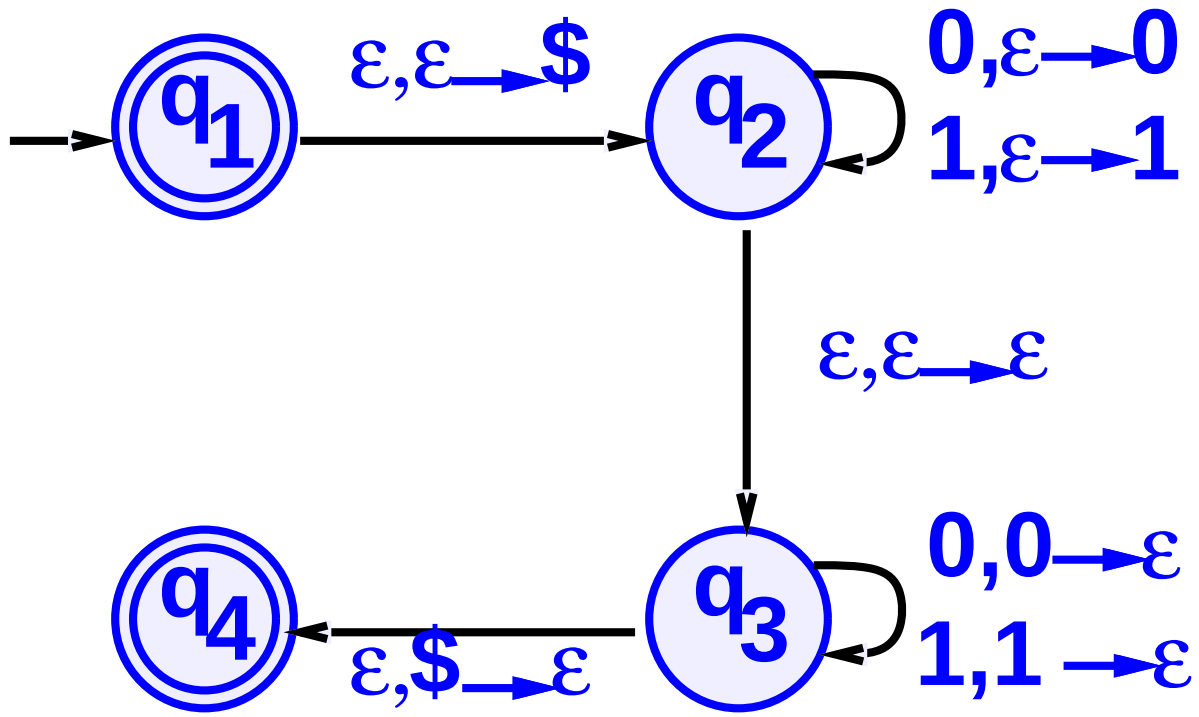
- "Madam I'm Adam"
- "Dennis and Edna sinned"
- "Red rum, sir, is murder"
- "In girum imus nocte et consumimur igni"

And Yet Another PDA Example

A palindrome has the form $ww^{\mathcal{R}}$.

- "Madam I'm Adam"
- "Dennis and Edna sinned"
- "Red rum, sir, is murder"
- "In girum imus nocte et consumimur igni"
- "$\nu\iota\psi o\nu\ \alpha\nu o\mu\eta\mu\alpha\tau\alpha\ \mu\eta\ \mu o\nu\alpha\nu\ o\psi\iota\nu$"

Yet Another Example (cont.)



This PDA accepts binary palindromes.

PDA Languages

The Push-Down Automata Languages, L_{PDA}, is the set of all languages that can be described by some PDA:

- $L_{PDA} = \{L : \exists \text{ PDA } M \wedge L[M] = L\}$

We already know $L_{PDA} \supsetneq L_{DFA}$ (why)?

PDA Languages

The Push-Down Automata Languages, L_{PDA}, is the set of all languages that can be described by some PDA:

- $L_{PDA} = \{L : \exists \text{ PDA } M \wedge L[M] = L\}$

We already know $L_{PDA} \supsetneq L_{DFA}$ –
since every DFA is just a PDA that ignores the stack.

PDA Languages

The Push-Down Automata Languages, L_{PDA}, is the set of all languages that can be described by some PDA:

- $L_{PDA} = \{L : \exists \text{ PDA } M \wedge L[M] = L\}$

We already know $L_{PDA} \supsetneq L_{DFA}$ –
since every DFA is just a PDA that ignores the stack.

- $L_{CFG} \subseteq L_{PDA}$?

PDA Languages

The Push-Down Automata Languages, L_{PDA}, is the set of all languages that can be described by some PDA:

- $L_{PDA} = \{L : \exists \text{ PDA } M \wedge L[M] = L\}$

We already know $L_{PDA} \supsetneq L_{DFA}$ –
since every DFA is just a PDA that ignores the stack.

- $L_{CFG} \subseteq L_{PDA}$?
- $L_{PDA} \subseteq L_{CFG}$?

Equivalence Theorem

Theorem: A language is context free if and only if some pushdown automaton accepts it.

This time, both the “if” part and the “only if” part are non-trivial.

$L_{CFG} \subseteq L_{PDA}$

Given any CFG G, can we create a PDA M such that $L[M] = L[G]$?

- Idea: We will first use the stack to derive the input string, and then check to make sure our derivation matches the input

If Part

Theorem: If a language is context free, then some pushdown automaton accepts it.

- Let L be a context-free language.
- We know L has a context-free grammar G.
- on input w, the PDA P should figure out if there is a derivation of w using G.

If Part

Theorem: If a language is context free, then some pushdown automaton accepts it.

- Let L be a context-free language.
- We know L has a context-free grammar G.
- on input w, the PDA P should figure out if there is a derivation of w using G.

Question: How does P figure out which substitution to make?

If Part

Theorem: If a language is context free, then some pushdown automaton accepts it.

- Let L be a context-free language.
- We know L has a context-free grammar G.
- on input w, the PDA P should figure out if there is a derivation of w using G.

Question: How does P figure out which substitution to make?

Answer: It guesses.

CFL Implies PDA

Informally:

- P pushes start variable S on stack

CFL Implies PDA

Informally:

- P pushes start variable S on stack
- keeps making substitutions

CFL Implies PDA

Informally:

- P pushes start variable S on stack
- keeps making substitutions
- when only terminals remain ...

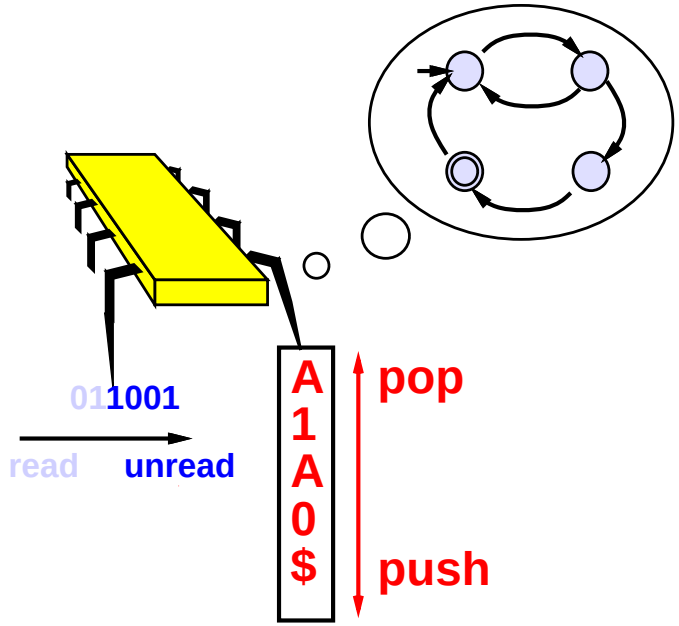
CFL Implies PDA

Informally:

- P pushes start variable S on stack
- keeps making substitutions
- when only terminals remain ...
- tests whether derived string equals input

CFL Implies PDA (cont.)

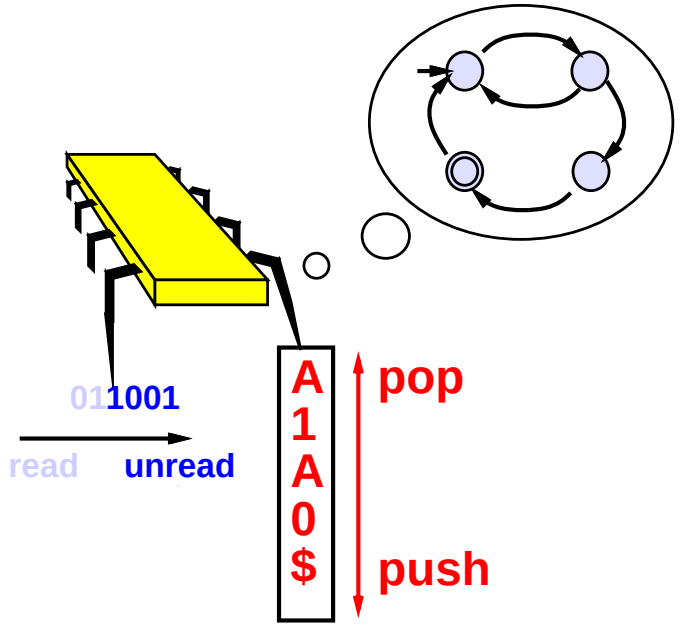
Where do we keep the intermediate string?



intermediate string: 01A1A0

CFL Implies PDA (cont.)

Where do we keep the intermediate string?

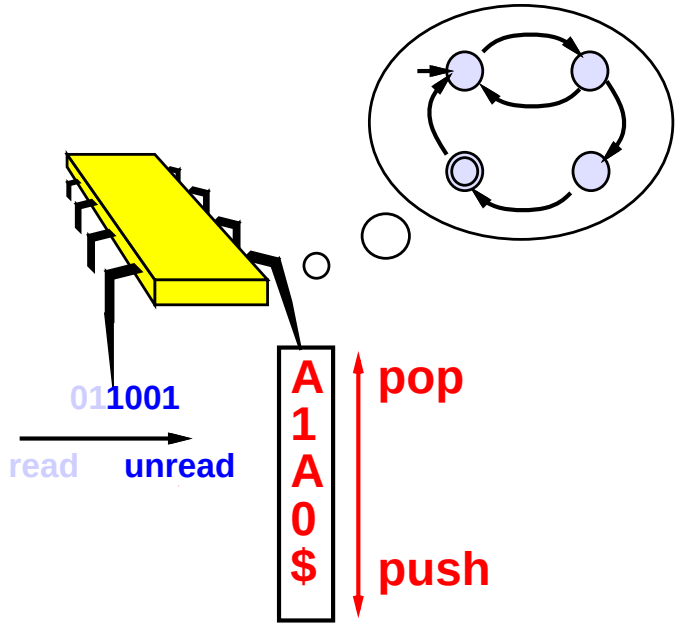


intermediate string: 01A1A0

- can't put it all on the stack

CFL Implies PDA (cont.)

Where do we keep the intermediate string?



intermediate string: 01A1A0

- can't put it all on the stack
- only strings whose first letter is a variable are kept on stack

CFL Implies PDA (cont.)

Informal description:

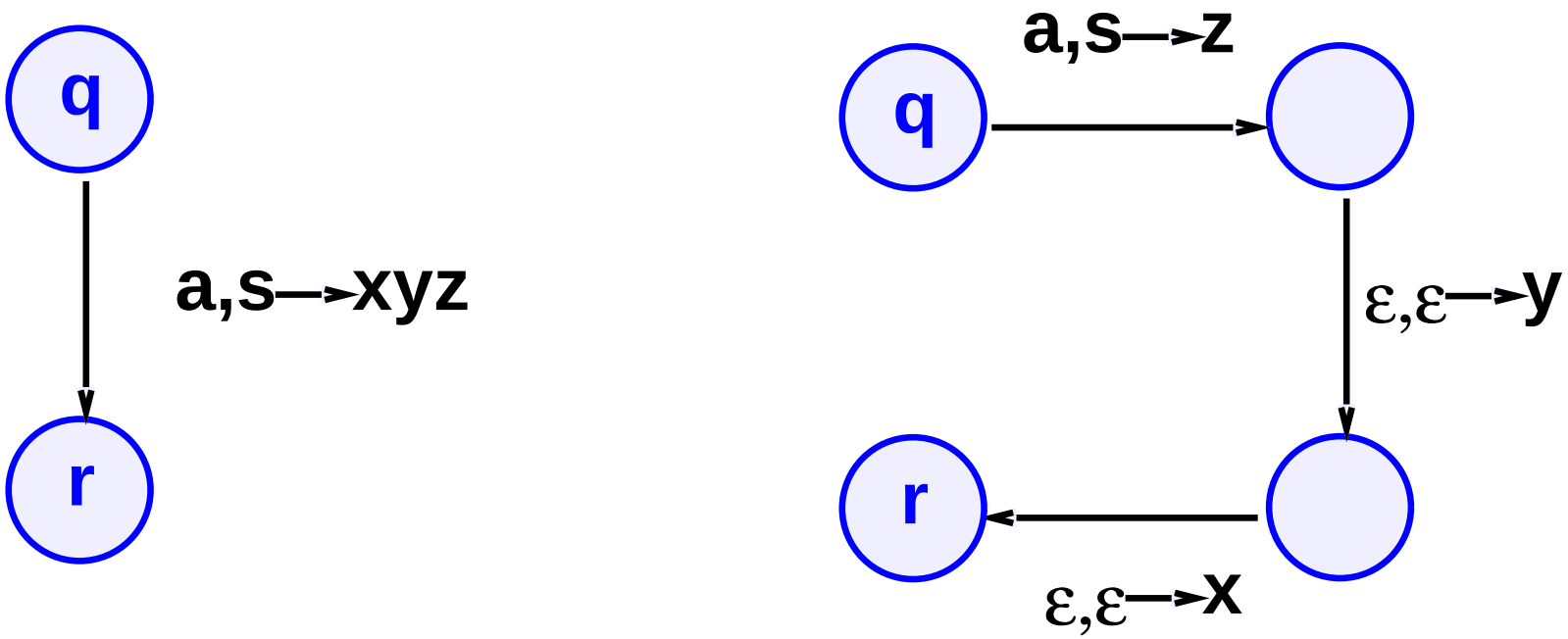
- push $S\$$ on stack
- if top of stack is variable A, non-deterministically select rule and substitute.
- if top of stack is terminal a, read next input and compare. If they differ, reject.

CFL Implies PDA (cont.)

Need shorthand to push entire string onto stack.

$$(r, w) \in \delta(q, a, s)$$

Easy to do by introducing intermediate states.



CFL Implies PDA (cont.)

States of P are

- start state q_s
- accept state q_a
- loop state q_ℓ

Transition Function

Initialize stack

$$\delta(q_s, \varepsilon, \varepsilon) = \{q_\ell, S\$\}$$

Top of stack is variable

$$\delta(q_\ell, \varepsilon, A) = \{(q_\ell, w) | \text{ where } A \rightarrow w \text{ is a rule } \}$$

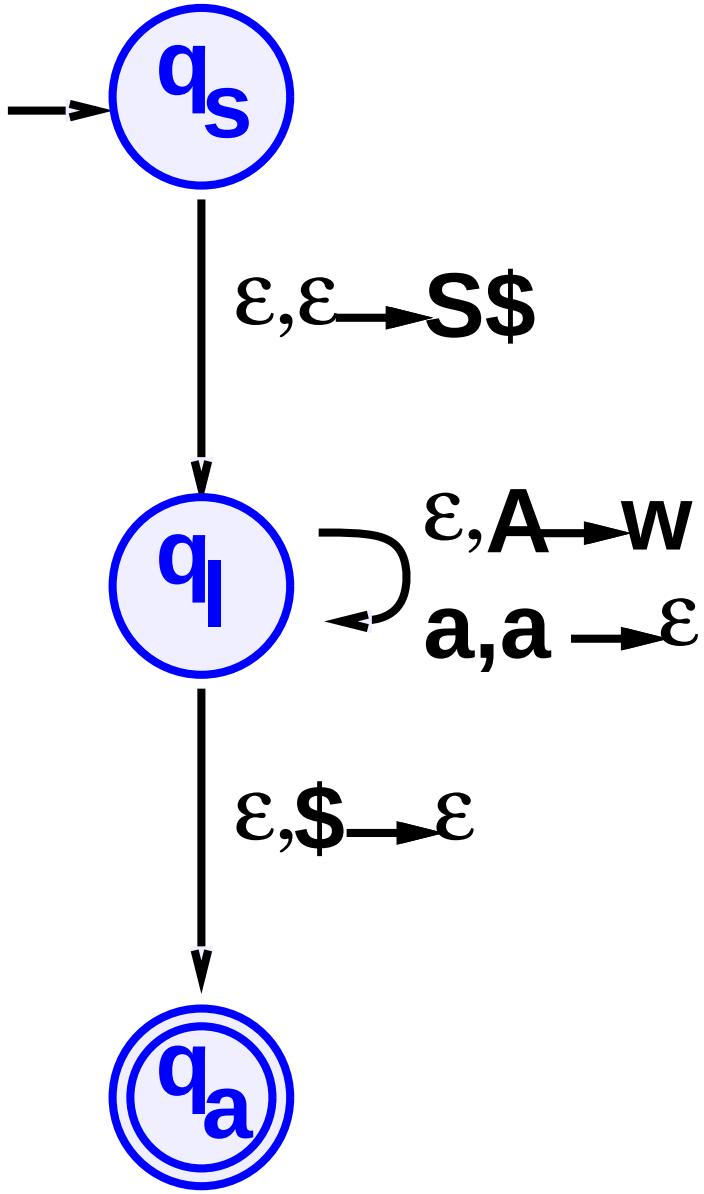
Top of stack is terminal

$$\delta(q_\ell, a, a) = \{(q_\ell, \varepsilon)\}$$

End of Stack

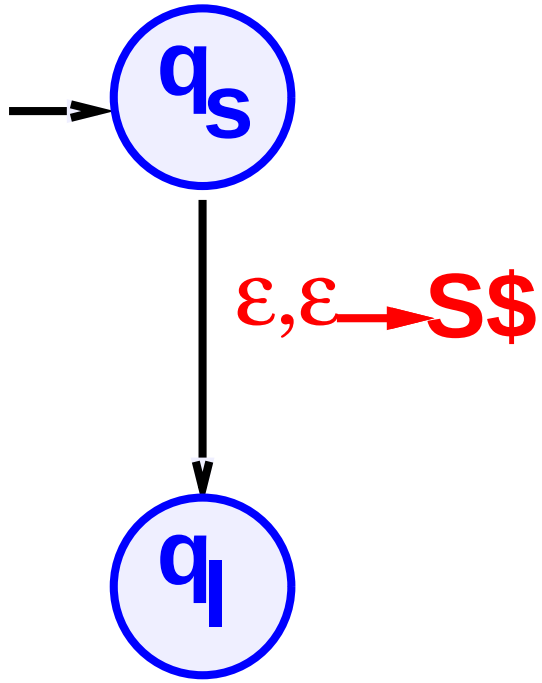
$$\delta(q_\ell, \varepsilon, \$) = \{(q_a, \varepsilon)\}$$

Transition Function



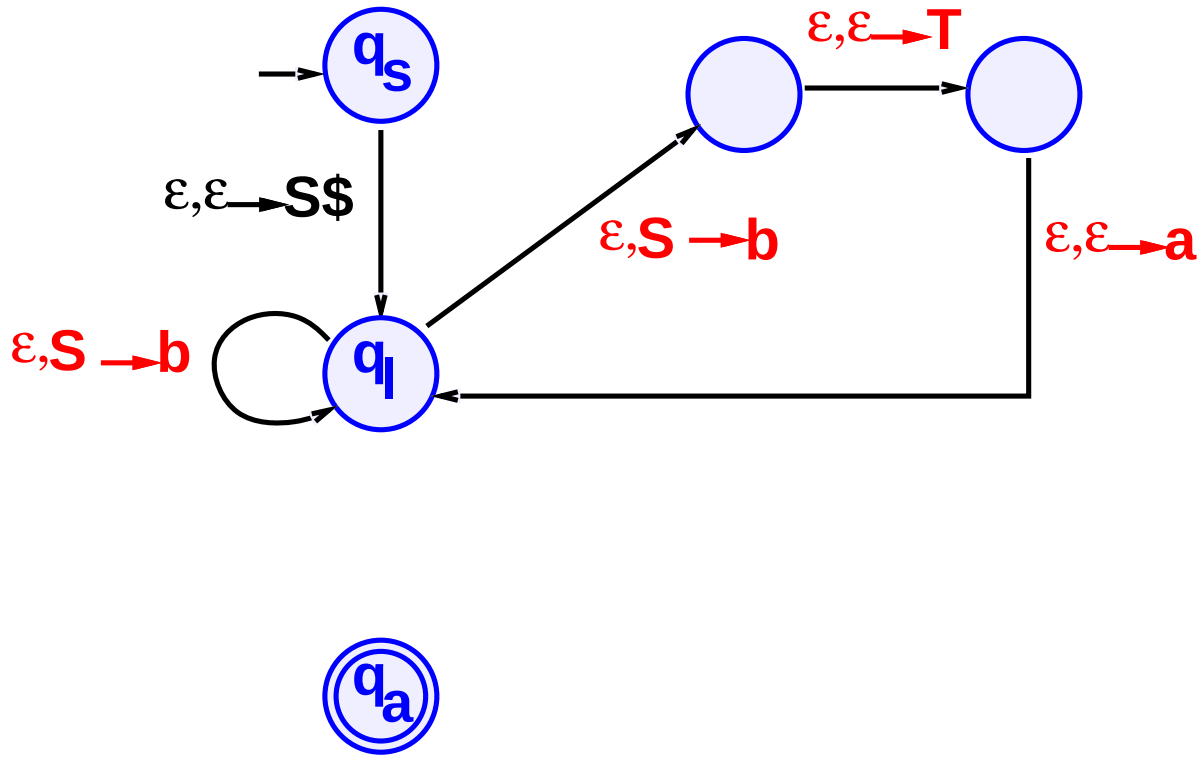
Example

$$
\begin{aligned}
S &\rightarrow aTb|b \\
T &\rightarrow Ta|\varepsilon
\end{aligned}
$$



Example

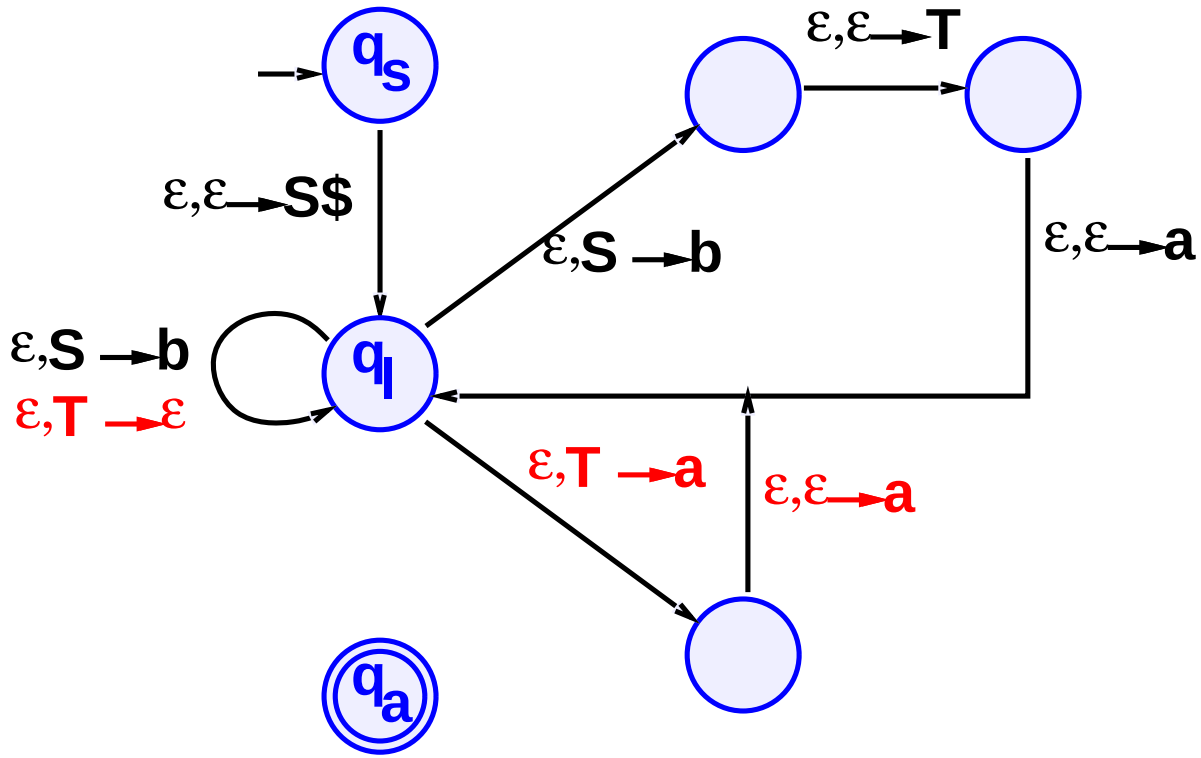
$$
\begin{aligned}
S &\rightarrow aTb|b \\
T &\rightarrow Ta|\varepsilon
\end{aligned}
$$



Rules for S

Example

$$
\begin{aligned}
S &\rightarrow aTb|b \\
T &\rightarrow Ta|\varepsilon
\end{aligned}
$$

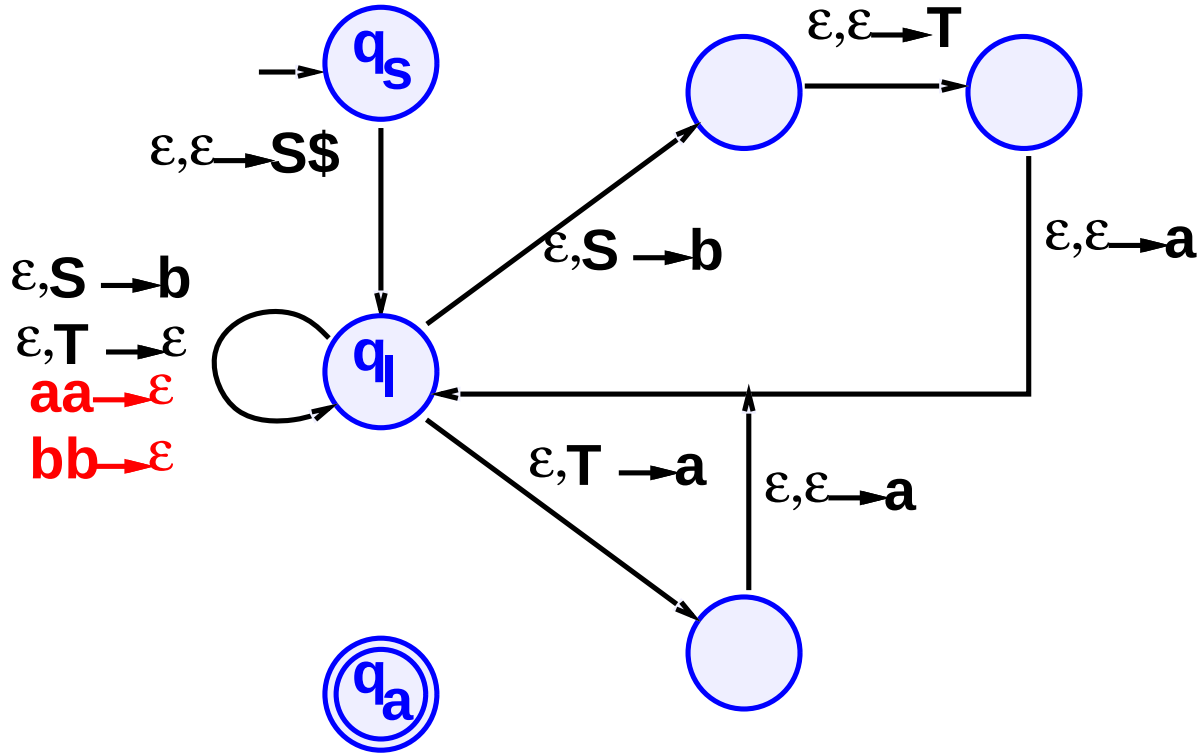


Rules for T

Example

$$
\begin{aligned}
S &\rightarrow aTb|b \\
T &\rightarrow Ta|\varepsilon
\end{aligned}
$$

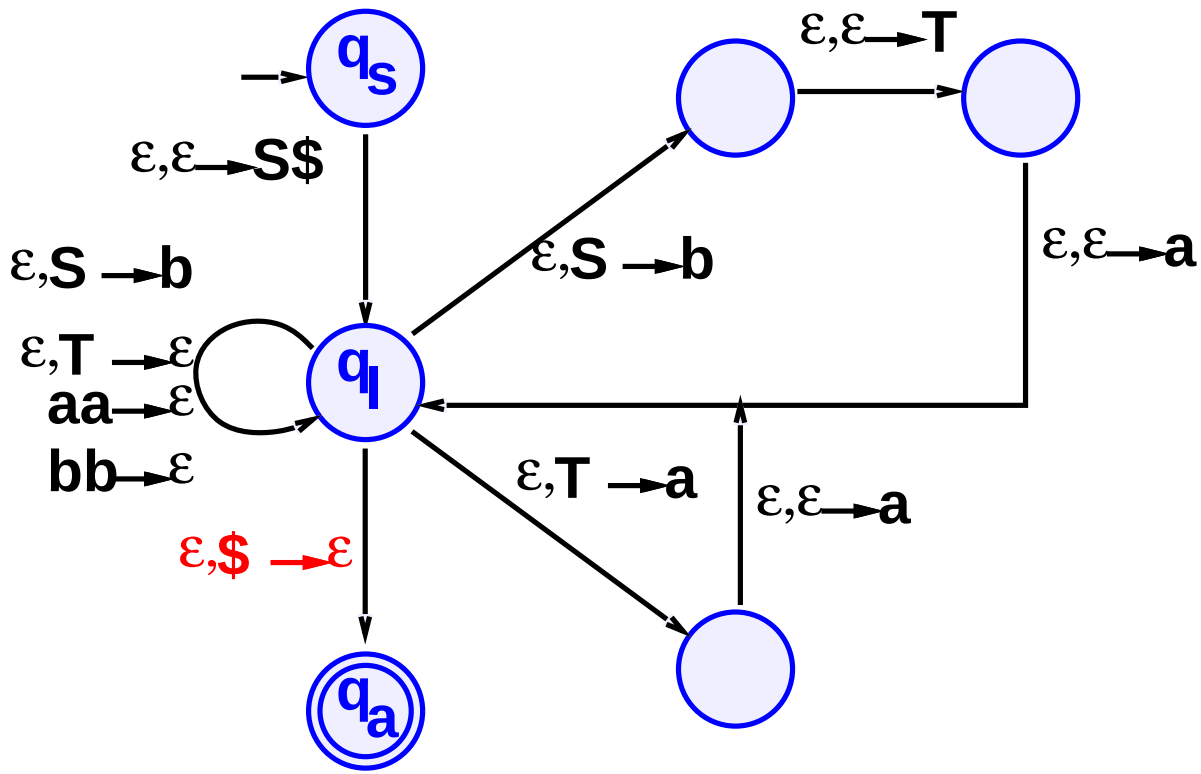
Rules for terminals



Example

$$S \rightarrow aTb|b$$
$$T \rightarrow Ta|\varepsilon$$

Termination:



Only If Part

Theorem: If a PDA accepts a language, L, then L is context-free.

Only If Part

Theorem: If a PDA accepts a language, L, then L is context-free.

Given a pushdown automata P, we want to design a context-free grammar G that "simulates" it:

The strings generated by G should be exactly those that cause P to go from the start state to an accepting state.

Only If Part

Theorem: If a PDA accepts a language, L, then L is context-free.

Only If Part

Theorem: If a PDA accepts a language, L, then L is context-free.

Proof: After constructing the grammar G , should prove it generates exactly the same language accepted by the PDA.
This is done by induction on the length of any computation of P on any input string x.

Only If Part

Theorem: If a PDA accepts a language, L, then L is context-free.

Proof: After constructing the grammar G , should prove it generates exactly the same language accepted by the PDA.
This is done by induction on the length of any computation of P on any input string x.

The induction argument is a bit lengthy and tedious, and we'll skip it.

Only If Part

Theorem: If a PDA accepts a language, L, then L is context-free.

Proof: After constructing the grammar G , should prove it generates exactly the same language accepted by the PDA.
This is done by induction on the length of any computation of P on any input string x.

The induction argument is a bit lengthy and tedious, and we'll skip it.

Only If Part

Theorem: If a PDA accepts a language, L, then L is context-free.

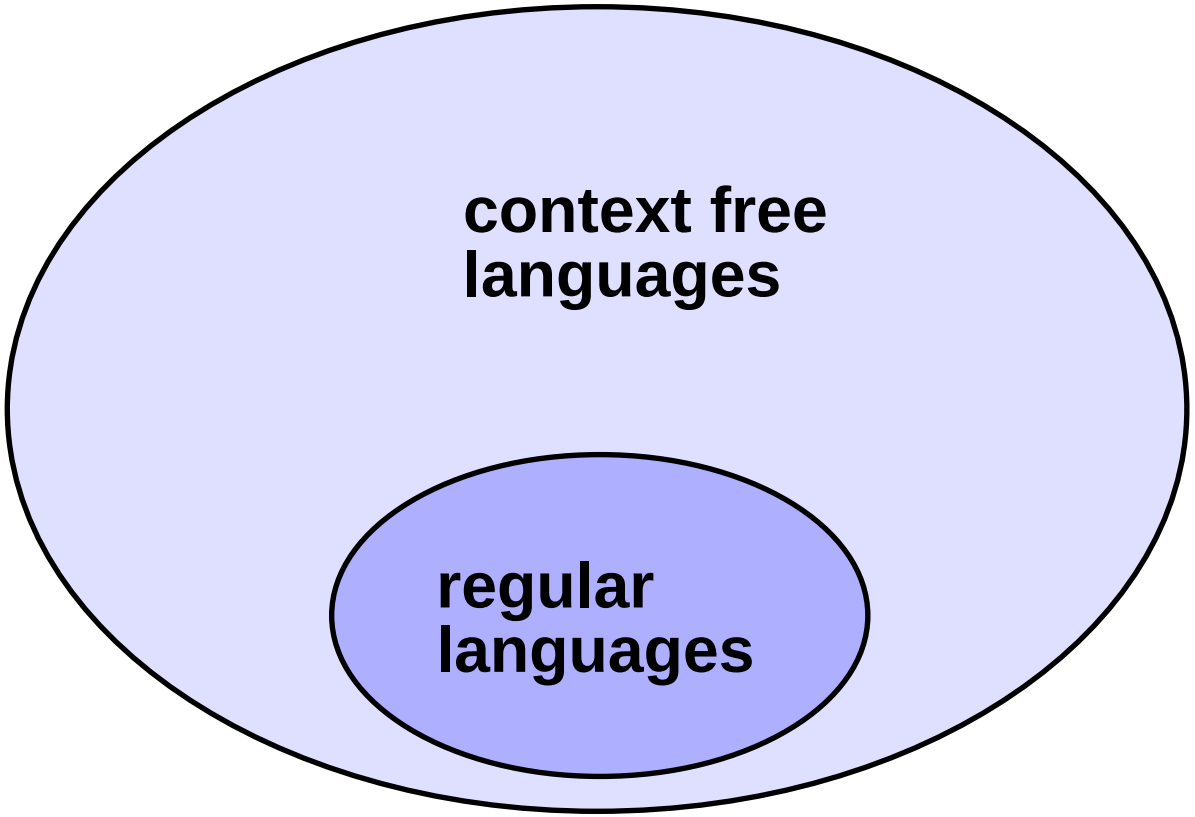
Proof: After constructing the grammar G , should prove it generates exactly the same language accepted by the PDA.
This is done by induction on the length of any computation of P on any input string x.

The induction argument is a bit lengthy and tedious, and we'll skip it.

Diehards are welcome to consult pp. 106–114 in Sipser's book, and/or last year's slides (fall 2003/4).

Corollary

Every regular language is context-free.



Non-Context-Free Languages

Non-Context-Free Languages

- pumping lemma for finite automata is our (almost) only tool for showing that languages are not regular.

Non-Context-Free Languages

- pumping lemma for finite automata is our (almost) only tool for showing that languages are not regular.
- we will now do the same for context-free languages.

Non-Context-Free Languages

- pumping lemma for finite automata is our (almost) only tool for showing that languages are not regular.
- we will now do the same for context-free languages.
- it's slightly more complicated …

Pumping Lemma for CFL

Also known as the $uvxyz$ Theorem.

Theorem: If A is a CFL , there is an ℓ (critical length), such that if $s \in A$ and $|s| \geq \ell$, then $s = uvxyz$ where

- for every $i \geq 0$, $uv^ixy^iz \in A$

Pumping Lemma for CFL

Also known as the $uvxyz$ Theorem.

Theorem: If A is a CFL , there is an ℓ (critical length), such that if $s \in A$ and $|s| \geq \ell$, then $s = uvxyz$ where

- for every $i \geq 0$, $uv^ixy^iz \in A$
- $|vy| > 0$, (non-triviality)

Pumping Lemma for CFL

Also known as the $uvxyz$ Theorem.

Theorem: If A is a CFL , there is an ℓ (critical length), such that if $s \in A$ and $|s| \geq \ell$, then $s = uvxyz$ where

- for every $i \geq 0$, $uv^ixy^iz \in A$
- $|vy| > 0$, (non-triviality)
- $|vxy| \leq \ell$.

Basic Intuition

Let A be a CFL and G its CFG.

Let s be a "very long" string in A.

Then s has a "tall" parse tree.

And some root-to-leaf path must repeat a symbol.

ahmmm,. . . why is that so?

Basic Intuition

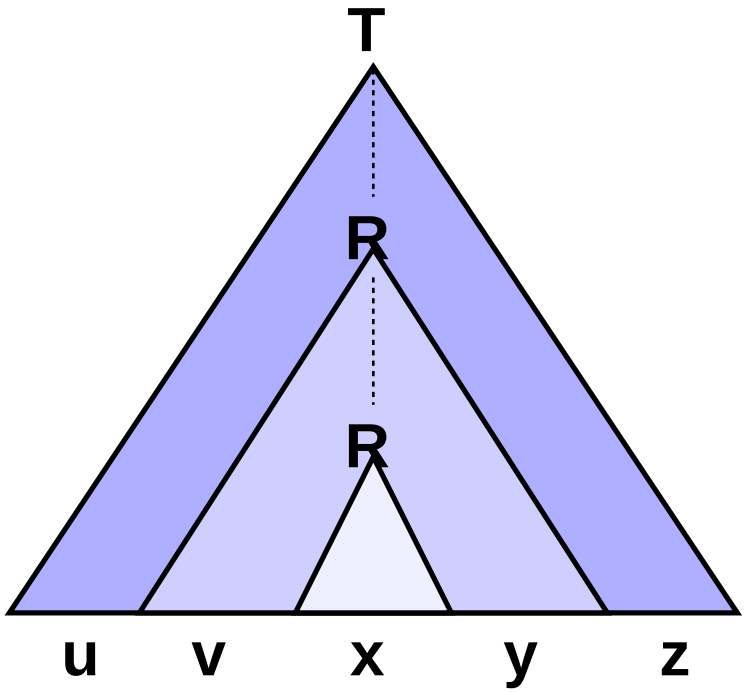
Let A be a CFL and G its CFG.

Let s be a "very long" string in A.

Then s has a "tall" parse tree.

And some root-to-leaf path must repeat a symbol.

ahmmm,. . . why is that so?



Proof

- let G be a CFG for CFL A.

Proof

- let G be a CFG for CFL A.
- let c be the max number of symbols in right-hand-side of any rule.

Proof

- let G be a CFG for CFL A.
- let c be the max number of symbols in right-hand-side of any rule.
- no node in parse tree has $> c$ children.

Proof

- let G be a CFG for CFL A.
- let c be the max number of symbols in right-hand-side of any rule.
- no node in parse tree has $> c$ children.
- at depth d, no more than c^d leaves.

Proof

- let G be a CFG for CFL A.
- let c be the max number of symbols in right-hand-side of any rule.
- no node in parse tree has $> c$ children.
- at depth d, no more than c^d leaves.
- let $|V|$ be the number of variables in G.

Proof

- let G be a CFG for CFL A.
- let c be the max number of symbols in right-hand-side of any rule.
- no node in parse tree has $> c$ children.
- at depth d, no more than c^d leaves.
- let $|V|$ be the number of variables in G.
- let $\ell = c^{|V|+2}$.

Proof (2)

- let s be a string where $|s| \geq \ell$

Proof (2)

- let s be a string where $|s| \geq \ell$
- let τ be parse tree for s with fewest nodes

Proof (2)

- let s be a string where $|s| \geq \ell$
- let τ be parse tree for s with fewest nodes
- τ has height $\geq |V| + 2$

Proof (2)

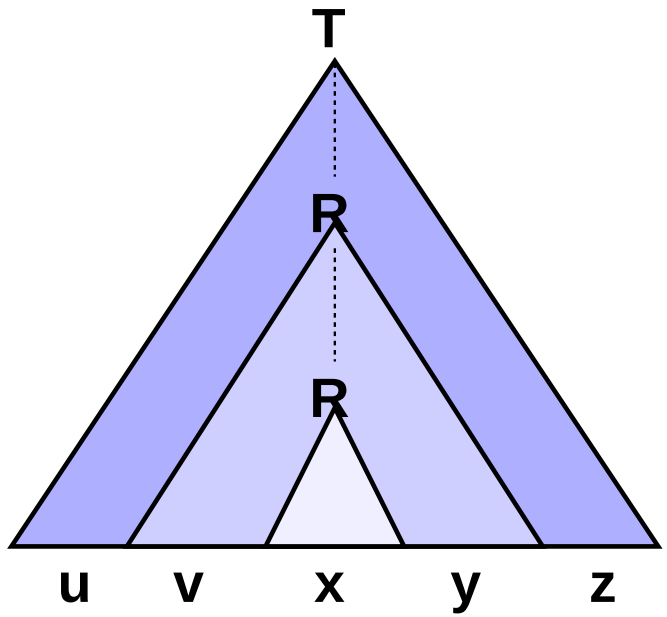
- let s be a string where $|s| \geq \ell$
- let τ be parse tree for s with fewest nodes
- τ has height $\geq |V| + 2$
- some path in τ has length $\geq |V| + 2$

Proof (2)

- let s be a string where $|s| \geq \ell$
- let τ be parse tree for s with fewest nodes
- τ has height $\geq |V| + 2$
- some path in τ has length $\geq |V| + 2$
- that path repeats a variable R

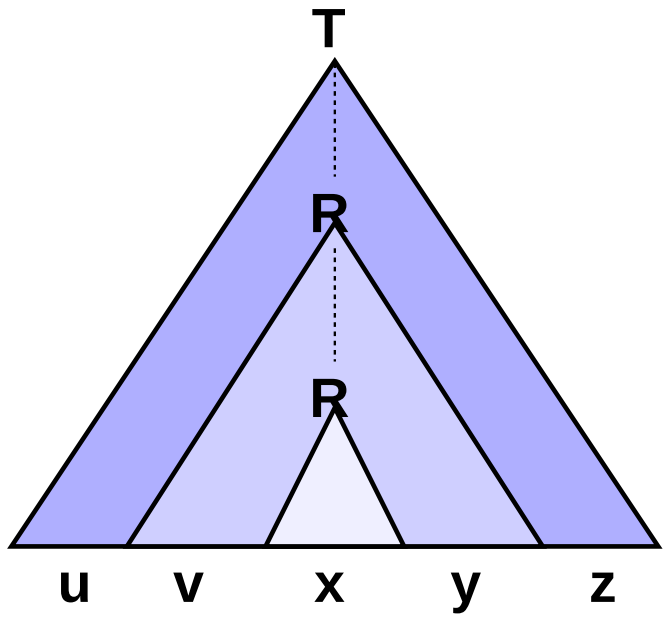
Proof (3)

Split $s = uvxyz$



Proof (3)

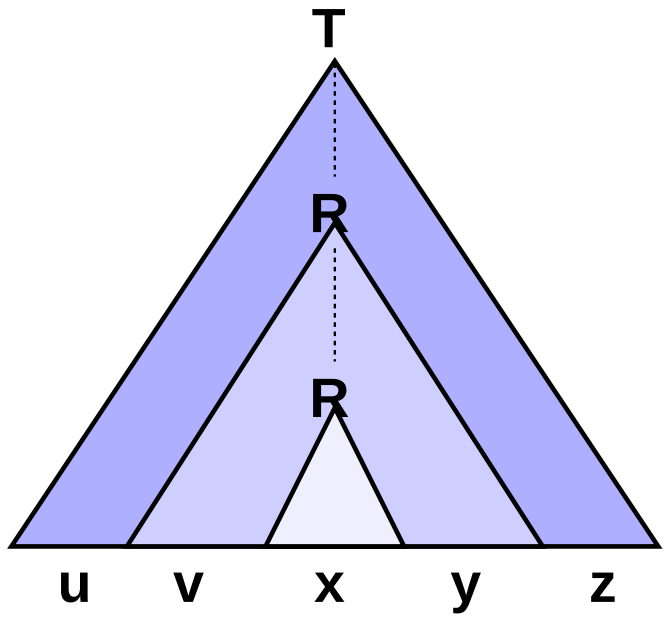
Split $s = uvxyz$



- each occurrence of R produces a string

Proof (3)

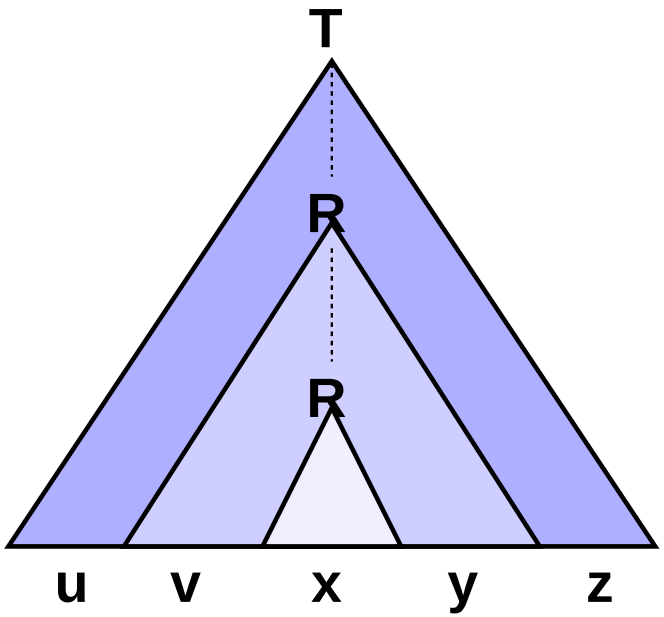
Split $s = uvxyz$



- each occurrence of R produces a string
- upper produces string vxy

Proof (3)

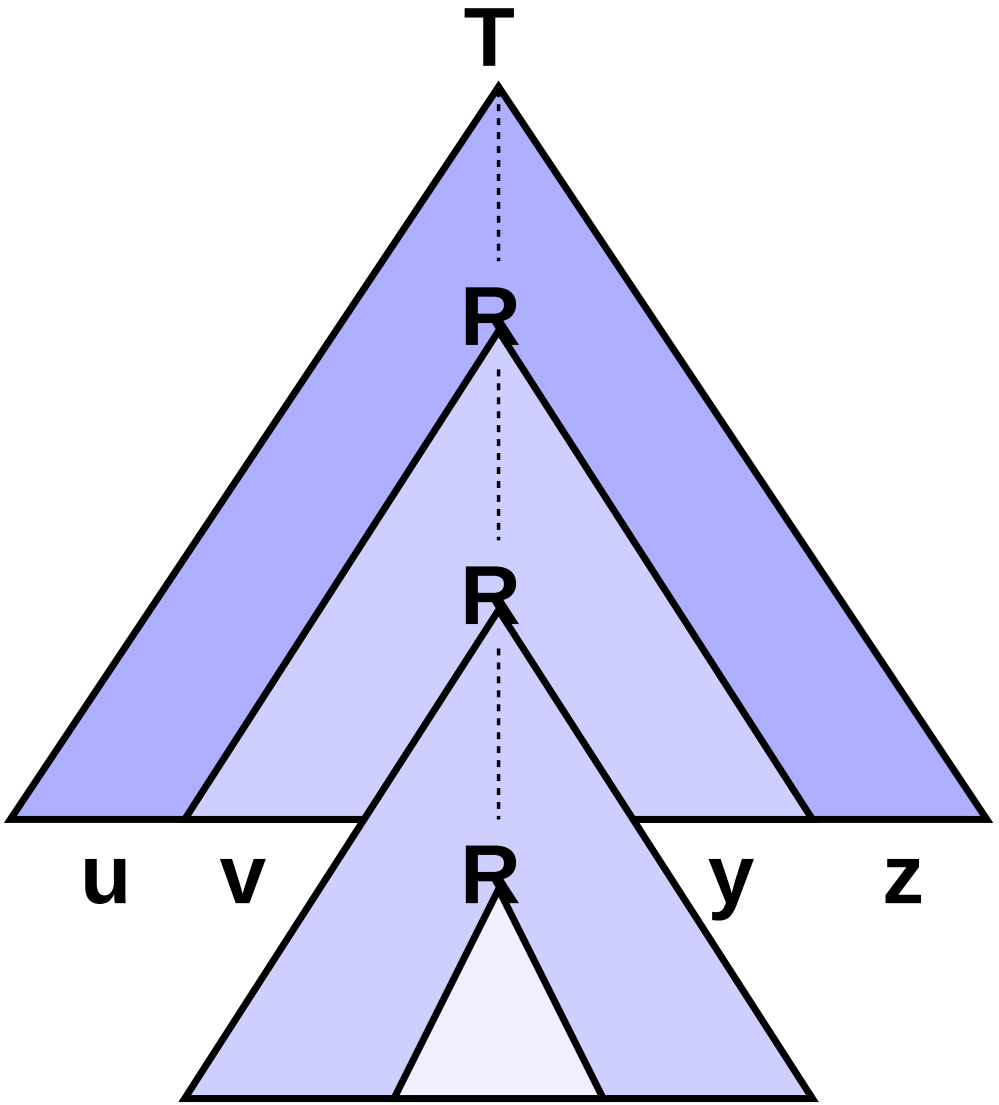
Split $s = uvxyz$



- each occurrence of R produces a string
- upper produces string vxy
- lower produces string x

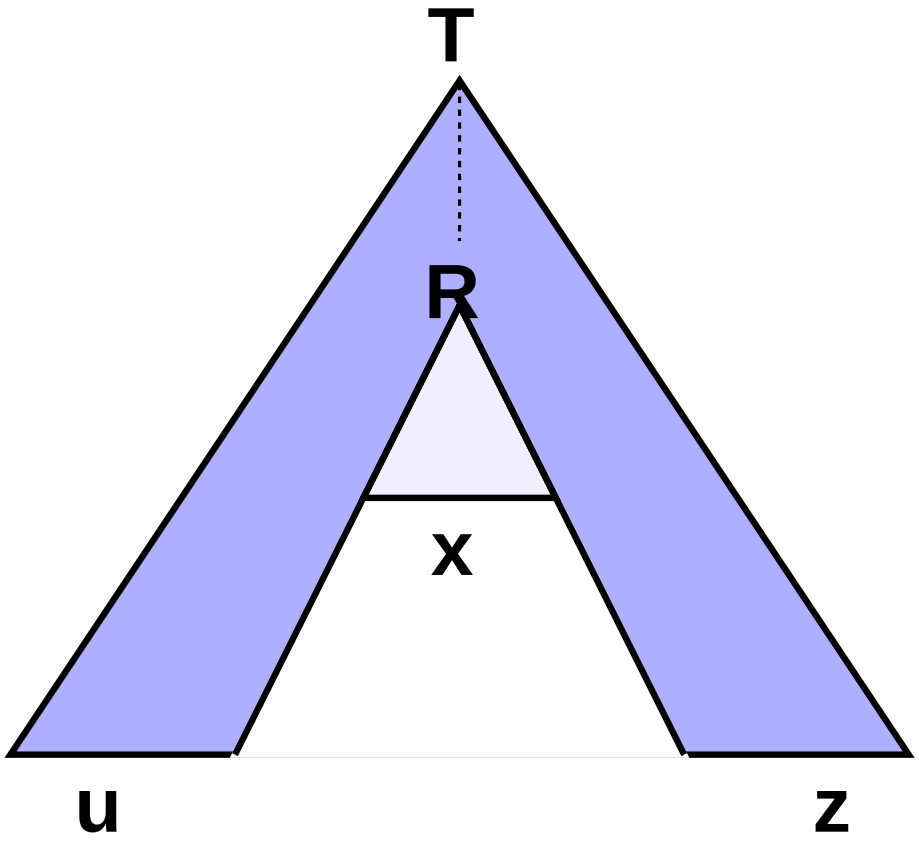
Proof

Replacing smaller by larger yields uv^ixy^iz, for $i > 0$.



Proof (4)

Replacing larger by smaller yields uxz.



Together, they establish:

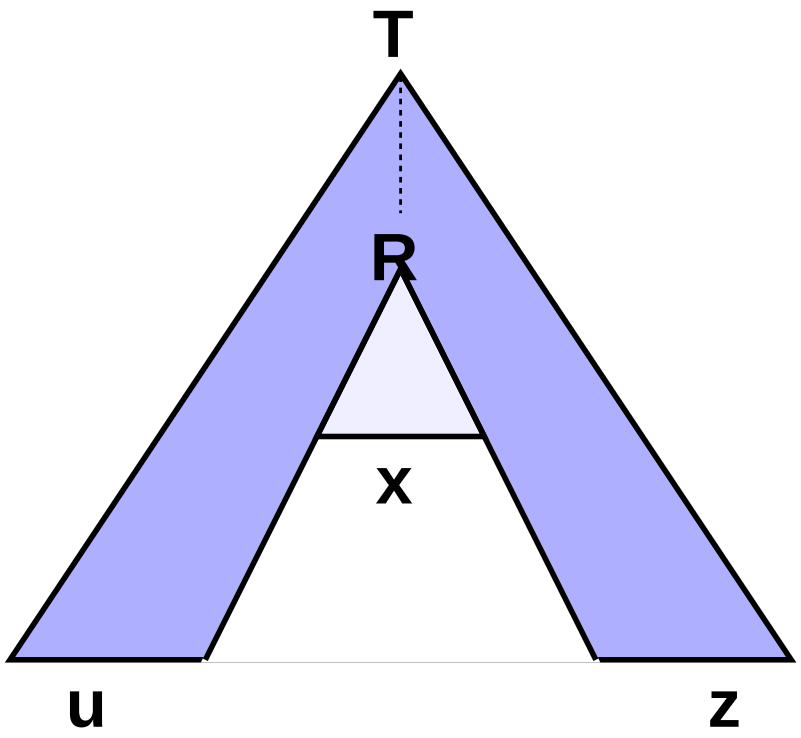
- for $i \geq 0$, $uv^ixy^iz \in A$

Proof (5)

Next condition is:

- $|vy| > 0$

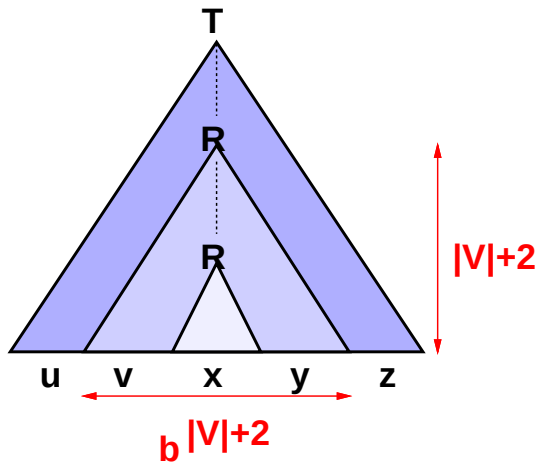
If v and y are both ε, then



is a parse tree for s with fewer nodes.

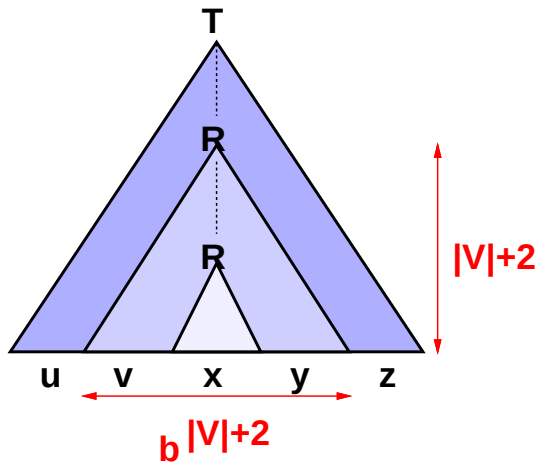
Proof (6)

Final condition is $|vxy| \leq \ell$:



Proof (6)

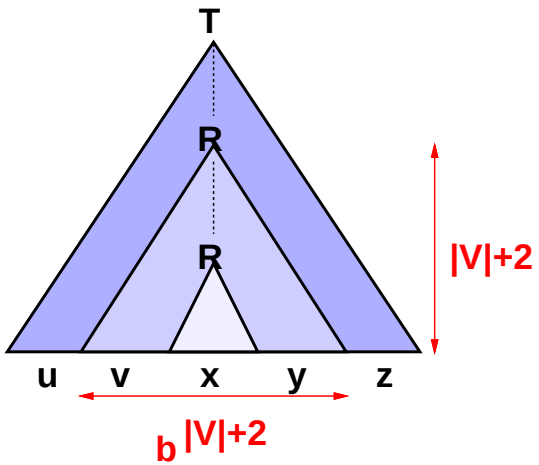
Final condition is $|vxy| \leq \ell$:



- the upper occurrence of R generates vxy.

Proof (6)

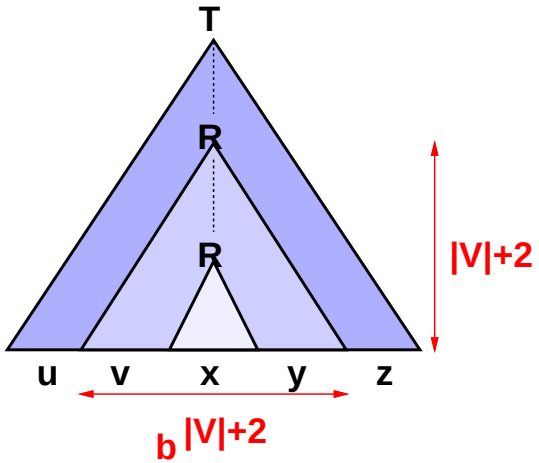
Final condition is $|vxy| \leq \ell$:



- the upper occurrence of R generates vxy.
- can choose symbols such that both occurrences of R lie in bottom $|V| + 1$ variables on path.

Proof (6)

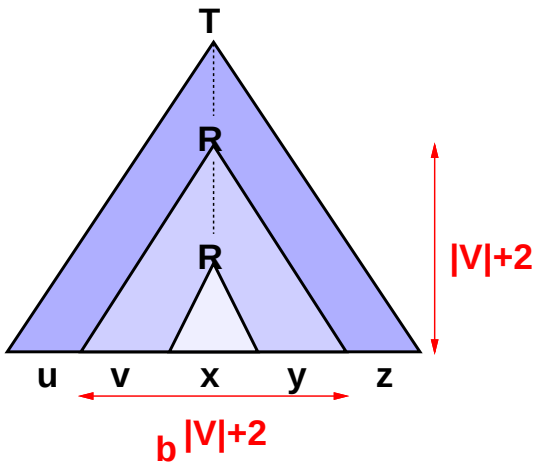
Final condition is $|vxy| \leq \ell$:



- the upper occurrence of R generates vxy.
- can choose symbols such that both occurrences of R lie in bottom $|V| + 1$ variables on path.
- subtree where R generates vxy is $\leq |V| + 2$ high.

Proof (6)

Final condition is $|vxy| \leq \ell$:



- the upper occurrence of R generates vxy.
- can choose symbols such that both occurrences of R lie in bottom $|V| + 1$ variables on path.
- subtree where R generates vxy is $\leq |V| + 2$ high.
- strings by subtree at most $b^{|V|+2} = \ell$ long.

Example

Theorem: $B = \{a^n b^n c^n\}$ is not a CFL.
Proof: By contradiction.

Let ℓ be the critical length.

Consider $s = a^\ell b^\ell c^\ell$.
If $s = uvxyz$, neither v nor y can contain

- both a's and b's, or
- both b's and c's, because

uv^2xy^2z would have out-of-order symbols.
But if v and y contain only one symbol, then uv^2xy^2z has an imbalance!

Example (2)

The language $C = \{a^i b^j c^k | 0 \leq i \leq j \leq k\}$ is not context-free.
Let ℓ be the critical length, and $s = a^\ell b^\ell c^\ell$.

Let $s = uvxyz$

- neither v nor y contains contains two distinct symbols, because uv^2xy^2z would have out-of-order symbols.

Example (2)

The language $C = \{a^i b^j c^k | 0 \leq i \leq j \leq k\}$ is not context-free.
Let ℓ be the critical length, and $s = a^\ell b^\ell c^\ell$.

Let $s = uvxyz$

- neither v nor y contains contains two distinct symbols, because uv^2xy^2z would have out-of-order symbols.
- vxy not all b's (why?)

Example (2)

The language $C = \{a^i b^j c^k | 0 \leq i \leq j \leq k\}$ is not context-free.
Let ℓ be the critical length, and $s = a^\ell b^\ell c^\ell$.

Let $s = uvxyz$

- neither v nor y contains contains two distinct symbols, because uv^2xy^2z would have out-of-order symbols.
- vxy not all b's (why?)
- $vxy < \ell$, so either

Example (2)

The language $C = \{a^i b^j c^k | 0 \leq i \leq j \leq k\}$ is not context-free.
Let ℓ be the critical length, and $s = a^\ell b^\ell c^\ell$.

Let $s = uvxyz$

- neither v nor y contains contains two distinct symbols, because uv^2xy^2z would have out-of-order symbols.
- vxy not all b's (why?)
- $vxy < \ell$, so either
  - v contains only a's and y contains only b's, but then uv^2xy^2z has too few c's.

Example (2)

The language $C = \{a^i b^j c^k | 0 \leq i \leq j \leq k\}$ is not context-free.
Let ℓ be the critical length, and $s = a^\ell b^\ell c^\ell$.

Let $s = uvxyz$

- neither v nor y contains contains two distinct symbols, because uv^2xy^2z would have out-of-order symbols.
- vxy not all b's (why?)
- $vxy < \ell$, so either
  - v contains only a's and y contains only b's, but then uv^2xy^2z has too few c's.
  - v contains only b's and y contains only c's. but then uv^0xy^0z has too many a's.

Example (3)

The language $D = \{ww | w \in \{0,1\}^*\}$ is not context-free.
Let $s = 0^\ell 1^\ell 0^\ell 1^\ell$ As before, suppose $s = uvxyz$.
Recall that $|vxy| \leq \ell$.

- if vxy is in the first half of s, uv^2xy^2z moves a 1 into the first position in second half.

Example (3)

The language $D = \{ww | w \in \{0,1\}^*\}$ is not context-free.
Let $s = 0^\ell 1^\ell 0^\ell 1^\ell$ As before, suppose $s = uvxyz$.
Recall that $|vxy| \leq \ell$.

- if vxy is in the first half of s, uv^2xy^2z moves a 1 into the first position in second half.
- if vxy is in the second half, uv^2xy^2z moves a 0 into the last position in first half.

Example (3)

The language $D = \{ww | w \in \{0,1\}^*\}$ is not context-free.
Let $s = 0^\ell 1^\ell 0^\ell 1^\ell$ As before, suppose $s = uvxyz$.
Recall that $|vxy| \leq \ell$.

- if vxy is in the first half of s, uv^2xy^2z moves a 1 into the first position in second half.
- if vxy is in the second half, uv^2xy^2z moves a 0 into the last position in first half.
- if vxy straddles the midpoint, then pumping *down* to uxz yields $0^\ell 1^i 0^j 1^\ell$ where i and j cannot both be ℓ.