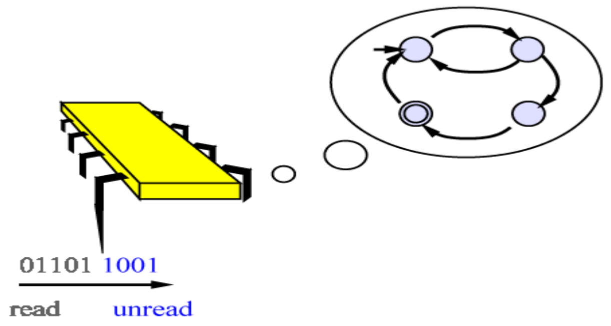
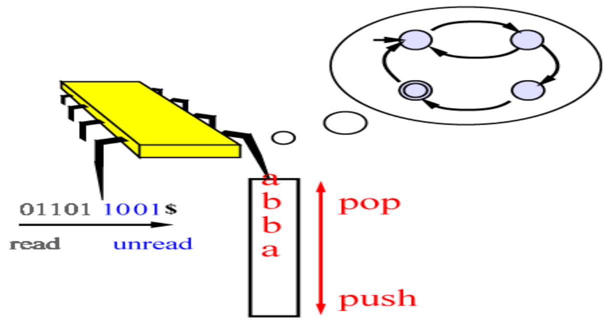
Computational Models - Lecture 4

- Context Free Grammars (including the special case of linear grammars)
- Pumping Lemma for context free languages
- Non context free languages: Examples
- Push Down Automata (PDA)
- Chomsky Normal Form





- Sipser's book, 2.1, 2.2 & 2.3

Short Overview of the Course (so far)

So far we saw

- finite automata,
- regular languages,
- regular expressions,
- Myhill-Nerode theorem
- pumping lemma for regular languages.

We now introduce stronger machines and languages with more expressive power:

- pushdown automata,
- context-free languages,
- context-free grammars,
- pumping lemma for context-free languages.

Formal Definitions

A context-free grammar is a 4-tuple (V, Σ, R, S) where

- V is a finite set of variables,
- Σ is a finite set of terminals,
- R is a finite set of rules: each rule has a variable on the left hand side, and a finite string of variables and terminals on the right hand side.
- S is the start symbol.

Formal Definitions

- If u and v are strings of variables and terminals,
- and $A \to w$ is a rule of the grammar, then
- we say uAv yields uwv, written $uAv \Rightarrow uwv$.

We write $u \overset{*}{\Rightarrow} v$ if $u = v$ or

$$u \Rightarrow u_1 \Rightarrow \ldots \Rightarrow u_k \Rightarrow v.$$

for some sequence $u_1, u_2, \ldots, u_k$.

Definition: The language of the grammar is
$\left\{ w \in \Sigma^* \mid S \overset{*}{\Rightarrow} w \right\}$.

Example

Consider $G_4 = (V, \{a, b\}, R, S)$.

R (Rules): $S \rightarrow aSb \mid SS \mid \varepsilon$.

Some words in the language: $aabb$, $aababb$.

Q.: What is this language?

Hint: Think of parentheses.

Arithmetic Example

Consider (V, Σ, R, E) where

- $V = \{E, T, F\}$
- $\Sigma = \{a, +, \times, (,)\}$

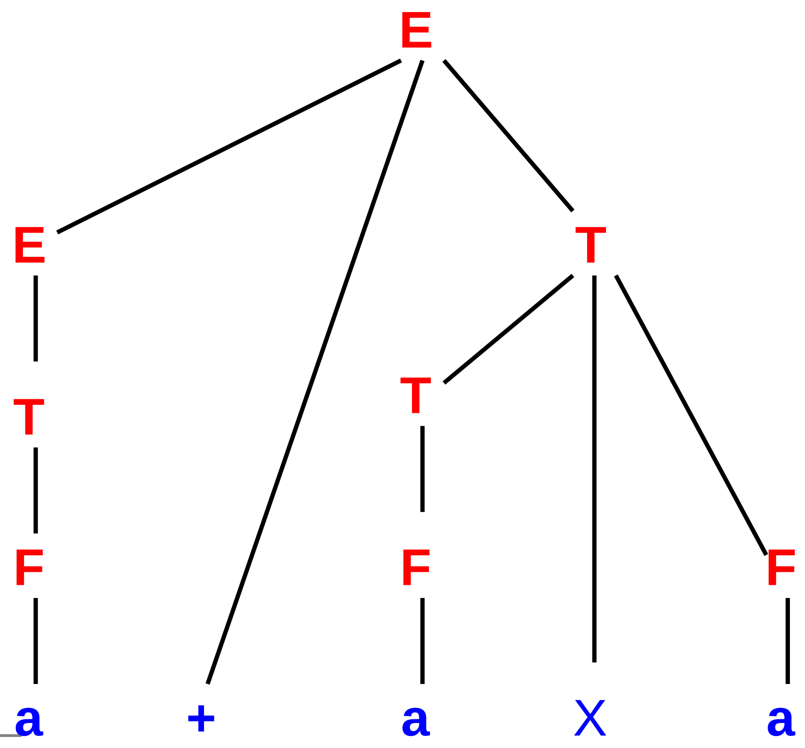
Rules:
$$\begin{array}{lcl} E & \rightarrow & E + T \mid T \\ T & \rightarrow & T \times F \mid F \\ F & \rightarrow & (E) \mid a \end{array}$$

Some strings generated by the grammar:
$a + a \times a$ and $(a + a) \times a$.
What is the language of this grammar?

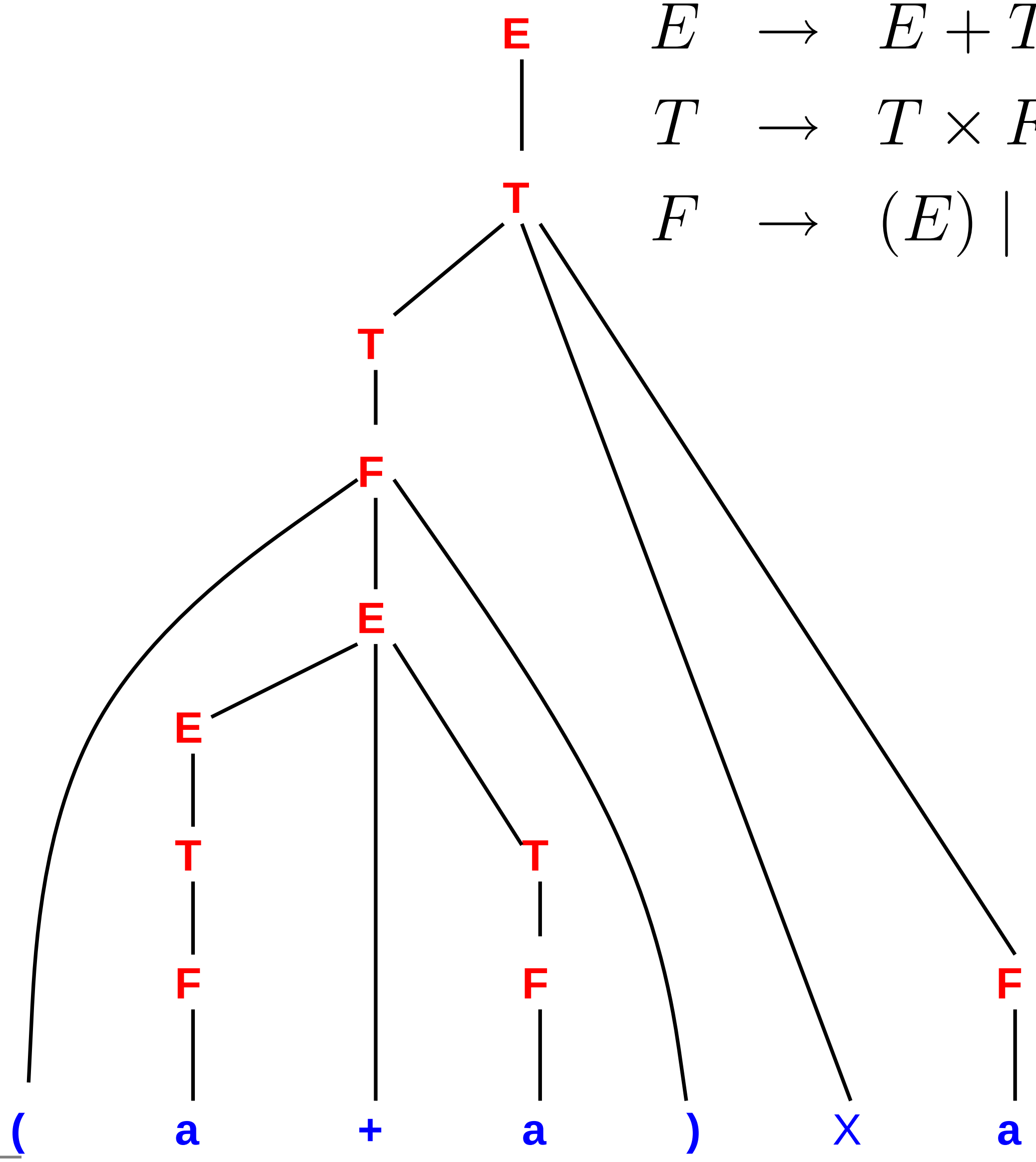
Hint: arithmetic expressions.
$E =$ expression, $T =$ term, $F =$ factor.

Parse Tree for $a + a \times a$

$$\begin{array}{rcl} E & \longrightarrow & E + T \mid T \\ T & \longrightarrow & T \times F \mid F \\ F & \longrightarrow & (E) \mid a \end{array}$$



Parse Tree for $(a + a) \times a$



$$
\begin{array}{lcl}
E & \longrightarrow & E + T \mid T \\
T & \longrightarrow & T \times F \mid F \\
F & \longrightarrow & (E) \mid a
\end{array}
$$

Designing Context-Free Grammars

No recipe in general, but few rules-of-thumb

- If CFG is the union of several CFGs, rename variables (not terminals) so they are disjoint, and add new rule $S \to S_1 \mid S_2 \mid \ldots \mid S_i$.
- For a regular language, grammar "follows" a DFA for the language. For initial state q_0, make R_0 the start variable. For state transition $\delta(q_i, a) = q_j$ add rule $R_i \to aR_j$ to grammar. For each final state q_f, add rule $R_f \to \varepsilon$ to grammar. This is called a linear grammar.
- For languages (like $\{0^n\#1^n | n \geq 0\}$), with linked substrings, a rule of form $R \to uRv$ is helpful to force desired relation between substrings.

Closure Properties

- Regular languages are closed under
  - union
  - concatenation
  - star
- Context-Free Languages are closed under
  - union : $S \to S_1 \mid S_2$
  - concatenation $S \to S_1 S_2$
  - star $S \to \varepsilon \mid SS$

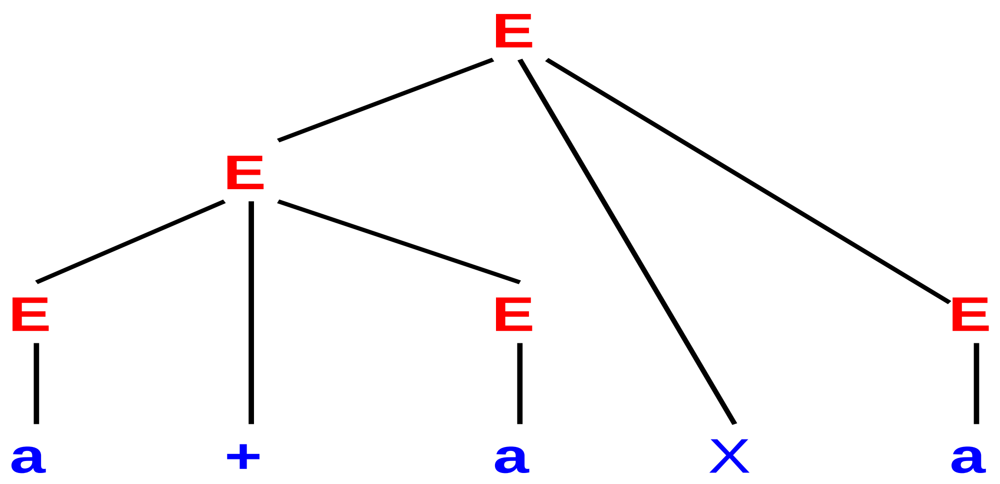
More Closure Properties

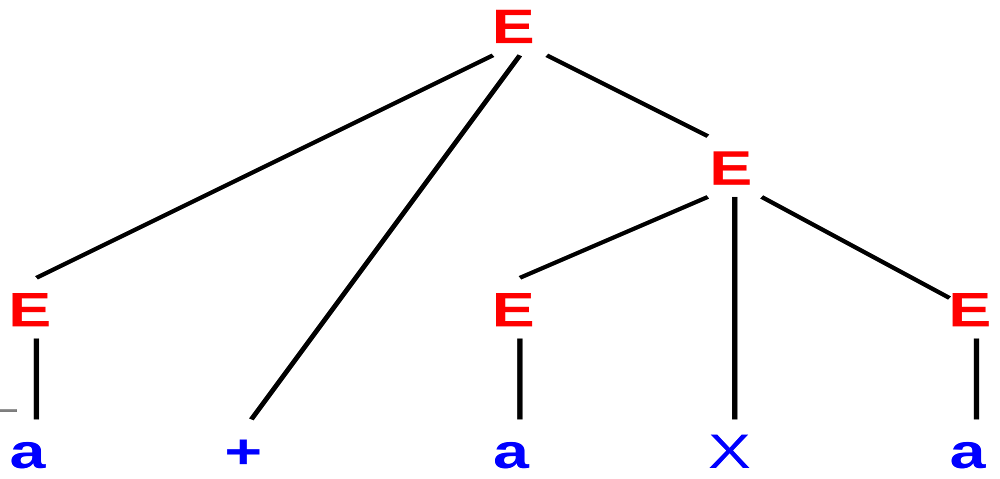
- Regular languages are also closed under
  - complement (replace accept/non-accept states of DFA)
  - intersection (using $L_1 \cap L_2 = \overline{\overline{L_1} \cup \overline{L_2}}$, or a direct construction using Cartesian product).
- What about closure under complement and intersection of context-free languages?
- Absolutely not clear (at this point) ...

Ambiguity

Grammar: $E \rightarrow E{+}E \mid E{\times}E \mid (E) \mid a$

$a + a \times a$ is a word in the language, with two different derivations.





Ambiguity

We say that a string w is derived ambiguously from grammar G if w has two or more parse trees (derivations) that generate it from G.

Ambiguity is usually not only a syntactic notion but also semantic, implying multiple meanings for the same string. Think of $a + a \times a$ from last grammar.

It is sometime possible to eliminate ambiguity by finding a different context free grammar generating the same language. For the grammar above, the unambiguous grammar from slide 6 generates a similar, “semantically correct”, but not identical, language (e.g. the string $(a + a) \times a$).

Some languages, *e.g.* $\{1^i 2^j 3^k \mid i = j$ or $j = k\}$ are inherently ambiguous.

Non-Context-Free Languages

- The pumping lemma for finite automata and Myhill-Nerode theorem are our tools for showing that languages are not regular.
- We do not have a simple Myhill-Nerode type theorem.
- However, will now show a similar pumping lemma for context-free languages.
- It is 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 $i \geq 0$, $uv^ixy^iz \in A$
- $|vy| > 0$, (non-triviality)
- $|vxy| \leq \ell$.

Basic Intuition

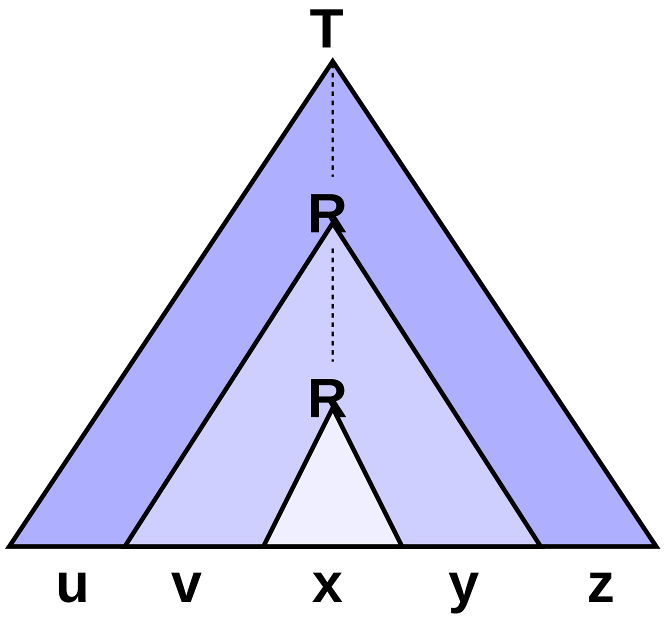
Let A be a CFL and G its CFG.

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

Then s must have a "tall" parse tree.

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

ahmmm,. . . why is that so?



Proof

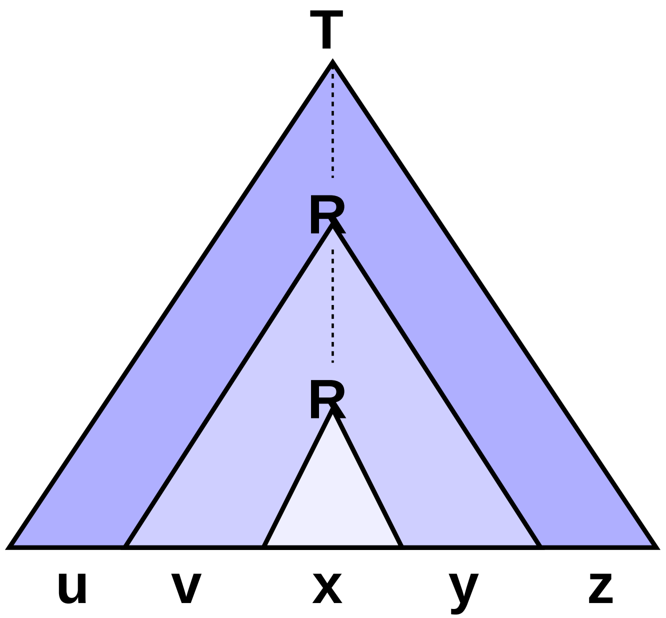
- let G be a CFG for CFL A.
- let b be the max number of symbols in right-hand-side of any rule.
- no node in parse tree has $> b$ children.
- at depth d, can have at most b^d leaves.
- let $|V|$ be the number of variables in G.
- set $\ell = b^{|V|+2}$.

Proof (2)

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

Proof (3)

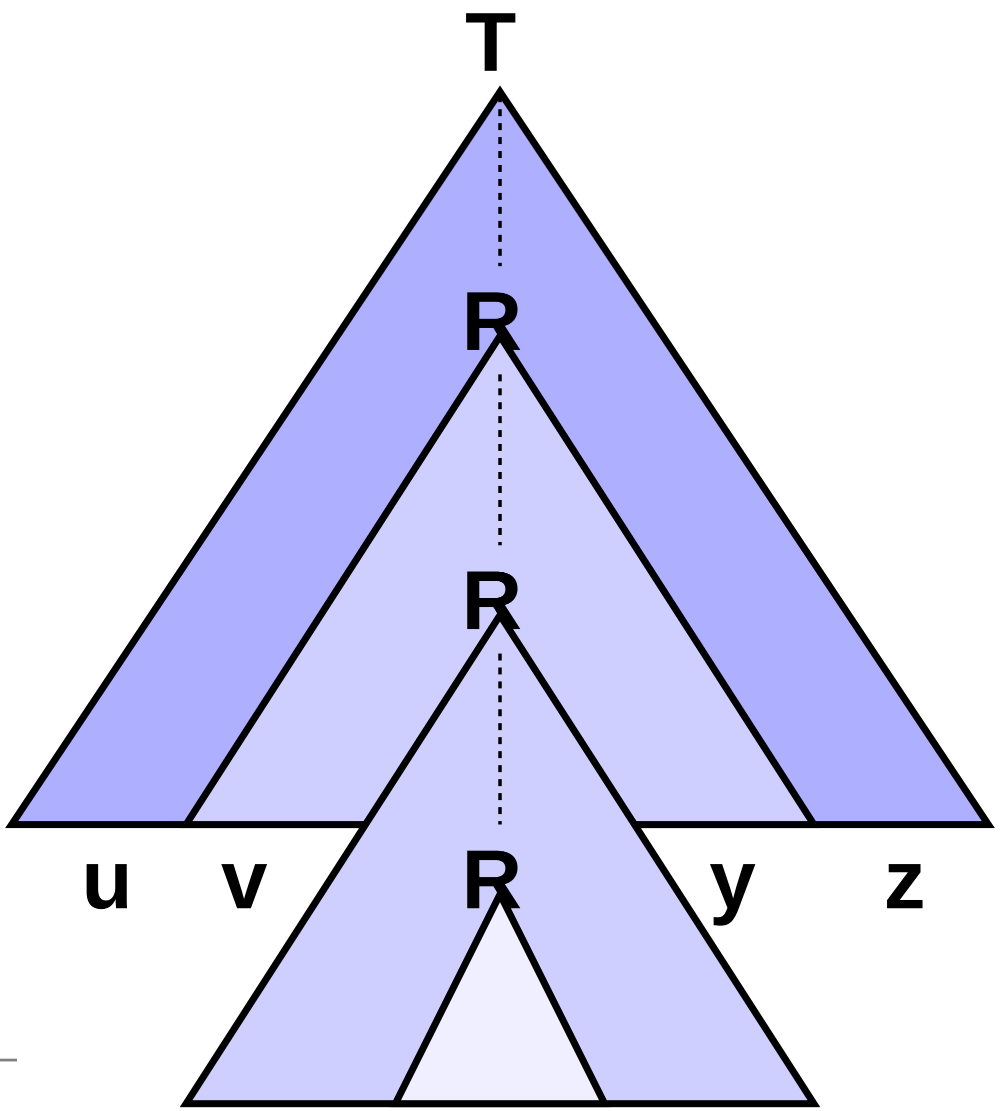
Split $s = uvxyz$



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

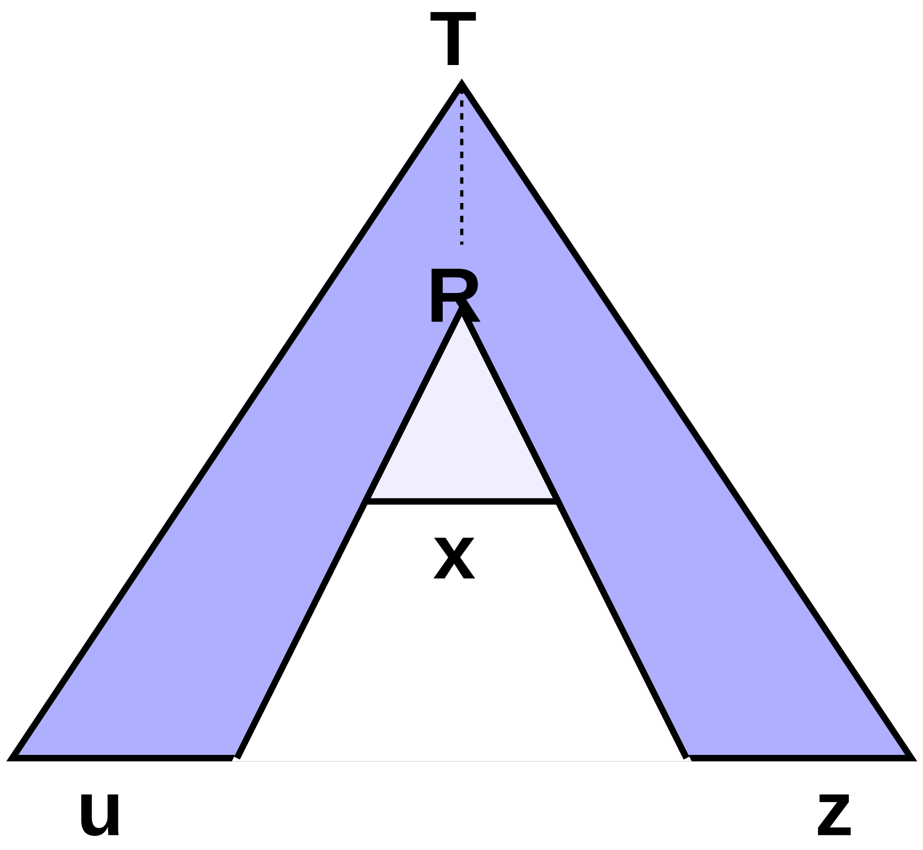
Proof

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



Proof (4)

Replacing larger by smaller yields uxz.



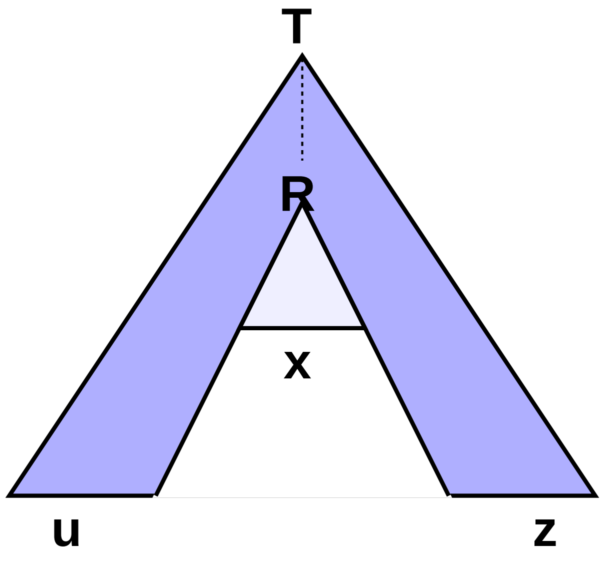
Together, they establish:

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

Proof (5)

Next condition is:

- $|vy| > 0$ (out of $uvxyz$)

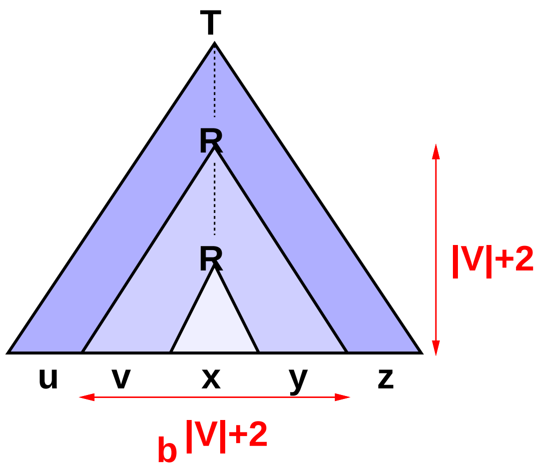


If v and y are both ε, then

is a parse tree for s with fewer nodes than T, contradiction.

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. ♠

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

Non CFL 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 two distinct symbols, because uv^2xy^2z would have out-of-order symbols.
- vxy cannot be 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 (pump down).

♠

Non CFL 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 $|vxy| \leq \ell$.

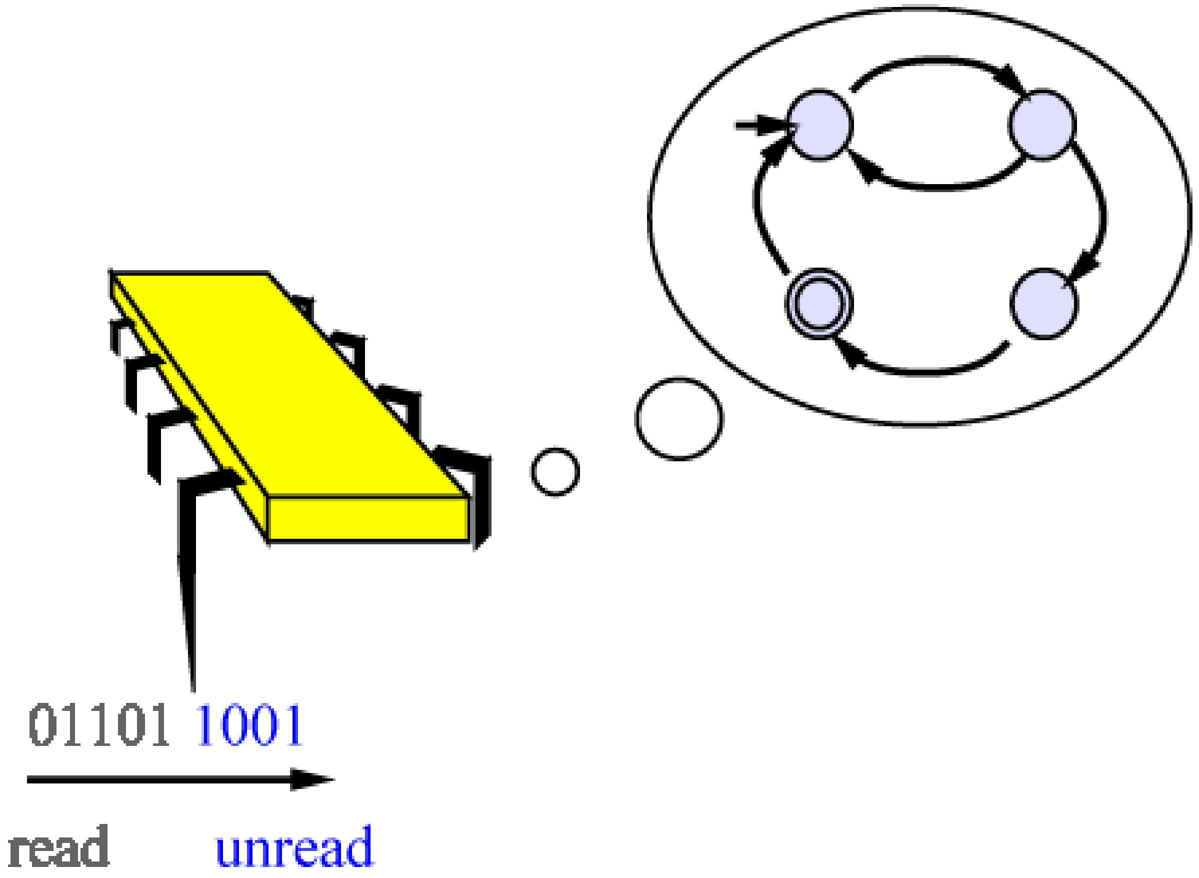
- if vxy is in the first half of s, uv^2xy^2z moves a 1 into the first position in second half.
- if vxy is in the second half, uv^2xy^2z moves a 0 into the last position in first half.
- if vxy straddles the midpoint, then pumping down to uxz yields $0^\ell 1^i 0^j 1^\ell$ where i and j cannot both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

Interestingly, we will shortly see that $D = \{ww^R | w \in \{0,1\}^*\}$ is a CFL.

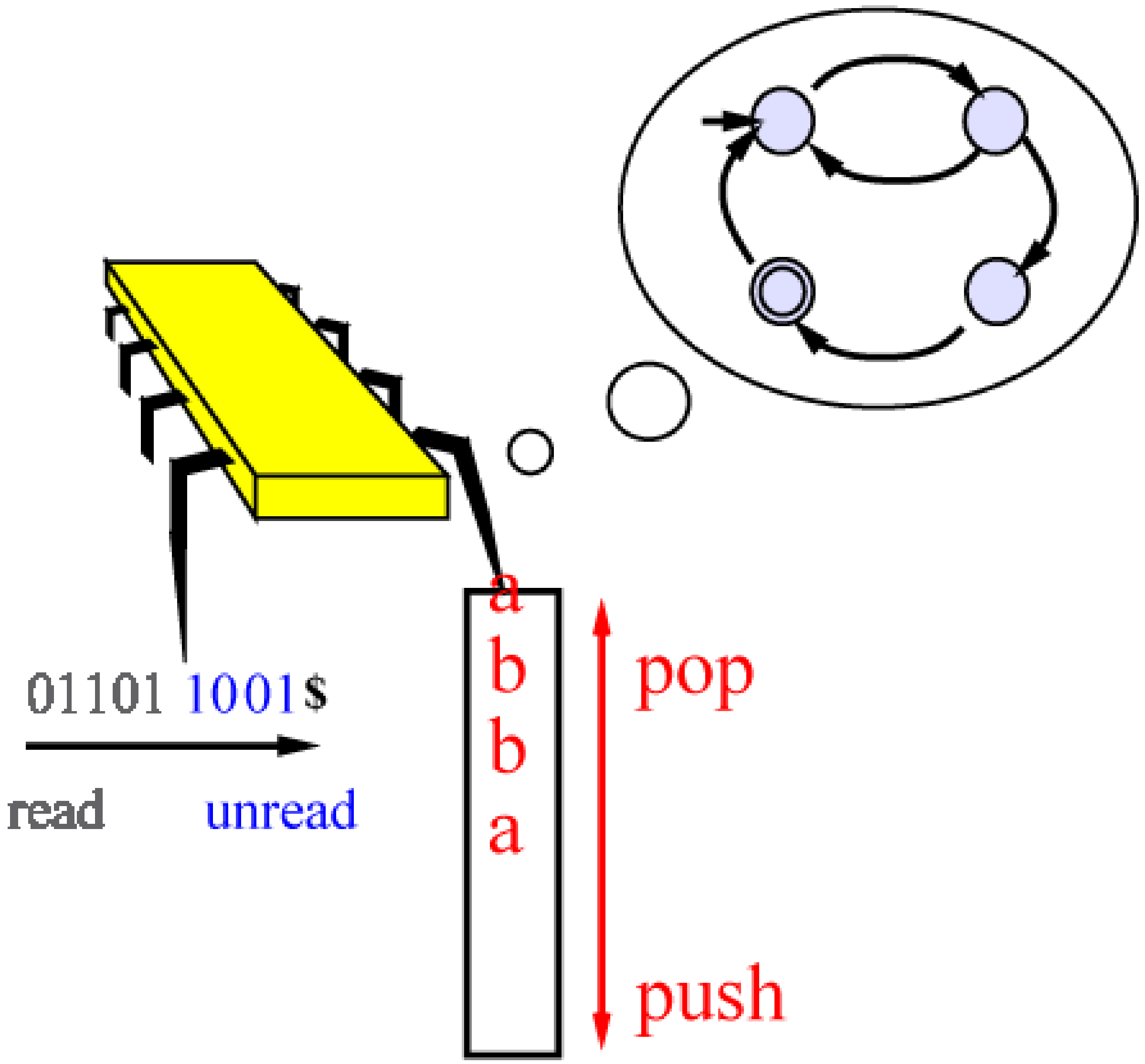
String Generators and String Acceptors

- 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

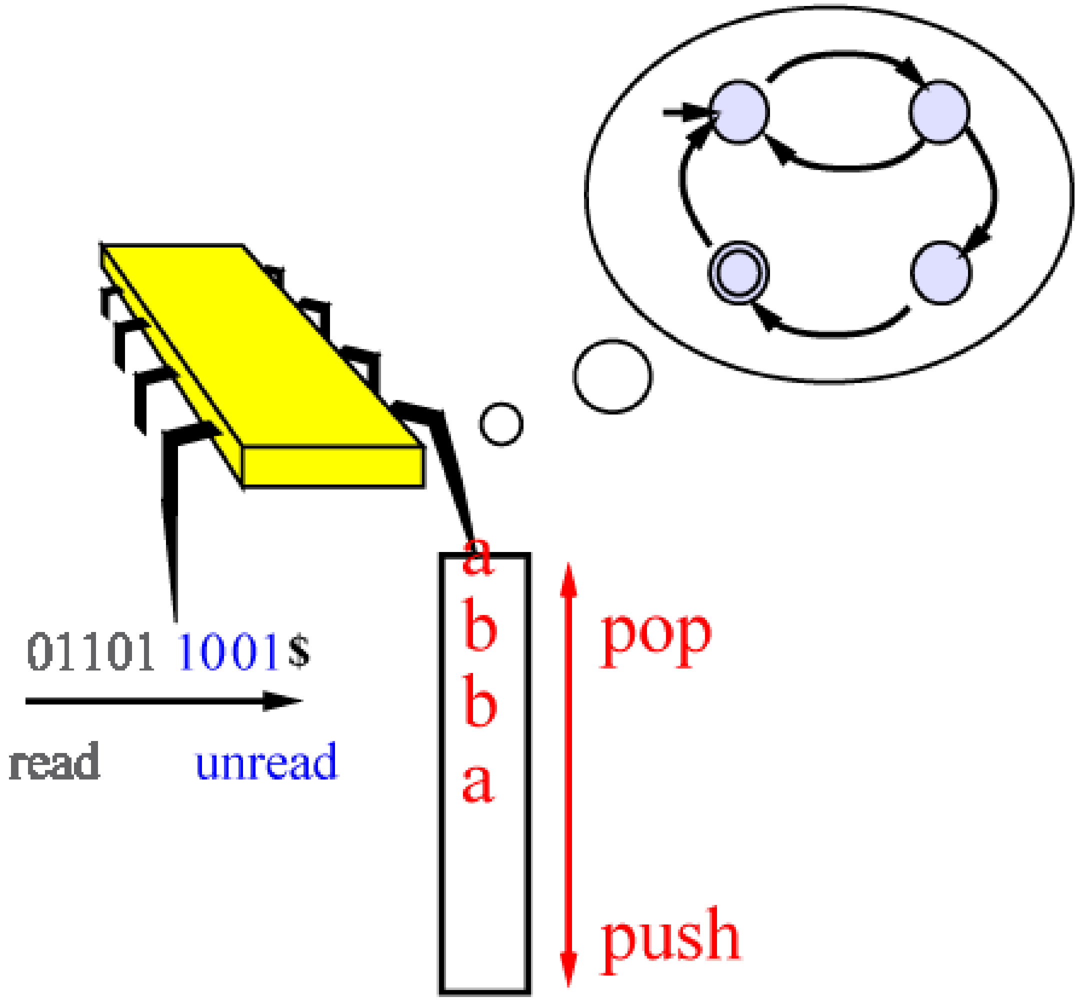


A PushDown Automaton



A PushDown Automaton

We add a memory device with restricted access: A stack (last in, first out; push/pop).



An Example

Recall that the language $\{0^n1^n \mid n \geq 0\}$ is not regular. Consider the following PDA:

- Read input symbols, one by one
- 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 end of input 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
- ($<$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.
- In the string describing the stack, we put the top of the stack on the left (technical item).

Comparing PDA and Finite Automata

- Our PDAs have a special end of input symbol, $\$$. This symbol is not part of input alphabet Σ. (This technical point differs from book. It is for convenience, and can be handled differently too.)
- 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
  - current state: Q
  - next input, if any: $\Sigma_{\varepsilon,\$}$ (=$\Sigma \cup \{\varepsilon\} \cup \{\$\}$)
  - 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: $\mathcal{P}($of 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*,
- 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.
  - Solution:
    - Start by pushing $\$$ onto stack.
    - When you see it again, stack is empty.
- Question: When is input string exhausted?
    - Answer: When $\$$ is read.

Semi Formal Definitions

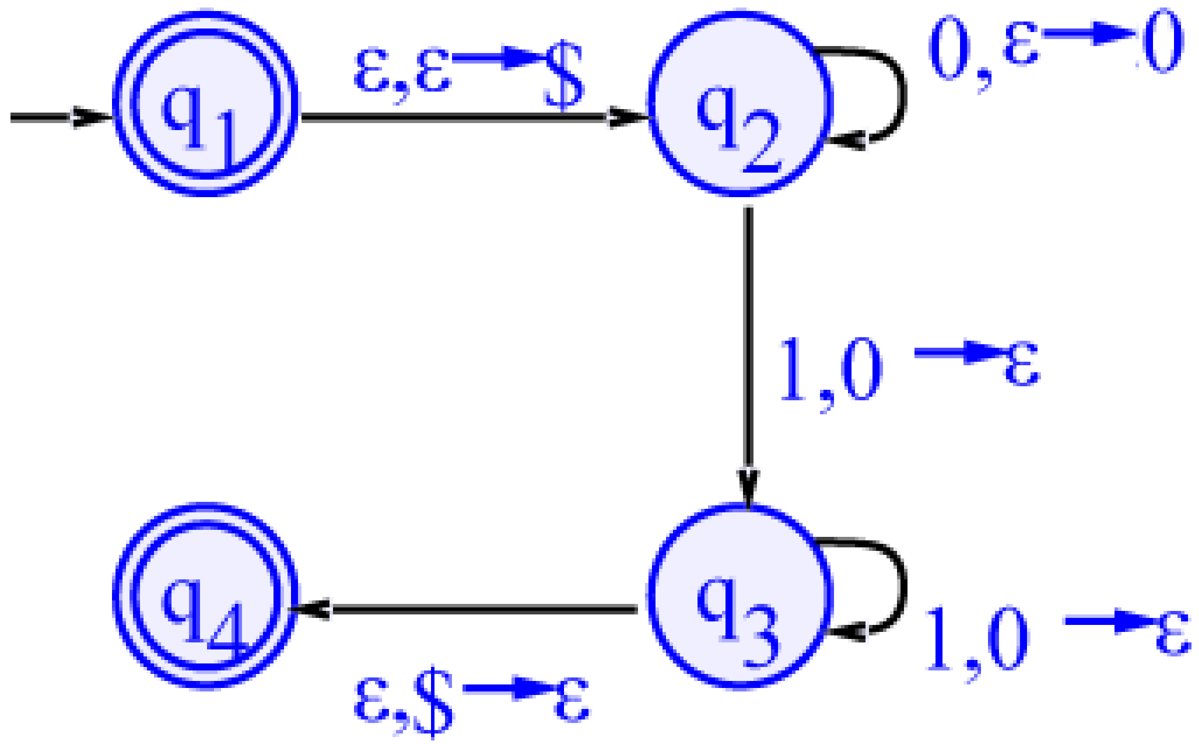
- A pushdown automaton (PDA) M accepts a string x if there is a computation of M on x (a sequence of state and stack transitions according to M's transition function and corresponding to x) that leads to an accepting state and empty stack.
- The language accepted by M is the set of all strings x accepted by M.

Notation

- Transition $a, b \to c$ means
  - if read a from input
  - and pop b from stack
  - then 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 $\{0^n 1^n | n \geq 1\}$.

Does it also accept the empty string?

Another Example

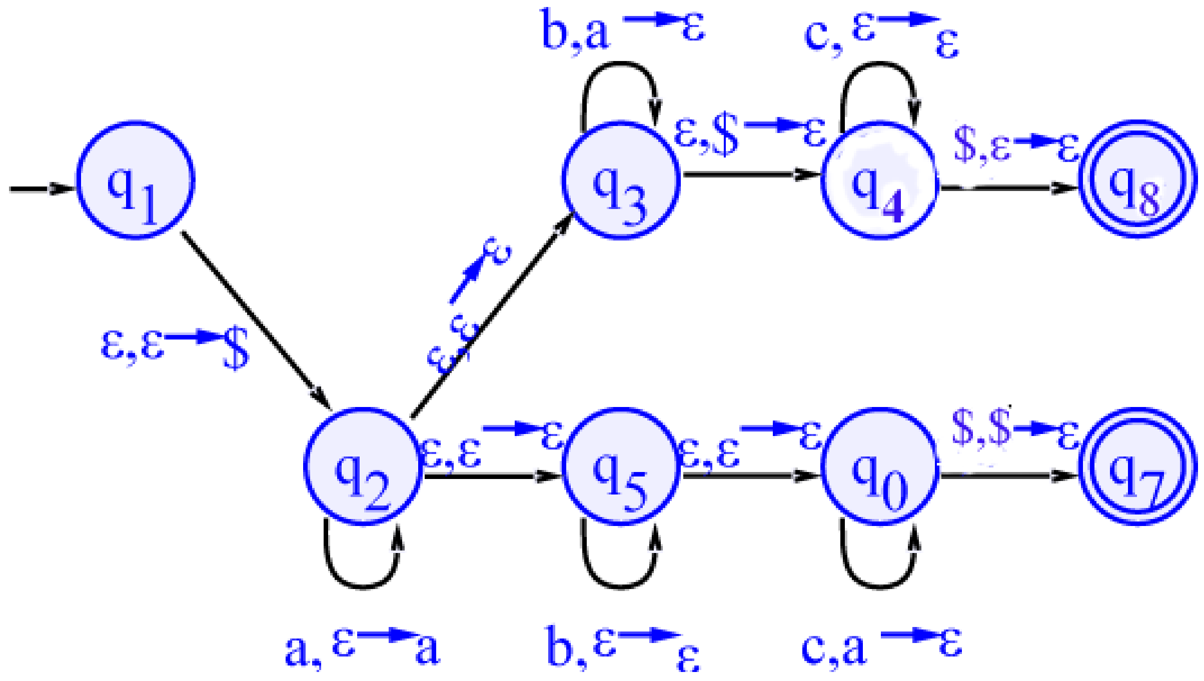
A PDA that accepts

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

Informally:

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

Another Example (cont.)



This PDA accepts

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

Another Example (cont.)

A PDA that accepts $\{a^i b^j c^k | i, j, k > 0$ and $i = j$ 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 that nondeterminism is indeed essential could go,
- $\vdots$
- and realize it does not seem trivial or immediate.
- We will later give a proof that the language $L = \{x^n y^n | n \geq 0\} \cup \left\{x^n y^{2n} \mid n \geq 0\right\}$ is accepted by a non-deterministic PDA but not by a deterministic one.

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.
We will present a high level view of the proof (not all details) later.

Chomsky Normal Form

A simplified, canonical form of context free grammars. Elegant by itself, useful (but not crucial) in proving equivalence theorem. Can also be used to slightly simplify proof of pumping lemma.

Every rule has the form

$$\begin{aligned} A &\rightarrow BC \\ A &\rightarrow a \\ S &\rightarrow \varepsilon \end{aligned}$$

where S is the start symbol, A, B and C are any variable, except B and C not the start symbol, and A can be the start symbol.

Theorem:

Any context-free language is generated by a context-free grammar in Chomsky normal form.
Basic idea:

- Add new start symbol S_0.
- Eliminate all ε rules of the form $A \to \varepsilon$.
- Eliminate all “unit” rules of the form $A \to B$.
- At each step, “patch up” rules so that grammar generates the same language.
- Convert remaining “long rules” to proper form.

Proof

Add new start symbol S_0 and rule $S_0 \rightarrow S$.
Guarantees that new start symbol does not appear on right-hand-side of a rule.

Proof

Eliminating ε rules.

Repeat:

- remove some $A \to \varepsilon$.
- for each $R \to uAv$, add rule $R \to uv$.
- and so on: for $R \to uAvAw$ add $R \to uvAw$, $R \to uAvw$, and $R \to uvw$.
- for $R \to A$ add $R \to \varepsilon$, except if $R \to \varepsilon$ has already been removed.

until all ε-rules not involving the original start variable have been removed.

Proof

Eliminate unit rules.

Repeat:

- remove some $A \rightarrow B$.
- for each $B \rightarrow u$, add rule $A \rightarrow u$, unless this is previously removed unit rule. (u is a string of variables and terminals.)

until all unit rules have been removed.

Proof

Finally, convert long rules.
To replace each $A \rightarrow u_1 u_2 \ldots u_k$ (for $k \geq 3$), introduce new non-terminals

$$N_1, N_2, \ldots, N_{k-1}$$

and rules

$$\begin{array}{rcl} A & \rightarrow & u_1 N_1 \\ N_1 & \rightarrow & u_2 N_2 \\ & \vdots & \\ N_{k-3} & \rightarrow & u_{k-2} N_{k-2} \\ N_{k-2} & \rightarrow & u_{k-1} u_k \end{array} \qquad \spadesuit$$

Conversion Example

Initial Grammar:

$$\begin{aligned} S &\rightarrow ASA \mid aB \\ A &\rightarrow B \mid S \\ B &\rightarrow b \mid \varepsilon \end{aligned}$$

(1) Add new start state:

$$\begin{aligned} S_0 &\rightarrow S \\ S &\rightarrow ASA \mid aB \\ A &\rightarrow B \mid S \\ B &\rightarrow b \mid \varepsilon \end{aligned}$$

Conversion Example (2)

$$\begin{array}{rcl} S_0 & \rightarrow & S \\ S & \rightarrow & ASA \mid aB \\ A & \rightarrow & B \mid S \\ B & \rightarrow & b \mid \varepsilon \end{array}$$

(2) Remove ε-rule $B \rightarrow \varepsilon$:

$$\begin{array}{rcl} S_0 & \rightarrow & S \\ S & \rightarrow & ASA \mid aB \mid a \\ A & \rightarrow & B \mid S \mid \varepsilon \\ B & \rightarrow & b \mid \varepsilon \end{array}$$

Conversion Example (3)

$$\begin{aligned} S_0 &\rightarrow S \\ S &\rightarrow ASA \mid aB \mid a \\ A &\rightarrow B \mid S \mid \varepsilon \\ B &\rightarrow b \end{aligned}$$

(3) Remove ε-rule $A \rightarrow \varepsilon$:

$$\begin{aligned} S_0 &\rightarrow S \\ S &\rightarrow ASA \mid aB \mid a \mid AS \mid SA \mid S \\ A &\rightarrow B \mid S \mid \varepsilon \\ B &\rightarrow b \end{aligned}$$

Conversion Example (4)

$$\begin{array}{rcl} S_0 & \rightarrow & S \\ S & \rightarrow & ASA \mid aB \mid a \mid AS \mid SA \mid S \\ A & \rightarrow & B \mid S \\ B & \rightarrow & b \end{array}$$

(4) Remove unit rule $S \rightarrow S$

$$\begin{array}{rcl} S_0 & \rightarrow & S \\ S & \rightarrow & ASA \mid aB \mid a \mid AS \mid SA \mid S \\ A & \rightarrow & B \mid S \\ B & \rightarrow & b \end{array}$$

Conversion Example (5)

$$\begin{array}{rcl} S_0 & \rightarrow & S \\ S & \rightarrow & ASA \mid aB \mid a \mid AS \mid SA \\ A & \rightarrow & B \mid S \\ B & \rightarrow & b \end{array}$$

(5) Remove unit rule $S_0 \rightarrow S$:

$$\begin{array}{rcl} S_0 & \rightarrow & S \mid ASA \mid aB \mid a \mid AS \mid SA \\ S & \rightarrow & ASA \mid aB \mid a \mid AS \mid SA \\ A & \rightarrow & B \mid S \\ B & \rightarrow & b \end{array}$$

Conversion Example (6)

$$\begin{aligned} S_0 &\rightarrow ASA \mid aB \mid a \mid AS \mid SA \\ S &\rightarrow ASA \mid aB \mid a \mid AS \mid SA \\ A &\rightarrow B \mid S \\ B &\rightarrow b \end{aligned}$$

(6) Remove unit rule $A \rightarrow B$:

$$\begin{aligned} S_0 &\rightarrow ASA \mid aB \mid a \mid AS \mid SA \\ S &\rightarrow ASA \mid aB \mid a \mid AS \mid SA \\ A &\rightarrow B \mid S \mid b \\ B &\rightarrow b \end{aligned}$$

Conversion Example (7)

$$\begin{array}{rcl} S_0 & \rightarrow & ASA \mid aB \mid a \mid AS \mid SA \\ S & \rightarrow & ASA \mid aB \mid a \mid AS \mid SA \\ A & \rightarrow & S \mid b \\ B & \rightarrow & b \end{array}$$

Remove unit rule $A \rightarrow S$:

$$\begin{array}{rcl} S_0 & \rightarrow & ASA \mid aB \mid a \mid AS \mid SA \\ S & \rightarrow & ASA \mid aB \mid a \mid AS \mid SA \\ A & \rightarrow & S \mid b \mid ASA \mid aB \mid a \mid AS \mid SA \\ B & \rightarrow & b \end{array}$$

Conversion Example (8)

$$\begin{array}{lcl} S_0 & \rightarrow & ASA \mid aB \mid a \mid AS \mid SA \\ S & \rightarrow & ASA \mid aB \mid a \mid AS \mid SA \\ A & \rightarrow & b \mid ASA \mid aB \mid a \mid AS \mid SA \\ B & \rightarrow & b \end{array}$$

(8) Final simplification – treat long rules:

$$\begin{array}{rcl} S_0 & \rightarrow & AA_1 \mid UB \mid a \mid SA \mid AS \\ S & \rightarrow & AA_1 \mid UB \mid a \mid SA \mid AS \\ A & \rightarrow & b \mid AA_1 \mid UB \mid a \mid SA \mid AS \\ A_1 & \rightarrow & SA \\ U & \rightarrow & a \\ B & \rightarrow & b \end{array}$$

$\surd$

Another PDA Example

A palindrome is a string w satisfying $w = w^{\mathcal{R}}$.

- “Madam I’m Adam”
- “Dennis and Edna sinned”
- “Red rum, sir, is murder”
- “Able was I ere I saw Elba”
- “In girum imus nocte et consumimur igni” (Latin: "we go into the circle by night, we are consumed by fire”.)
- “$\nu\iota\psi o\nu\ \alpha\nu o\mu\eta\mu\alpha\tau\alpha\ \mu\eta\ \mu o\nu\alpha\nu\ o\psi\iota\nu$”
- Palindromes also appear in nature. For example as DNA restriction sites – short genomic strings over $\{A, C, T, G\}$, being cut by (naturally occurring) restriction enzymes.

A PDA that Recognizes Palindromes

- On input x, the PDA starts pushing x into stack.
- At some point, PDA guesses that the mid point of x was reached.
- Pops and compares to input, letter by letter.
- If end of input occurs together with emptying of stack, accept.
- This PDA accepts palindromes of *even length* over the alphabet (all lengths is easy modification).
- Again, non-determinism (at which point to make the switch) seems necessary.