Context Free Languages and Pushdown Automata

Last Updated: August 25th, 2026

1 Introduction

Context free languages are foundational for defining several types of computing languages that occur in practice; including programming languages, markup languages, and languages for communication protocols. CFL's were first studied in relation to natural-language processing in the 1950's.

Their importance stems from the following.

1. CFL's are significantly more expressive than regular languages in that they are capable of defining recursive languages that may have unlimited recursive depth.
2. A CFL can be recognized by a pushdown automaton (PDA). Unlike DFA's a PDA has unlimited memory, albeit in the form of a stack whose access is limited to i) reading the top of the stack, ii) pushing on to the stack, and iii) popping the top of the stack. PDA's are also interesting because their nondeterministic counterparts (NPDA's) are more powerful than PDA's, and the set of languages accepted by an NPDA is equal to the set of CFL languages.
3. Letting `CFL` denote the class of all context-free languages, it can be shown that $\texttt{CFL} \subseteq \texttt{P}$. This is due to the fact that there exists a dynamic-programming algorithm that can decide any `CFL` language in $O(n^3)$ steps, where n is the length of the input word. See Sipser, page 290.

Although every regular language is also a CFL (see the exercises), the converse is not true. For example, it can be proved that the language

$$L = \{a^n b^n | n \geq 0\}$$

is a CFL but is *not* regular.

Proposition 1. The language

$$L = \{a^n b^n | n \geq 0\}$$

is not regular.

Proof.

1. Suppose that L is regular and let M be a DFA that accepts L and has initial state q_0.
2. Since M has a finite number of states and there are infinitely many n values, there must be at least one state q for which there is an $n_1 > 0$ and $n_2 > 0$ for which
   (a) $n_1 \neq n_2$
   (b) When either a^{n_1} or a^{n_2} is read starting in state q_0, the computation ends at q.
   (c) When reading b^{n_1} starting in state q, the computation ends in an accepting state q_a.
3. But then the facts stated in 2) imply that, when starting in state q_0 and reading $a^{n_2}b^{n_1}$, the computation moves to state q after reading a^{n_2}, followed by moving to state q_a after reading b^{n_1}.
4. Then by 3), it follows that M accepts $a^{n_2}b^{n_1}$ which is a contradiction, since $n_1 \neq n_2$. □

$a^{n_1} b^{n_1}$

$a^{n_2} b^{n_1} \notin L$ but is accepted

2 Context-Free Grammars

A **Context-Free Grammar (CFG)** is a 4-tuple (V, Σ, R, S), where

1. V is a finite set of variables
2. Σ is a finite set that is disjoint from V, called the **terminal set**
3. R is a finite set of rules where each **rule** has the form

   $$A \to s,$$

   where $A \in V$ and $s \in (V \cup \Sigma)^*$. Variable A is referred to as the **head** of the rule, while s is referred to its **body**.
4. $S \in V$ is the **start variable**

$A \to BabAC$

$A, B, C \in V$

s

$a, b \in \Sigma$

Example 1. Consider the set of rules

$$R = \{S \to SS, S \to aSb, S \to \varepsilon\}.$$

Then we may use this set of rules to define a CFG $G = (V, \Sigma, R, S)$, where

$$V = \{S\},$$

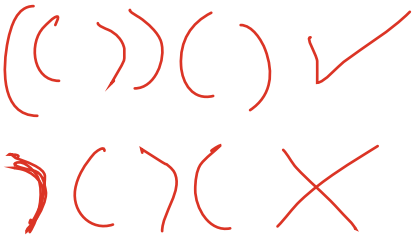
$$\Sigma = \{a, b\},$$

and variable S is the start variable.

For brevity we may list together rules having the same head as follows.

$$S \to SS \mid aSb \mid \varepsilon.$$

Here, each of the rule bodies is separated by a vertical bar. □



Example 2. One common use of CFG's is to provide grammatical formalism for natural languages. For example, consider the set of rules R:

$$\langle \text{SENTENCE} \rangle \to \langle \text{NOUN-PHRASE} \rangle \langle \text{VERB-PHRASE} \rangle$$

$$\langle \text{NOUN-PHRASE} \rangle \to \langle \text{COMPLEX-NOUN} \rangle \mid \langle \text{COMPLEX-NOUN} \rangle \langle \text{PREP-PHRASE} \rangle$$

$$\langle \text{VERB-PHRASE} \rangle \to \langle \text{COMPLEX-VERB} \rangle \mid \langle \text{COMPLEX-VERB} \rangle \langle \text{PREP-PHRASE} \rangle$$

$$\langle \text{PREP-PHRASE} \rangle \to \langle \text{PREP} \rangle \langle \text{COMPLEX-NOUN} \rangle$$

$$\langle \text{COMPLEX-NOUN} \rangle \to \langle \text{ARTICLE} \rangle \langle \text{NOUN} \rangle$$

$$\langle \text{COMPLEX-VERB} \rangle \to \langle \text{VERB} \rangle \mid \langle \text{VERB} \rangle \langle \text{NOUN-PHRASE} \rangle$$

$$\langle \text{ARTICLE} \rangle \to \text{a} \mid \text{the}$$

$$\langle \text{NOUN} \rangle \to \text{trainer} \mid \text{dog} \mid \text{whistle}$$

$$\langle \text{VERB} \rangle \to \text{calls} \mid \text{pets} \mid \text{sees}$$

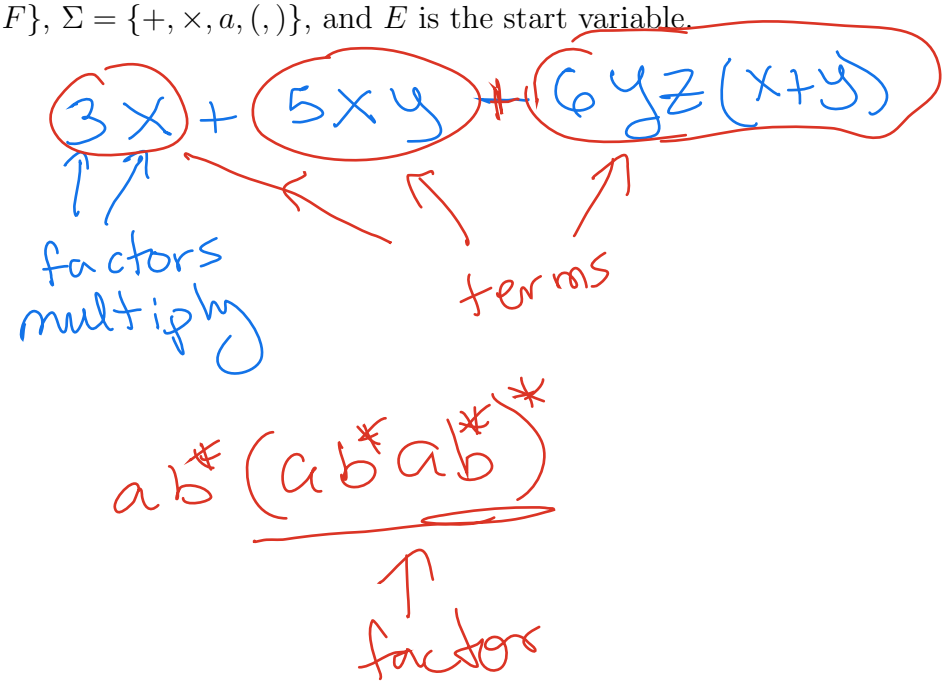
$$\langle \text{PREP} \rangle \to \text{with} \mid \text{in}$$

Here, the variables are the ten parts of speech delimited by $\langle\ \rangle$, Σ is the lowercase English alphabet, including the space character, and $\langle \text{SENTENCE} \rangle$ is the start variable.

Example 3. A CFG may also be used to define the syntax of a programming language. One fundamental language component to any programming language is that of an *expression*. The following rules imply a CFG for defining expressions formed by a single terminal `a`, parentheses, and the two arithmetic operations $+$ and $\times$. Here E stands for `expression`, T for `term`, and F for `factor`.

$$E \to E + T \mid T$$
$$T \to T \times F \mid F$$
$$F \to (E) \mid a$$

We have $V = \{E, T, F\}$, $\Sigma = \{+, \times, a, (,)\}$, and E is the start variable.



2.1 Grammar derivations

Let $G = (V, \Sigma, R, S)$ be a CFG, then the language $D(G) \subseteq (V \cup \Sigma)^*$ of **derived words** is structurally defined as follows.

Atom $S \in D(G)$.

Compound Rule Suppose $s \in D(G)$, s is of the form uAv for some $u, v \in (V \cup \Sigma)^*$, $A \in V$, and $A \to \gamma$ is a rule of G, then

$$u\gamma v \in D(G).$$

In this case we write

$$s \Rightarrow u\gamma v,$$

and say that s **yields** $u\gamma v$. In words, to get a new derived word, take an existing derived word and replace one of its variables A with the body of a rule whose head is A.

The subset $L(G)$ of derived words $w \in D(G)$ for which $w \in \Sigma^*$ is called the **context-free language (CFL)** associated with G. Thus, the words of $L(G)$ consist only of terminal symbols.

2.2 The Derivation relation

Let u and v be words in $(V \cup \Sigma)^*$. We say that u **derives** v, written

$$u \stackrel{*}{\Rightarrow} v$$

if and only if either $u = v$ or there is a sequence of words $w_1, w_2, \ldots, w_n$ such that

$$u = w_1 \Rightarrow w_2 \Rightarrow w_3 \Rightarrow \cdots \Rightarrow w_n = v.$$

Such a sequence is called a **derivation sequence** from u to v.

$$L(G) = \{w \in \Sigma^* | S \stackrel{*}{\Rightarrow} w\}.$$

Example 4. For the arithmetic-expression CFG from Example 3 and having rule set

$$E \to E + T \mid T$$
$$T \to T \times F \mid F$$
$$F \to (E) \mid a$$

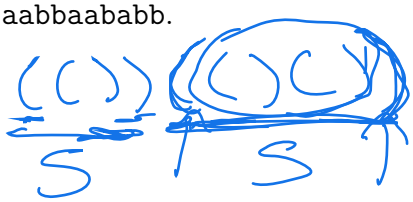
which of the following statements are true?

1. $E \Rightarrow a \times a$
2. $a + F \Rightarrow a + (E)$
3. $E \overset{*}{\Rightarrow} a + T$
4. $(F) \overset{*}{\Rightarrow} (a \times a)(a + a)$

a = (b =)

Example 5. Use the CFG from Example 1 to derives the word aabbaababb.

$S \to SS \mid aSb \mid \varepsilon$



Solution.

$S \Rightarrow SS \Rightarrow aSbS \Rightarrow aaSbbS$

$\Rightarrow aabbS \Rightarrow$

$aabbaSb \Rightarrow aabbaSSb$

$\Rightarrow aabbaaSbSb \Rightarrow$

$aabbaaSbb \Rightarrow$

$aabbaababb \Rightarrow$

aabb | aababb

(( )) (( ) ( ))

Leftmost Derivation

2.3 Derivation parse trees

Determining if an arbitrary word belongs to $L(G)$ is of fundamental importance. But in addition, it is sometimes important to know the structure of the grammar's derivation of the word. For example, if a CFG generates arithmetic expressions, then knowing the structure of the derivation allows one to readily evaluate the expression (assuming the expression terminals have assigned values and the expression operations are properly defined). A **parse tree** for a word $w \in L(G)$ is a tree whose structure and node labels reflect the derivation w, where, from left to right, the leaves of the tree are labeled with the letters of w. Indeed, consider the derivation sequence

$$S = w_1 \Rightarrow w_2 \Rightarrow \cdots \Rightarrow w_n = w.$$

Then the parse tree for w can be defined in a step-by-step manner. To begin the parse tree T_1 for $S = w_1$ consists of a single node labeled with S.

Now suppose a parse tree T_k has been associated with w_k, the k th word of the derivation. Moreover, assume that, from left to right, the leaves of T_k are labeled in one-to-one correspondence with the symbols of w_k. Moreover, assume that w_k has the form $w_k = uAv$, where A is substituted for a word γ, so that $w_{k+1} = u\gamma v$. Then T_{k+1} is obtained from T_k by assigning the leaf node labeled with A a number of children equal to the length of γ and for which, from left to right, the i th child is labeled with the ith symbol of γ.

Example 6. Use the CFG from Example 3 to derive the expression $a \times (a + a)$, and provide the parse tree associated with the derivation.

$$E \to E + T \mid T$$

$$T \to T \times F \mid F$$

$$F \to (E) \mid a$$

Solution.

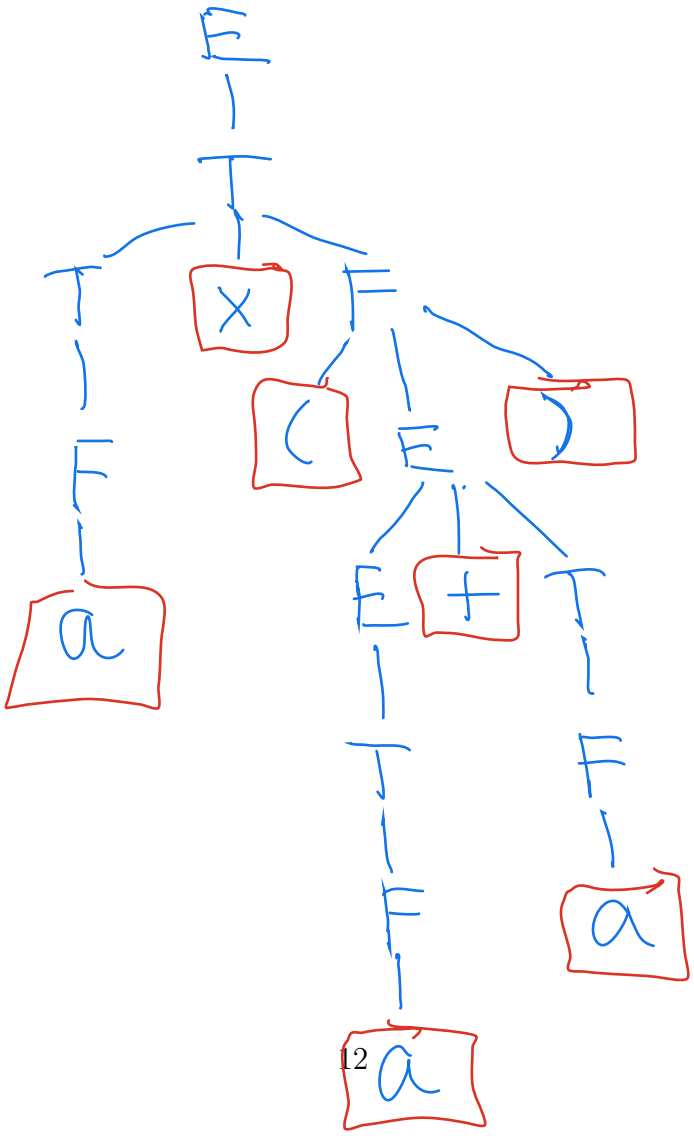
$E \Rightarrow T \Rightarrow T \times F \Rightarrow F \times F \Rightarrow$

$a \times F \Rightarrow a \times (E) \Rightarrow a \times (E + T)$

$\Rightarrow a \times (T + T) \Rightarrow a \times (F + T) \Rightarrow$

$a \times (a + T) \Rightarrow a \times (a + F) \Rightarrow$

$a \times (a + a)$



2.4 Ambiguity

Given a CFG G, and a word $w \in L(G)$, there may be several different derivations of w from start symbol S. Many of these derivations however will yield identical parse trees. But in the event that two different derivation sequences of w from S yield two different parse trees, then we call G **ambiguous**. It turns out that an easy way to check for ambiguity is to check that no word w has more than one *leftmost derivation*.

Given grammar $G = (V, \Sigma, R, S)$ and word $w \in L(G)$, a derivation sequence $S = w_0 \Rightarrow w_1 \Rightarrow \cdots \Rightarrow w_{n-1} \Rightarrow w_n = w$ is called a **leftmost derivation** of w provided that, for every $0 \leq i \leq n-1$, the yielding of w_i from w_{i-1} was obtained by replacing the leftmost variable A of w_{i-1} with the body of one of a rule whose head is A. Therefore, if w has more than one leftmost derivation, it must be the case that a different sequence of rules were used to derive w. When this happens we call G ambiguous, since some words in the grammar have more than one parsing structure.

Example 7. Show that the grammar defined by the following rules is ambiguous.

the trainer calls the dog with the whistle

$$\langle \text{SENTENCE} \rangle \to \langle \text{NOUN-PHRASE} \rangle \langle \text{VERB-PHRASE} \rangle$$

$$\langle \text{NOUN-PHRASE} \rangle \to \langle \text{COMPLEX-NOUN} \rangle \mid \langle \text{COMPLEX-NOUN} \rangle \langle \text{PREP-PHRASE} \rangle$$

$$\langle \text{VERB-PHRASE} \rangle \to \langle \text{COMPLEX-VERB} \rangle \mid \langle \text{COMPLEX-VERB} \rangle \langle \text{PREP-PHRASE} \rangle$$

$$\langle \text{PREP-PHRASE} \rangle \to \langle \text{PREP} \rangle \langle \text{COMPLEX-NOUN} \rangle$$

$$\langle \text{COMPLEX-NOUN} \rangle \to \langle \text{ARTICLE} \rangle \langle \text{NOUN} \rangle$$

$$\langle \text{COMPLEX-VERB} \rangle \to \langle \text{VERB} \rangle \mid \langle \text{VERB} \rangle \langle \text{NOUN-PHRASE} \rangle$$

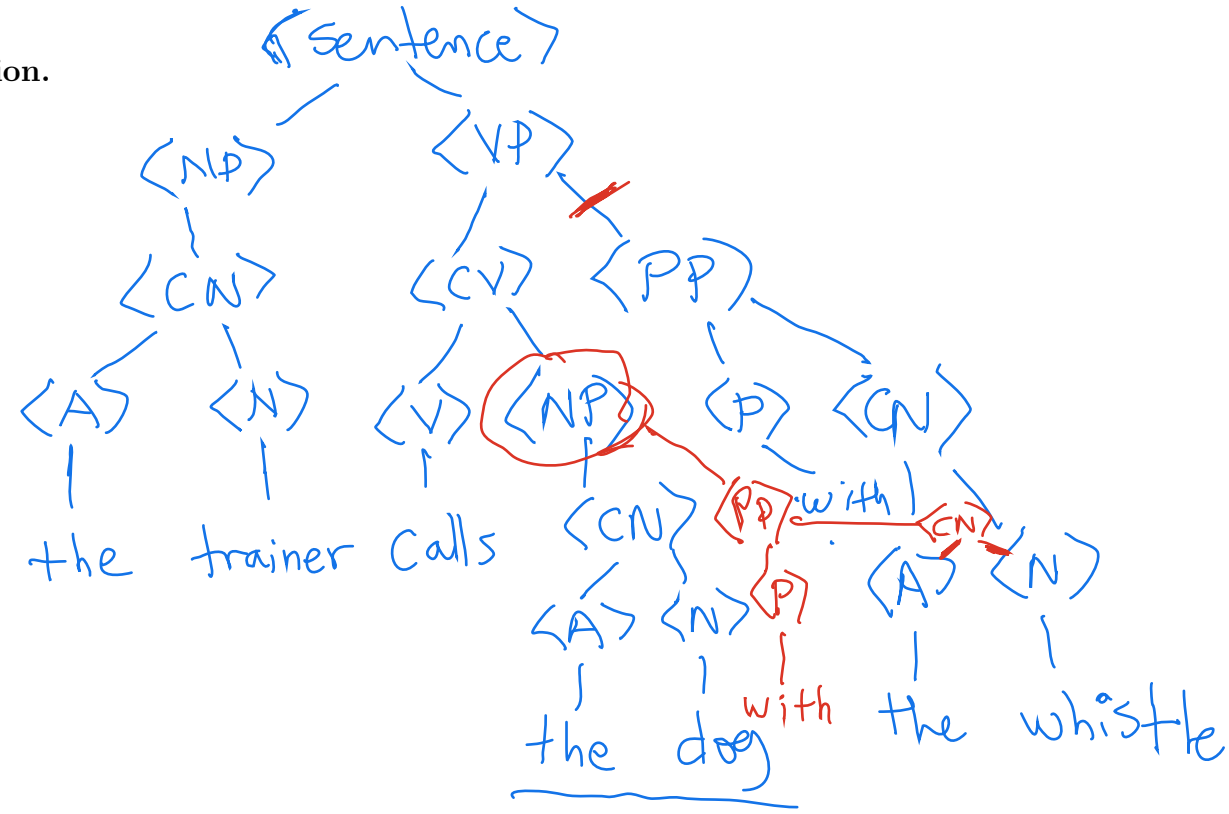
$$\langle \text{ARTICLE} \rangle \to \text{a} \mid \text{the}$$

$$\langle \text{NOUN} \rangle \to \text{trainer} \mid \text{dog} \mid \text{whistle}$$

$$\langle \text{VERB} \rangle \to \text{calls} \mid \text{pets} \mid \text{sees}$$

$$\langle \text{PREP} \rangle \to \text{with} \mid \text{in}$$

Solution.



2.5 Designing CFG's

The following are some tips for designing context free grammars.

CFG's for Regular Languages. The first step is to design a DFA (or NFA)

$$M = (Q, \Sigma, \delta, Q_0, F)$$

for the language, where $Q = \{Q_0, Q_1, \ldots, Q_m\}$ is the set of states. Then the desired CFG is (Q, Σ, R, Q_0), where R consists of rules of the following form. Suppose $\delta(Q_i, s) = Q_j$. Then we add the rule

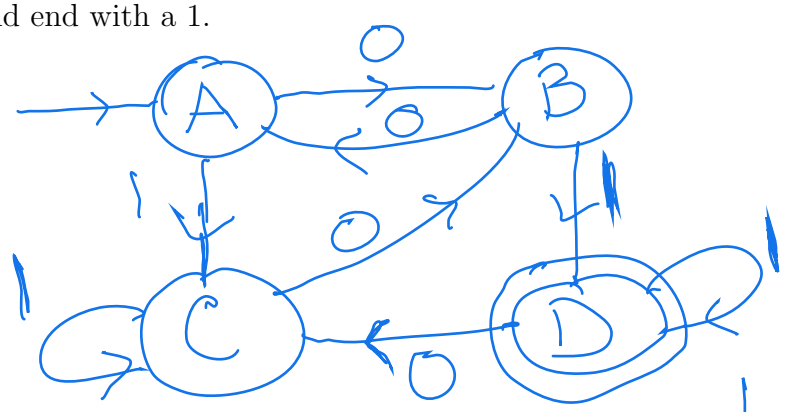
$$Q_i \to sQ_j.$$

We also add the rule

$$Q' \to \varepsilon$$

for every accepting state $Q' \in F$.

Example 8. Provide a CFG for the language consisting of all binary words that have an odd number of 0's and end with a 1.



$V = \{A, B, C, D\}$

Start $= A$

$\Sigma = \{0, 1\}$

Rules:

$A \to 0B \mid 1C$

$B \to 1D \mid 0A$

$C \to 1C \mid 0B$

$D \to 0C \mid 1D \mid \varepsilon$

$A \Rightarrow 1C \Rightarrow 10B \Rightarrow 101D$

$\Rightarrow 101$

Starting from the outside and working inward. In many applications the left side of a word must depend on its right side. This can be done by creating one or more rules that establish a left-right relationship starting from the outside and working inward. The next two examples illustrate this idea.

Example 9. Provide a CFG G for which

$$L(G) = \{a^n b^n | n \geq 0\}.$$

Provide a derivation of a^2b^2 and draw its parse tree.

$S \to aSb \mid \varepsilon$

$S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow aabb = a^2b^2$

CFG's that are the union of other CFG's. As is the case with regular languages, quite often is the case that a CFL can be expressed as the union of two or more CFL's. Hence, one can construct a CFG for the language by "unioning" the CFG's of the unioned languages. For example, suppose CFL L is the union of three CFL's L_1, L_2, L_3. Moreover, suppose G_i is a CFG for L_i, $i = 1, 2, 3$ and that these CFG's do not have any common variables and that their respective start variables are S_1, S_2, S_3. Then a valid CFG for L would be $G = G_1 \cup G_2 \cup G_3$ together with the added start variable S and rules

$$S \to S_1 \mid S_2 \mid S_3.$$

This strategy yields the following result.

Proposition 2. The union of two or more CFL's is also a CFL.

Example 10. Provide a CFG G for which

$$L(G) = \{x^i y^j z^k | i, j, k \geq 0 \text{ and } j = 2i \text{ or } k = 2j\}.$$

xxyzz

$S_{xy} \to W_{xy} Z$

$W_{xy} \to x W_{xy} yy \mid \varepsilon$

$Z \to Zz \mid \varepsilon$

$S_{yz} \to X \mid W_{yz}$

$X \to Xx \mid \varepsilon$

$W_{yz} \to y W_{yz} zz \mid \varepsilon$

$S \to S_{xy} \mid S_{yz}$

$S \Rightarrow S_{yz} \Rightarrow X \mid W_{yz} \Rightarrow Xx \mid W_{yz} \Rightarrow Xxx W_{yz}$

$\Rightarrow xx \mid W_{yz} \Rightarrow xxy W_{yz} zz \Rightarrow xxyzz$

Example 11. Use the CFG from the previous example to provide a derivation of xy^2z and draw its parse tree.

3 Chomsky Normal Form

Definition 1. Two CFG's G_1 and G_2 are said to be equivalent iff $L(G_1) = L(G_2)$.

Sometimes when considering a CFG, it is helpful to assume that its derivations yield binary parse trees. It turns out that any CFG can be converted to an equivalent one that has this property.

Definition 2. A CFG $G = (V, \Sigma, R, S)$ is in **Chomsky Normal Form** iff the following conditions hold.

1. The start variable S may not appear on the right-hand side of any rule in R.
2. Rule $A \to \varepsilon$ is only allowed when $A = S$.
3. All other rules must be of the form $A \to BC$, where $B, C \in V - \{S\}$, or $A \to a$, where $a \in \Sigma$.

Algorithm for converting a CFG to one in CNF.

- **Input:** CFG $G = (V, \Sigma, R, S)$
- **Output:** A new CFG $\hat{G}$ in Chomsky Normal Form that is equivalent to G.
- The terminal alphabet of $\hat{G}$ is equal to Σ
- Initialize the variables and rules of $\hat{G}$ as being equal to those of G
- Add a new variable S_0 to $\hat{G}$ and designate it as the start variable for $\hat{G}$
- Add the rule $S_0 \to S$ to $\hat{G}$
- **Comment: remove the ε production rules**
- While there is rule of the form $A \to \varepsilon$ in $\hat{G}$ and $A \neq S_0$
  - For each rule of the form $B \to u_1 A u_2 A \cdots u_n A u_{n+1}$ (where A is not a symbol of any u_i word)
    - Create 2^n new rules which represent the different possible ways to make a new rule from $B \to u_1 A u_2 A \cdots u_n A u_{n+1}$ by either keeping or removing each of the A variables.
    - Remove any of these new 2^n rules that have the form $D \to \varepsilon$, and have already been processed in the outer while-loop (otherwise the algorithm will loop forever)
  - Add all the newly created rules from the previous loop to $\hat{G}$
  - Remove $A \to \varepsilon$ from $\hat{G}$
- **Comment: eliminate unit rules of the form $A \to B$**
- While there is rule of the form $A \to B$ in $\hat{G}$ and $B \in V$
  - For each rule of the form $B \to u$
    - Create a new rule $A \to u$ and add it to $\hat{G}$ unless it is a unit rule already processed in the outer while loop
  - Remove $A \to B$ from $\hat{G}$
- **Comment: eliminate all remaining rules not in proper form**
- For each rule of the form $A \to u_1 u_2 \dots u_k$, where $k \geq 3$
  - Replace this rule with the set of rules $A \to u_1 A_1$, $A_1 \to u_2 A_2$, …, $A_{k-2} \to u_{k-1} u_k$, where the A_i variables are all new
  - For each i, if u_i is a terminal symbol, then replace it with unused variable U_i, and add the rule $U_i \to u_i$

Example 12. Apply the CNF conversion algorithm to the following CFG.

$S \to ASC|Bb$

$A \to C|S$

$B \to a|\varepsilon$

$C \to b|B$

Solution.

1. Add rule $S_0 \to S$

   $\mathbf{S_0 \to S}$
   $S \to ASC|Bb$
   $A \to C|S$
   $B \to a|\varepsilon$
   $C \to b|B$

2. Remove ε-rule $B \to \varepsilon$

   $S_0 \to S$
   $S \to ASC|Bb|\mathbf{b}$
   $A \to C|S$
   $B \to a$
   $C \to b|\varepsilon$

3. Remove ε-rule $C \to \varepsilon$

   $S_0 \to S$
   $S \to ASC|\mathbf{AS}|Bb|b$
   $A \to \varepsilon|S$
   $B \to a$
   $C \to b$

4. Remove ε-rule $A \to \varepsilon$

   $S_0 \to S$
   $S \to ASC|\mathbf{SC}|AS|\mathbf{S}|Bb|b$
   $A \to S$
   $B \to a$

$C \to b$

5. Remove unit rule $S_0 \to S$

$S_0 \to \mathbf{ASC}|\mathbf{SC}|\mathbf{AS}|\mathbf{Bb}|\mathbf{b}$
$S \to ASC|SC|AS|S|Bb|b$
$A \to S$
$B \to a$
$C \to b$

6. Remove unit rule $S \to S$

$S_0 \to \mathbf{ASC}|\mathbf{SC}|\mathbf{AS}|\mathbf{Bb}|\mathbf{b}$
$S \to ASC|SC|AS|Bb|b$
$A \to S$
$B \to a$
$C \to b$

7. Remove unit rule $A \to S$

$S_0 \to ASC|SC|AS|Bb|b$
$S \to ASC|SC|AS|Bb|b$
$A \to \mathbf{ASC}|\mathbf{SC}|\mathbf{AS}|\mathbf{Bb}|\mathbf{b}$
$B \to a$
$C \to b$

8. Add rule $A_1 \to AS$

$S_0 \to \mathbf{A_1}C|SC|AS|Bb|b$
$S \to \mathbf{A_1}C|SC|AS|Bb|b$
$A \to \mathbf{A_1}C|SC|AS|Bb|b$
$B \to a$
$C \to b$
$\mathbf{A_1} \to \mathbf{AS}$

9. Add rule $B_1 \to b$

$S_0 \to A_1C|SC|AS|B\mathbf{B_1}|b$
$S \to A_1C|SC|AS|B\mathbf{B_1}|b$
$A \to A_1C|SC|AS|B\mathbf{B_1}|b$
$B \to a$
$C \to b$
$A_1 \to AS$
$\mathbf{B_1} \to \mathbf{b}$

Example 13. Convert the CFG from Example 1 to an equivalent one in Chomsky Normal Form.

Example 14. Convert the CFG from Example 3 to an equivalent one in Chomsky Normal Form.

4 Pushdown Automata

After the DFA/NFA, the pushdown automaton (PDA) represents the next level of computational power for automata. A PDA is more powerful than DFA because i) every regular language can be accepted by a PDA, and ii) there are nonregular languages that can also be accepted by PDA's. In fact, the definition for a PDA is quite similar to that for an NFA. The only difference is that a PDA has access to a stack memory for which it is able to push and pop symbols from the stack.

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

Q is a finite set with each member of Q called a **state**

Σ a finite alphabet that is used to represent an input word

Γ a finite **stack alphabet** that represent the symbols that may be pushed on to the stack

δ a transition function

$$\delta : Q \times \Sigma_\varepsilon \times \Gamma_\varepsilon \to \mathcal{P}(Q \times \Gamma_\varepsilon).$$

that takes as input a

$$(\text{state}, \text{input symbol}, \text{stack symbol})$$

triple and maps it to a subset of (state, stack symbol) pairs, where input triple represents the i) current state, ii) input symbol being read, and iii) stack symbol being read (and simultaneously popped), and an output pair represents i) the next state and ii) the next stack symbol to be pushed on to the stack.

$q_0 \in Q$ is the **initial state**.

$F \subseteq Q$ is the set of **accepting states**.

Some notes.

1. The set Σ_ε is shorthand for $\Sigma \cup \{\varepsilon\}$. Γ_ε is similarly defined. For example, the triple (q, ε, x) means that the current state is q, no input symbol is being read, and x is being read/popped from the stack, while (q, x, ε) means is similar, but now input symbol x is being read, and the top of the stack is not being read/popped.

2. Notice that the above definition incorporates nondeterminism, since their are possibly multiple (next state, stack symbol) pairs. In addition, it is possible to simutaneously execute two different inputs to δ (why?).

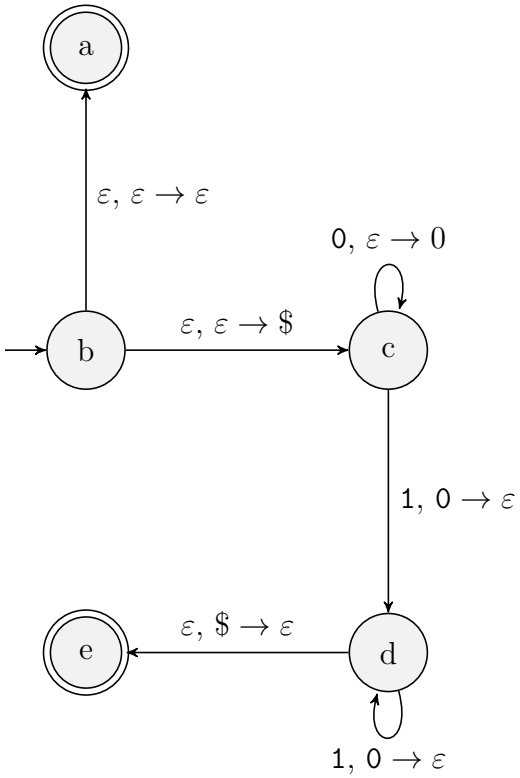
3. Technically, this definition is for that of an NPDA. DPDA's may also be defined and are of great practical importance. However, their study is considered somewhat more advanced and is not considered here. In this lecture, PDA is short for NPDA.

Like NFA's, a PDA may be represented with a state diagram using an almost identical notation. However, each edge label must be of the form

$$x, y \to z,$$

where x is the read input symbol, y is the read/popped stack symbol, and z is the pushed stack symbol.

Example 15. Provide the 6-tuple for the PDA whose state diagram is shown below and that accepts the binary language $\{0^n1^n | n \geq 0\}$. Recall that this language is nonregular.



PDA computation. Let $M = (Q, \Sigma, \Gamma, \delta, q_0, F)$ be a PDA, and $w = w_1 w_2 \cdots w_m$ be a word in Σ^*. Then the **computation of M on input** w is a tree $T(M, w)$ for which each node is labeled with a **configuration**, i.e. a triple (q, r, st), where $q \in Q$, is the current state, $r \in \Sigma_\varepsilon$ is the next symbol of w to be read, and $st \in \Gamma^*$ is the current stack configuration, s being the top stack symbol, and t representing the rest of the symbols on the stack. The tree is defined recursively as follows.

Base case The root of $T(M, w)$ is labeled with (q_0, w_1, ε), where q_0 is the initial state of M. Again, the computation starts in the initial state with an empty stack, and w_1 is the next input symbol to be read.

Recursive case Let n be a node of $T(M, w)$ that is labeled with (q, r, st), where $q \in Q$, $r = w_i$, for some $1 \leq i \leq m$, $s \in \Gamma_\varepsilon$ and $t \in \Gamma^*$. We now describe the kinds of children that n can have. In what follows we assume $q' \in Q$, $r' = w_{i+1}$ if $i < m$, and $r' = \varepsilon$, otherwise. Also, assume $u \in \Gamma$.

Case 1: Read only. For each pair $(q', \varepsilon) \in \delta(q, r, \varepsilon)$, there is a child node labeled with (q', r', st).

Case 2: Read and pop. For each pair $(q', \varepsilon) \in \delta(q, r, s)$, there is a child node labeled with (q', r', t).

Case 3: Read and push. For each pair $(q', u) \in \delta(q, r, \varepsilon)$, there is a child node labeled with (q', r', ust).

Case 4: Read, pop and push. For each pair $(q', u) \in \delta(q, r, s)$, there is a child node labeled with (q', r', ut).

Case 5: No read and pop. For each pair $(q', \varepsilon) \in \delta(q, \varepsilon, s)$, there is a child node labeled with (q', r, t).

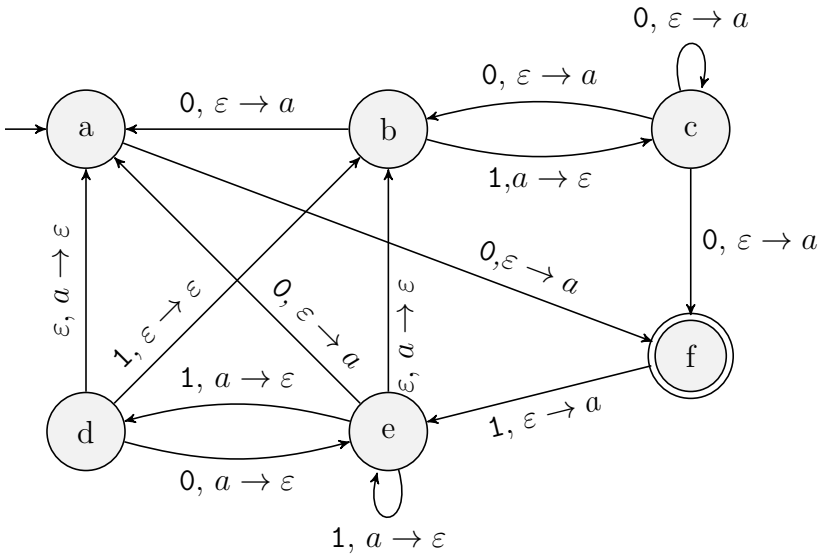
Case 6: No read and push. For each pair $(q', u) \in \delta(q, \varepsilon, \varepsilon)$, there is a child node labeled with (q', r, ust).

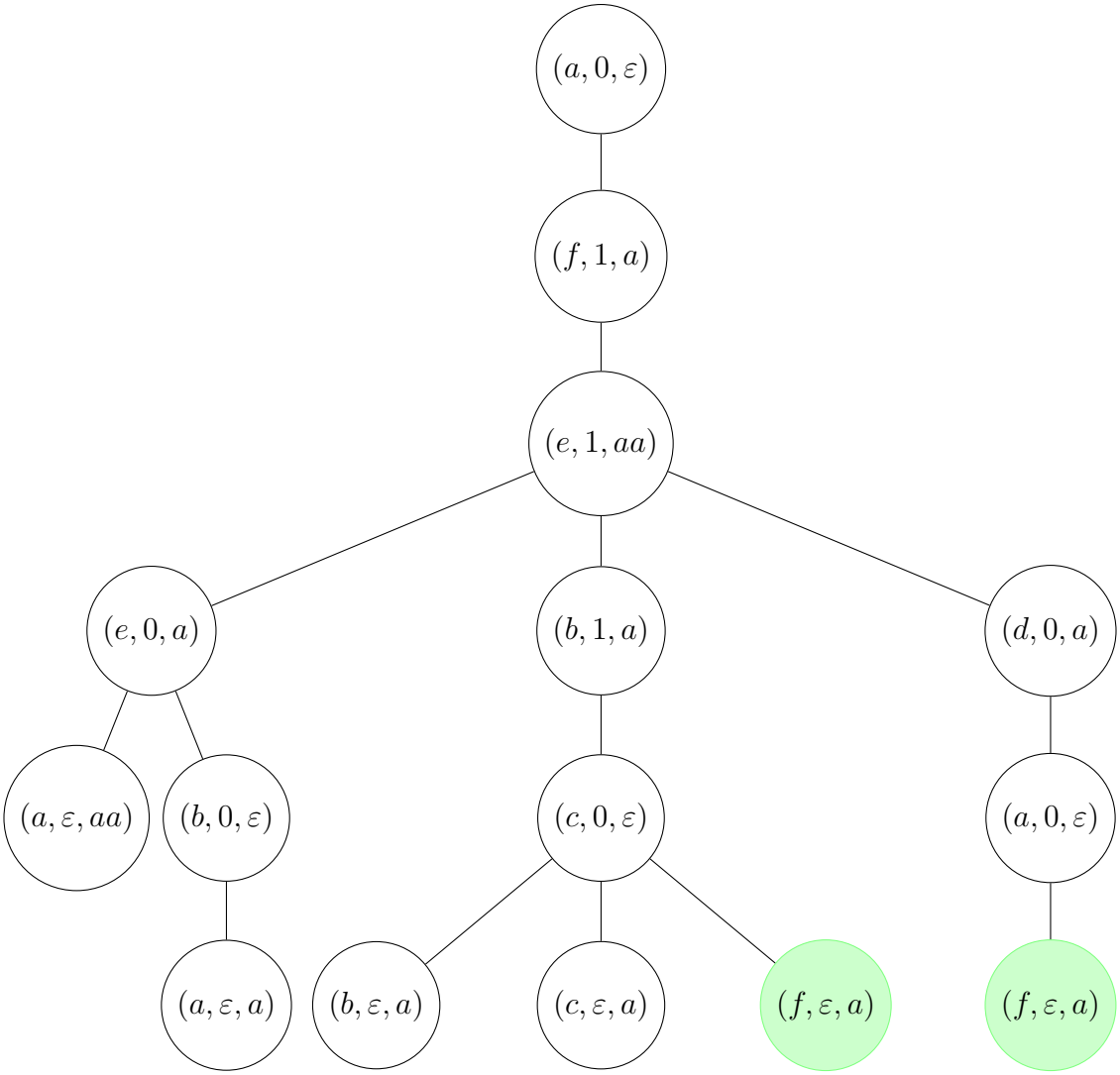
Case 7: No read, pop and push. For each pair $(q', u) \in \delta(q, \varepsilon, s)$, there is a child node labeled with (q', r, ut).

Case 8: No read, no pop, no push. For each pair $(q', \varepsilon) \in \delta(q, \varepsilon, \varepsilon)$, there is a child node labeled with (q', r, st).

Finally, M accepts input w iff there is a branch of $T(M, w)$ whose leaf node is labeled with the configuration (q, r, st), where $q \in F$ and $r = \varepsilon$. In other words, there is some branch that terminates with i) an acccepting state, ii) having read all the input, and iii) there are no further steps (e.g. further pushes and pops) that can be taken in the computation. Such a branch is called an **accepting branch**. If $T(M, w)$ has no accepting branch, then M rejects w. Finally, $L(M)$ denotes the set of input words accepted by M.

Example 16. For the PDA M with state diagram shown below, the computation tree $T(M, 0110)$ is shown on the next page.





Accepting branches end in green

4.1 Designing PDA's

Example 17. Draw the state diagram and provide the 6-tuple for a PDA that accepts the binary language that has an equal number of 1's and 0's.

Example 18. Let $L = \{x^i y^j z^k | i, j, k \geq 0 \text{ and } j = 2i \text{ or } k = 2j\}$. Provide the state diagram of a PDA that accepts L.

5 On the Equivalence of PDA's and CFG's

In this section we show that i) every CFL can be accpeted by some PDA and ii) for every language L accepted by a PDA, there is a grammar G for which $L = L(G)$.

Theorem 1. Every CFL is accepted by some PDA.

Proof of Theorem 1. Let $G = (V, \Sigma, R, S)$ and $L = L(G)$ be its associated CFL. We design a PDA M that accepts L as follows. To begin, the delimiter stack symbol \$ is first pushed on to the stack, followed by variable S being pushed, followed by entering a self-looping main state q_{loop} that allows for the following kinds of transitions.

Terminal Matching If $s \in \Sigma$ is the input symbol being read and is also on top of the stack, then s is both read and popped from the stack.

Variable Substitution If variable A is on top of the stack and $A \to u$ is a rule, then A is popped from the stack and the symbols of u are pushed on to the stack in reverse order (note: this may require several push transitions before returning to the q_{loop}. Note: this is where nondeterminism plays an important role, since a single variable A can be the head of more than one rule, and thus the computation will branch accordingly.

Read Delimiter If \$ is on top of the stack, then transition to an accept state from which no further input may be read.

Based on the above transitions, some thought and reflection should convince the reader that M accepts exactly those terminal words that are derived by G. This is because, in order to be accepted, the input symbols must match symbol-for-symbol some terminal word derived by the grammar.

The PDA defined above will be referred to as the **natural PDA** associated with CFG G.

Example 19. Draw a state diagram for the natural PDA associated with the grammar from Example 1 and having rules Consider the set of rules

$$R = \{S \to SS, S \to aSb, S \to \varepsilon\}.$$

Show an accepting branch of the computation tree for the word `abab`.

( )( ) ✓

( )((✗

1. Push S onto Stack: S
2. Guess a rule for S (nondeterministic)
   $S \to SS$
3. pop S, push S, push S:
   S S
   a b a b
   ↑
4. Guess a rule for S:
   $S \to aSb$
5. pop S, push b, push S, push a:
   a S b S
   a b a b
   ↑
6. read a, pop a:
   S b S
   a b a b
   ↑
7. Guess a Rule for S:
   $S \to \varepsilon$
8. pop S
   b S
   a b a b
   ↑
9. read b, pop b:
   S
   a b a b ←
   ↑

Repeat Steps 4-9 to read the final ab

Example 19 Continued.

5.1 Defining a CFG for a PDA

In this section we make the following assumptions about an arbitrary PDA M that we consider in the next theorem.

1. M has only one accept state q_f.
2. M either pushes or pops the stack on each transition, but not both, and
3. M always reaches an accept state with an empty stack.

Note that for any PDA M, there is always a PDA N that meets the above criteria and for which $L(M) = L(N)$. Thus, no generality is lost by making these assumptions.

Theorem 2. Let PDA $M = (Q, \Sigma, \Gamma, \delta, q_0, \{q_f\})$ satisfy the above simplifying assumptions. Then $L(M) = L(G)$ for some grammar $G = (V, \Sigma, R, S)$.

We now define a CFG $G = (V, \Sigma, R, S)$ for which $L(G) = L(M)$. The idea is to define, for every pair of states $p, q \in Q$, a variable A_{pq} so that A_{pq} has the ability to derive w if and only if starting in state p with the stack empty, a computation branch of M on input w transitions to state q, and upon reaching q has an empty stack. There are two cases to consider.

Case 1: During the computation the stack remains nonempty (until the final step). In this case we add the rule $A_{pq} \to w_1 A_{rs} w_n$, where we have $(r, t) \in \delta(p, w_1, \varepsilon)$ and $(q, \varepsilon) \in \delta(s, w_n, t)$. In words, we know that r is a candidate for the 2nd state of the computation (assuming $t \in \Gamma$ was pushed first), while s is a candidate for the next-to-last state of the computation, since it is a state for which w_n can be read, and for which t can be popped. Notice how t must be popped at the end, since it was the first symbol pushed, and the stack did not empty until the very end.

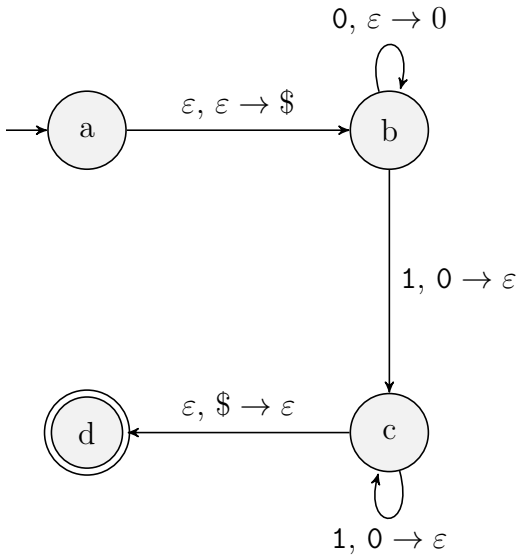
Case 2: During the computation the stack empties (before reaching the final state). Let r be the first state in which the stack is empty during the computation. Then there is no guarantee that the first symbol pushed is the last symbol popped. So we need a more conservative rule of the form $A_{pq} \to A_{pr} A_{rq}$. Thus, A_{pq} is allowing for A_{pr} to derive the first part of w, while A_{rq} will derive the second part of w. In other words, the computation is cut up into pieces, where each piece represents a computation that starts with an empty stack, and ends with an empty stack, with the stack remaining nonempty until the end.

The rules for Case 1 will be called type-1, while the rules for Case 2 are called type-2. Putting it all together, a CFG G for which $L(G) = L(M)$ will have start variable A_{q_0,q_f}, and the following rules:

1. For every pair of states $p, q \in Q$, the type-1 rules $A_{pq} \to aA_{rs}b$, for every $a, b \in \Sigma$ and $r, s \in Q$ for which $(r, t) \in \delta(p, a, \varepsilon)$ and $(q, \varepsilon) \in \delta(s, b, t)$.
2. For every triple of states $p, q, r \in Q$, the type-2 rules $A_{pq} \to A_{pr}A_{rq}$.
3. For each $p \in Q$, the rule $A_{pp} \to \varepsilon$

The grammar defined above will be referred to as the **natural grammar CFG** for PDA M.

Example 20. Recall the PDA from Example 15. Modify it so that it only accepts words of the form 0^n1^n, for all $n \geq 1$. Provide a natural grammar derivation of the word 000111.



Type 1 $A_{ad} \to A_{bc}$ Push \$ while moving from a to b. Pop \$ while moving from c to d.

$A_{bc} \to 0A_{bc}1$ Push 0 while moving from b to b. Pop 0 while moving from c to c.

$A_{bc} \to 0A_{bb}1$ Push last 0 while moving from b to b. Pop last zero while moving from b to c.

Type 2 None needed

ε **Rules** $A_{bb} \to \varepsilon$

Example 21. Suppose a PDA has the following configuration sequence:

$$(a, 1, \varepsilon), (b, 0, x), (c, \varepsilon, yx), (b, \varepsilon, zyx), (a, 0, zzyx), (c, 1, zyx), (d, 1, yx), (a, 0, zyx), (e, \varepsilon, yx),$$

$$(b, 1, x), (d, \varepsilon, \varepsilon).$$

Show the portion of the natural CFG that is used to derive the read input word 1001101. Note: here the 2nd component corresponds with what was read by the PDA, rather than what *could* have been read at that moment.

CFL Core Exercises

1. For the CFG defined in Example 1, provide a derivation for the following words.

   a. ababab

   b. aaababbbab

   c. aababaabbbaabb

2. For the CFG defined in Example 3, provide a derivation and parse tree for the following expressions.

   a. a

   b. $a + a$

   c. $a \times (a \times a)$

   d. $((a))$

3. Consider the following rules for some context free grammar G.

$$R \to XRX \mid S$$

$$S \to aTb \mid bTa$$

$$T \to XTX \mid X \mid \varepsilon$$

$$X \to a \mid b$$

   Do the following.

   a. Provide the set V of variables, the set Σ of terminals, and the start variable. Hint: we may assume that the start variable never appears in the body of a rule, unless the head of the rule is itself the start variable.

   b. Provide three words that are in $L(G)$ and three words that are not in $L(G)$.

   c. Which of the following statements are a) true as stated ? b) true only when $\Rightarrow$ is replaced by $\overset{*}{\Rightarrow}$? c) false even when $\Rightarrow$ is replaced by $\overset{*}{\Rightarrow}$?

      i. $T \Rightarrow aba$

      ii. $T \Rightarrow T$

      iii. $XXX \Rightarrow aba$

      iv. $X \Rightarrow aba$

      v. $T \Rightarrow XX$

      vi. $T \Rightarrow XXX$

      vii. $S \Rightarrow \varepsilon$

   d. Provide a succinct natural-language description of $L(G)$.

4. For each of the following context free languages, provide a set of rules that describes the language. Assume that the terminal set for each is $\{0,1\}$.

$S \to 0S1 \mid 0S0 \mid 1S0 \mid 1S1 \mid 00$

a. All binary words with at least three 1's
b. All binary words that start and end with the same symbol
c. All binary words having odd length
d. All binary words of positive even length and for which the middle two bits are 00
e. All binary words that are palindromes (i.e. read the same forwards as backwards)
f. All binary words for which there are twice as many 0's as 1's
g. The empty set
h. The set $\{\varepsilon\}$
i. All binary words for which there are more 1's than 0's.
j. The complement of $\{a^n b^n | n \geq 0\}$

5. For each of the following context free languages, provide a state diagram for a PDA that accepts the language. Assume that the terminal set for each is $\{0, 1\}$.

a. All binary words having an odd number of 1's.
b. All binary words that are palindromes (i.e. read the same forwards as backwards)
c. All binary words of positive even length and for which the middle two bits are 00
d. All binary words for which there are more 1's than 0's.
e. All binary words for which there are twice as many 0's as 1's.

Exercise Solutions

1. We have the following derivations.

a. ababab

$$S \Rightarrow SS \Rightarrow aSbS \Rightarrow abS \Rightarrow abSS \Rightarrow abaSbS \Rightarrow ababS \Rightarrow ababaSb \Rightarrow ababab.$$

b. aaababbbab

$$S \Rightarrow SS \Rightarrow aSbS \Rightarrow aaSbbS \Rightarrow aaSSbbS \Rightarrow aaaSbSbbS \Rightarrow aaabSbbS \Rightarrow aaabaSbbbS.$$
$$\Rightarrow aaababbbS \Rightarrow aaababbbaSb \Rightarrow aaababbbab.$$

c. aababaabbbaabb

$$S \Rightarrow SS \Rightarrow aSbS \Rightarrow aSSbS \Rightarrow aaSbSbS \Rightarrow aabSbS \Rightarrow aabSSbS$$
$$\Rightarrow aabaSbSbS \Rightarrow aababSbS \Rightarrow aababaSbbS \Rightarrow aababaaSbbbS \Rightarrow aababaabbbS$$
$$\Rightarrow aababaabbbaSb \Rightarrow aababaabbbaaSbb \Rightarrow aababaabbbaabb.$$

2. For the CFG defined in Example 3, provide a derivation and parse tree for the following expressions.

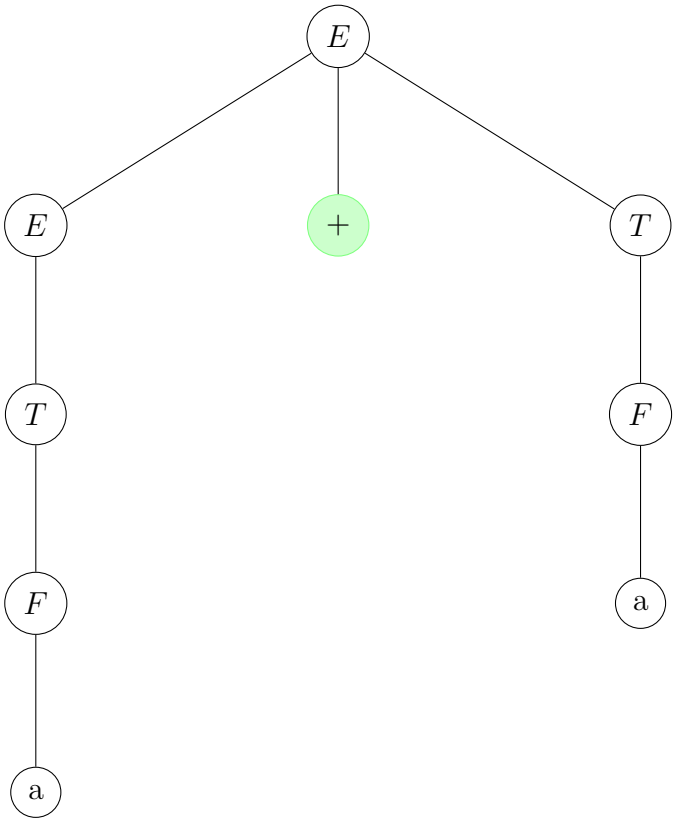
a. a

$$E \Rightarrow T \Rightarrow F \Rightarrow \text{a}.$$



b. $a + a$

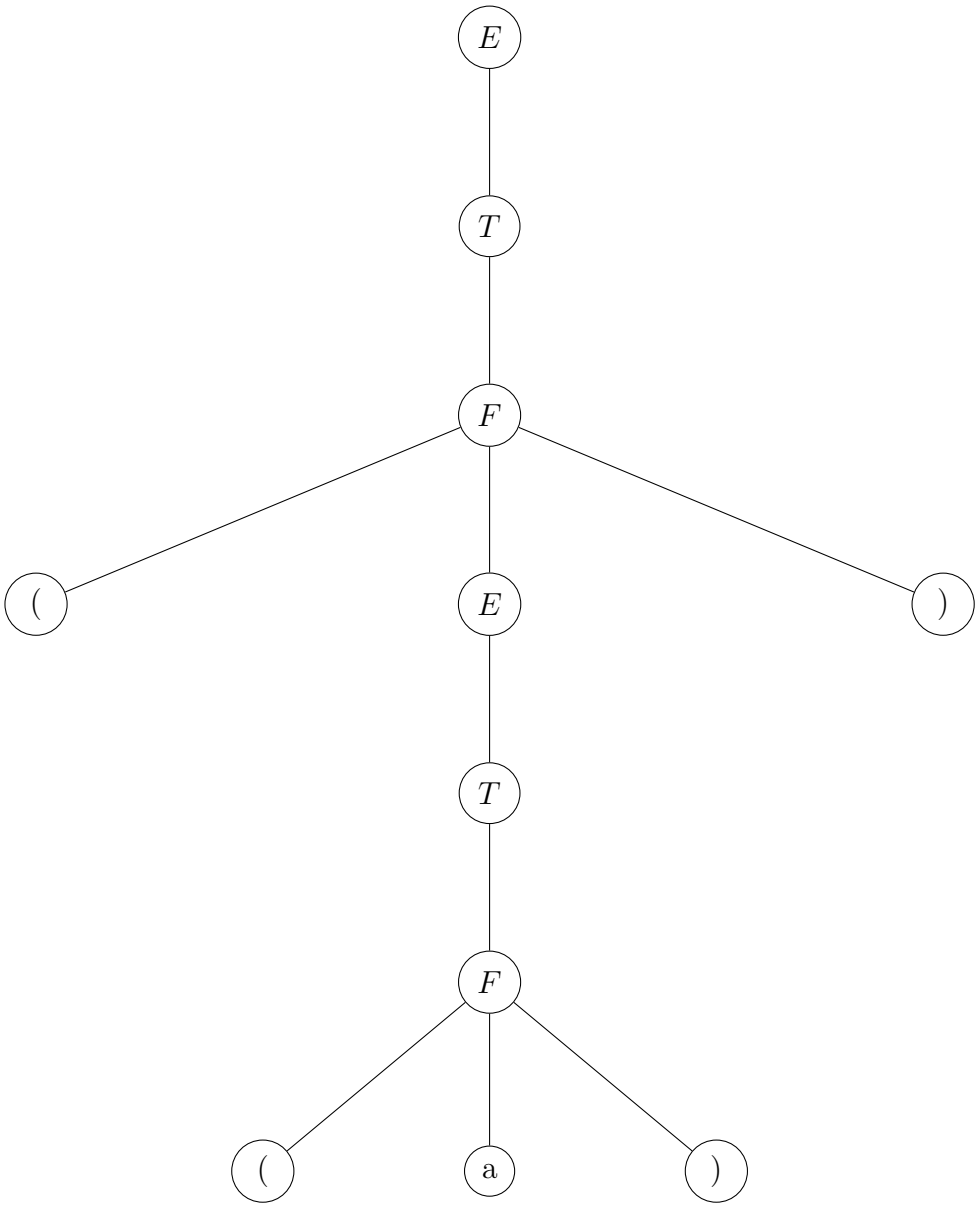
$$E \Rightarrow E + T \Rightarrow T + T \Rightarrow F + T \Rightarrow \text{a} + T \Rightarrow \text{a} + F \Rightarrow \text{a} + \text{a}.$$



c. $a \times (a \times a)$

d. $((a))$

$$E \Rightarrow T \Rightarrow F \Rightarrow (E) \Rightarrow (T) \Rightarrow (F) \Rightarrow ((E)) \Rightarrow T \Rightarrow F \Rightarrow ((\text{a})).$$



3. We have the following.

   a. $V = \{R, S, T, X\}$, $\Sigma = \{a, b\}$, and R is the start variable.

   b. $aa, bb, aab \in L(G)$ and $\varepsilon, a, b \notin L(G)$.

   c. Statements iv and vii are false even when "yields" is replaced by "derives". The remaining statements are true when "yields" is replaced by "derives". For iii) it's important to note that, by definition, only one variable replacement is allowed per step, and so it would take three steps to derive aba.

   d. All words consisting of a's and b's and having length at least two.

4. We have the following rule sets.

a. All binary words with at least three 1's

$$S \to 1S1S1S \mid 0 \mid 1 \mid \varepsilon$$

b. All binary words that start and end with the same symbol

$$S \to 0T0 \mid 1T1 \mid 0 \mid 1 \mid \varepsilon$$
$$T \to 0T \mid 1T \mid \varepsilon$$

c. All binary words having odd length

$$S \to 0S0 \mid 0S1 \mid 1S0 \mid 1S1 \mid 0 \mid 1$$

d. All binary words of positive even length and for which the middle two bits are 00

$$S \to 0S0 \mid 0S1 \mid 1S0 \mid 1S1 \mid 00$$

e. All binary words that are palindromes (i.e. read the same forwards as backwards)

$$S \to 0S0 \mid 1S1 \mid 0 \mid 1 \mid 00 \mid 11$$

f. All binary words for which there are twice as many 0's as 1's

$$S \to S0S0S1S \mid S1S0S0S \mid S0S1S0S \mid \varepsilon$$

Notice how the rules provide maximum flexibility in terms of which 1 and two 0's are going to be written. This is needed because there could be any ordering of the 1's and 0's. For example, we could have 110000000011. Provide a derivation of this word using the above rules.

g. Any CFG $G = (V, \Sigma, R, S)$ for which $R = \emptyset$.

h. The set $\{\varepsilon\}$

$$S \to \varepsilon$$

i. All binary words for which there are more 1's than 0's.

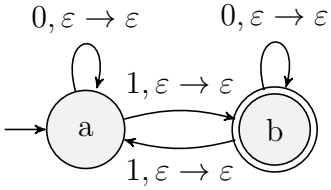
$$S \to S1S0S \mid S0S1S \mid 1S \mid \varepsilon$$

j. The complement of $\{a^n b^n | n \geq 0\}$. Hint. Notice that the set is equal to the subset of $\{a, b\}^*$ having the form $A \cup B \cup C$, where A consists of odd-length words, B consists of nonempty even-length words for which there is at least one b in the first half of the word, and C consists of nonempty even-length words for which there is at least one a in the second half of the word. Then we have

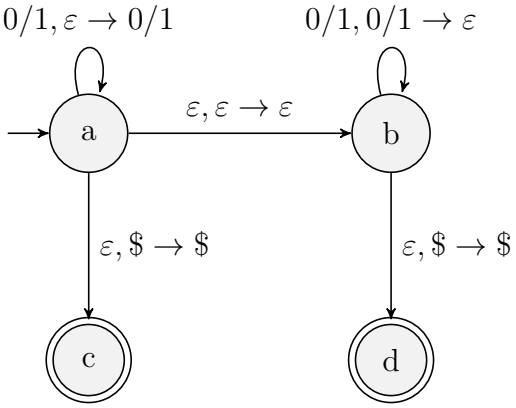
$$S \to X \mid Y \mid Z$$

where X, Y, and Z are the respective start symbols for CFG's that describe languages A, B, and C, respectively. Now provide GFG's for each of these languages and take the union the grammars (making sure that the variable set for each grammar is pairwise disjoint with the other two variable sets).

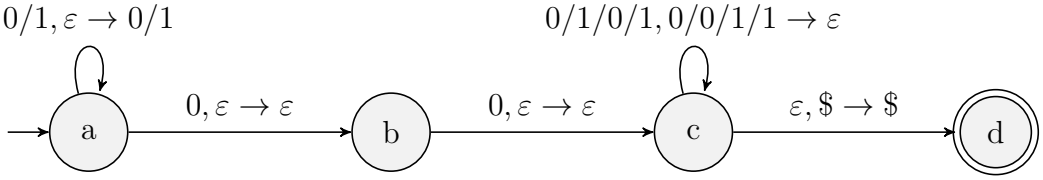
a. All binary words having an odd number of 1's.



b. All binary words that are palindromes (i.e. read the same forwards as backwards)

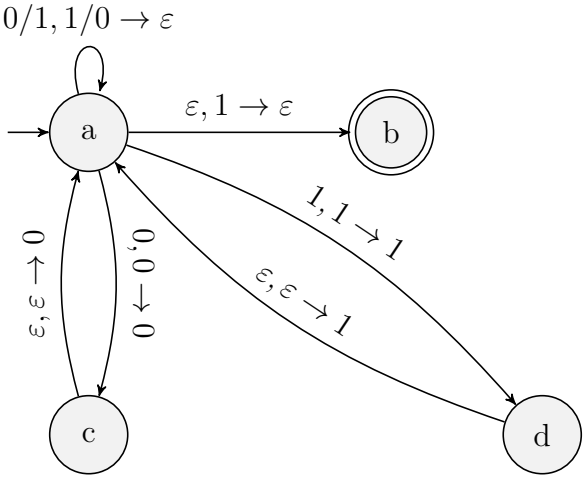


c. All binary words of positive even length and for which the middle two bits are 00.



d. All binary words for which there are more 1's than 0's. For this problem, show the δ-transition function evaluations for each edge of the PDA.

e. All binary words that are palindromes (i.e. read the same forwards as backwards)



f. Instead of drawing the state diagram, we give a description of each state (i through xii)

i. Initial state, empty stack

ii. Accepting state, empty stack

iii. Stack consists of one or more 1's

iv. A 0 has been read and a 1 has been popped, making a deficit of one 0. When in this state no input is read, and the next state is either v or xii.

v. Stack consists of one or more 1's and there is a deficit of one 0.

vi. Stack consists of one 0.

vii. One more 0 needs to be popped from the stack.

viii. Decide if the stack is empty or if there is at least one 0 on the stack (this requires popping the 0 from the stack).

ix. Push the 0 that was popped from the stack if the stack is currently empty. Otherwise, defer pushing the 0 back on to the stack to state x.

x. Push the 0 back on to the stack and go to state xi

xi. Stack consists of two or more 0's.

xii. Stack is empty and there is a deficit of one 0.