Computational Theory

Context-free Languages

Curtis Larsen

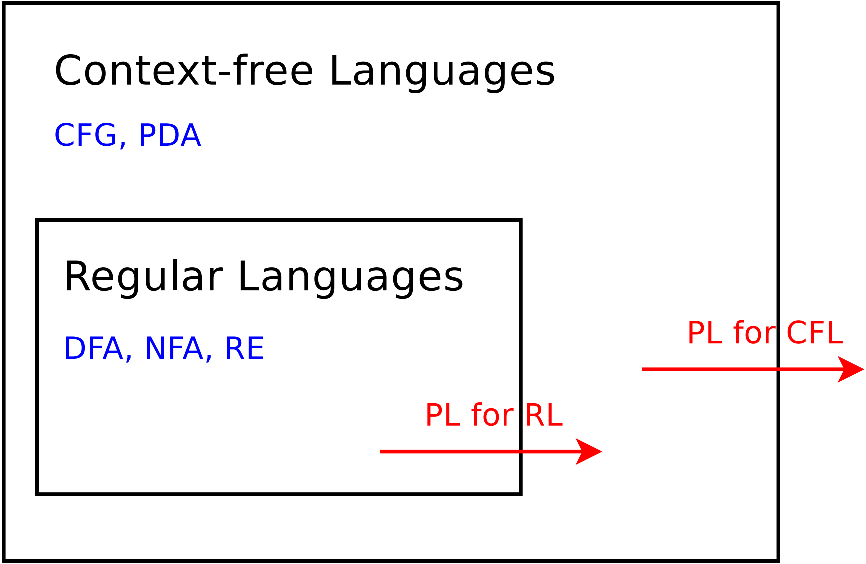
Utah Tech University—Computing

Fall 2024

Adapted from notes by Russ Ross

Adapted from notes by Harry Lewis

Summary


Context-free Languages
CFG, PDA
Regular Languages
DFA, NFA, RE
PL for RL
PL for CFL

Context-free Grammars

Reading: Sipser §2.1 Context-free Grammars

Formal Definitions for CFGs

- A CFG $G = (V, \Sigma, R, S)$

  V = Finite set of **variables** (or **nonterminals**)

  Σ = The alphabet, a finite set of **terminals** ($V \cap \Sigma = \emptyset$).

  R = A finite set of **rules**, each of the form $A \to w$
  for $A \in V$ and $w \in (V \cup \Sigma)^*$.

  S = The **start variable**, $S \in V$

  e.g. $(\{S\}, \{a, b\}, \{S \to aSb, S \to \varepsilon\}, S)$

Formal Definitions for CFGs

- A CFG $G = (V, \Sigma, R, S)$

  $V =$ Finite set of **variables** (or **nonterminals**)

  $\Sigma =$ The alphabet, a finite set of **terminals** ($V \cap \Sigma = \emptyset$).

  $R =$ A finite set of **rules**, each of the form $A \to w$
  for $A \in V$ and $w \in (V \cup \Sigma)^*$.

  $S =$ The **start variable**, $S \in V$

  e.g. $(\{S\}, \{a, b\}, \{S \to aSb, S \to \varepsilon\}, S)$

- **Derivations:** For $\alpha, \beta \in (V \cup \Sigma)^*$ (strings of terminals and nonterminals),

  $\alpha \Rightarrow \beta$ ("α **yields** β") if $\alpha = uAv, \beta = uwv$, for some $u, v \in (V \cup \Sigma)^*$, and R contains rule $A \to w$.

  $\alpha \stackrel{*}{\Rightarrow} \beta$ ("α **derives** β") if there is a sequence $\alpha_0, \ldots, \alpha_k$ for $k \geq 0$ such that
  $\alpha_0 = \alpha$, $\alpha_k = \beta$, and $\alpha_{i-1} \Rightarrow \alpha_i$ for each $i = 1, \ldots, k$.

Definition of Context-free Language

- The set of strings that can be derived from a context-free grammar is the language generated by the grammar.
  $L(G) = \{w | w$ can be derived by G $\}$
  $L(G) = \{w \in \Sigma^* : S \stackrel{*}{\Rightarrow} w\}$ (strings of terminals only!)
- Any language that can be generated by a context-free grammar is a context-free language (CFL).

Example CFG G_1

- G_1:

$$A \to 0A1$$
$$A \to B$$
$$B \to \#$$

Example CFG G_1

- G_1:

$$
\begin{aligned}
A &\rightarrow 0A1 \\
A &\rightarrow B \\
B &\rightarrow \#
\end{aligned}
$$

- Variables? Terminals? Rules? Start variable?

Example CFG G_1

- G_1:

$$A \to 0A1$$
$$A \to B$$
$$B \to \#$$

- Variables? Terminals? Rules? Start variable?
- Alternate G_1:

$$A \to 0A1|B$$
$$B \to \#$$

Example CFG G_1

- G_1:

$$
\begin{aligned}
A &\rightarrow 0A1 \\
A &\rightarrow B \\
B &\rightarrow \#
\end{aligned}
$$

- Variables? Terminals? Rules? Start variable?
- Alternate G_1:

$$
\begin{aligned}
A &\rightarrow 0A1|B \\
B &\rightarrow \#
\end{aligned}
$$

- Strings derived from G_1?

Example CFG G_1

- G_1:

$$A \to 0A1$$
$$A \to B$$
$$B \to \#$$

- Variables? Terminals? Rules? Start variable?
- Alternate G_1:

$$A \to 0A1 | B$$
$$B \to \#$$

- Strings derived from G_1?

  $\#$, $0\#1$, $00\#11$, $000\#111$, ...

Example CFG G_1

- G_1:

$$A \to 0A1$$
$$A \to B$$
$$B \to \#$$

- Variables? Terminals? Rules? Start variable?
- Alternate G_1:

$$A \to 0A1|B$$
$$B \to \#$$

- Strings derived from G_1?

  $\#$, $0\#1$, $00\#11$, $000\#111$, ...
- $L(G_1) =$?

Example CFG G_1

- G_1:

$$A \rightarrow 0A1$$
$$A \rightarrow B$$
$$B \rightarrow \#$$

- Variables? Terminals? Rules? Start variable?
- Alternate G_1:

$$A \rightarrow 0A1|B$$
$$B \rightarrow \#$$

- Strings derived from G_1?

  $\#$, $0\#1$, $00\#11$, $000\#111$, ...
- $L(G_1) =$?

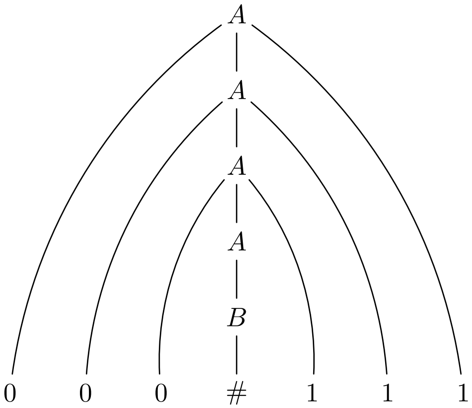
  $\{0^n\#1^n | n \geq 0\}$

Parse Trees

- A parse tree is a pictorial representation of a single derivation.
- The parse tree for $w = 000\#111$, derived from G_1.

G_1:

$$A \rightarrow 0A1$$
$$A \rightarrow B$$
$$B \rightarrow \#$$



More examples of CFGs

- Arithmetic Expressions

G_2:

$$\begin{aligned}
EXPR &\rightarrow TERM \mid EXPR + TERM \\
TERM &\rightarrow TERM * FACTOR \mid FACTOR \\
FACTOR &\rightarrow (EXPR) \mid x \mid y
\end{aligned}$$

More examples of CFGs

- Arithmetic Expressions

  G_2:

$$\begin{aligned} EXPR &\to TERM \mid EXPR + TERM \\ TERM &\to TERM * FACTOR \mid FACTOR \\ FACTOR &\to (EXPR) \mid x \mid y \end{aligned}$$

- Derived strings?

More examples of CFGs

- Arithmetic Expressions

  G_2:

$$\begin{aligned} EXPR &\rightarrow TERM \mid EXPR + TERM \\ TERM &\rightarrow TERM * FACTOR \mid FACTOR \\ FACTOR &\rightarrow (EXPR) \mid x \mid y \end{aligned}$$

- Derived strings?
- $L(G_2)$?

More examples of CFGs

- Arithmetic Expressions

  G_2:

$$
\begin{aligned}
EXPR &\to TERM \mid EXPR + TERM \\
TERM &\to TERM * FACTOR \mid FACTOR \\
FACTOR &\to (EXPR) \mid x \mid y
\end{aligned}
$$

- Derived strings?
- $L(G_2)$?
- Parse tree for some string?

More examples of CFGs

- $L(G_3) = \{x \in \{(,)\}^*$: parentheses in x are properly 'balanced'$\}$.
  $G_3 =$?

- $L(G_4) = \{x \in \{a, b\}^* : x$ has the same # of a's and b's$\}$.
  $G_4 =$?

Chomsky Normal Form

Def: A grammar is in **Chomsky normal form** if

- the only possible rule with ε as the RHS is $S \to \varepsilon$
  (Of course, this rule occurs iff $\varepsilon \in L(G)$)
- Every other rule is of the form
  1. $X \to YZ$
     where X, Y, Z are variables
  2. $X \to \sigma$
     where X is a variable and σ is a single terminal symbol

Transforming a CFG into Chomsky Normal Form

Definitions:

- **ε-rule:** one of the form $X \to \varepsilon$
- **Long Rule:** one of the form $X \to \alpha$ where $|\alpha| > 2$
- **Unit Rule:** one of the form $X \to Y$
  where $X, Y \in V$
- **Terminal-Generating Rule:** one of the form $X \to \alpha$
  where $\alpha \notin V^*$ and $|\alpha| \geq 1$ (α has at least one terminal)

Eliminate non-Chomsky-Normal-Form Rules in Order:

1. All ε-rules, except maybe $S \to \varepsilon$
2. All unit rules
3. All long rules
4. All terminal-generating rules

Note: while eliminating rules of type j, we make sure not to reintroduce rules of type $i < j$.

Eliminating ε-Rules

0. Ensure start variable does not appear on the RHS of any rule (by adding new start variable with rule $S \to S_{old}$ if necessary).
1. To eliminate ε-rules, repeatedly do the following:
   a. Pick a ε-rule $Y \to \varepsilon$ and remove it.
   b. Given a rule $X \to \alpha$, where α contains n occurrences of Y, replace it with 2^n rules in which $0, \dots, n$ occurrences are replaced by ε. (Do not add $X \to \varepsilon$ if previously removed.)

   e.g.

$$X \to aYZbY \qquad \Rightarrow$$

   (Why does this terminate?)

Eliminating ε-Rules

0. Ensure start variable does not appear on the RHS of any rule (by adding new start variable with rule $S \to S_{old}$ if necessary).
1. To eliminate ε-rules, repeatedly do the following:
   a. Pick a ε-rule $Y \to \varepsilon$ and remove it.
   b. Given a rule $X \to \alpha$, where α contains n occurrences of Y, replace it with 2^n rules in which $0, \ldots, n$ occurrences are replaced by ε. (Do not add $X \to \varepsilon$ if previously removed.)

   e.g.

$$X \to aYZbY \qquad \Rightarrow \qquad \begin{aligned} X &\to aYZbY \\ X &\to aZbY \\ X &\to aYZb \\ X &\to aZb \end{aligned}$$

   (Why does this terminate?)

Eliminating Unit and Long Rules

2. To eliminate unit rules, repeatedly do the following:
    a. Pick a unit rule $A \to B$ and remove it.
    b. For every rule $B \to u$, add rule $A \to u$ unless this is a unit rule that was previously removed.
3. To eliminate long rules, repeatedly do the following:
    a. Remove a long rule $A \to u_1 u_2 \cdots u_k$, where each $u_i \in V \cup \Sigma$ and $k \geq 3$.
    b. Replace with rules $A \to u_1 A_1, A_1 \to u_2 A_2, \ldots, A_{k-2} \to u_{k-1} u_k$, where $A_1, \ldots, A_{k-2}$ are newly introduced variables used only in these rules.

Eliminating Terminal-Generating Rules

4. To eliminate terminal-generating rules:
   a. For each terminal a introduce a new nonterminal A.
   b. Add the rules $A \to a$
   c. "Capitalize" existing rules, e.g.
      replace $X \to aY$
      with $X \to AY$

Example of Transformation to Chomsky Normal Form

Starting grammar:

$$S \to XX$$
$$X \to aXb \mid \varepsilon$$

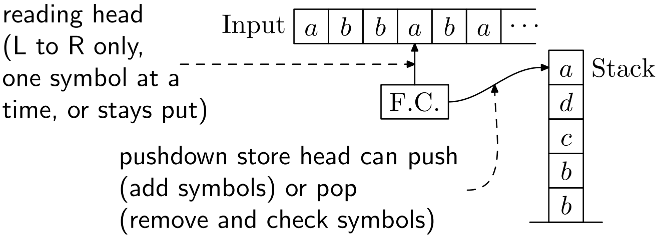
Pushdown Automata

Reading: Sipser §2.2.

Pushdown Automata

A **pushdown automaton** = a finite automaton + “pushdown store”.

The **pushdown store** is a stack of symbols of unlimited size which the machine can read and alter only at the top.



Transitions of PDA are of form $(q, \sigma, \gamma) \mapsto (q', \gamma')$, which means:

If in state q with σ on the input tape and γ on top of the stack, replace γ by γ' on the stack and enter state q' while advancing the reading head over σ.

(Nondeterministic) PDA for "even palindromes"

$\{ww^{\mathcal{R}} : w \in \{a, b\}^*\}$

$(q, a, \varepsilon) \mapsto (q, a)$	Push a's
$(q, b, \varepsilon) \mapsto (q, b)$	and b's
$(q, \varepsilon, \varepsilon) \mapsto (r, \varepsilon)$	switch to other state
$(r, a, a) \mapsto (r, \varepsilon)$	pop a's matching input
$(r, b, b) \mapsto (r, \varepsilon)$	pop b's matching input

So the precondition (q, σ, γ) means that

- the next $|\sigma|$ symbols (0 or 1) of the input are σ and
- the top $|\gamma|$ symbols (0 or 1) on the stack are γ

(Nondeterministic) PDA for "even palindromes"

$\{ww^{\mathcal{R}} : w \in \{a, b\}^*\}$

$(q, a, \varepsilon) \mapsto (q, a)$	Push a's
$(q, b, \varepsilon) \mapsto (q, b)$	and b's
$(q, \varepsilon, \varepsilon) \mapsto (r, \varepsilon)$	switch to other state
$(r, a, a) \mapsto (r, \varepsilon)$	pop a's matching input
$(r, b, b) \mapsto (r, \varepsilon)$	pop b's matching input

Need to test whether stack empty: push $\$$ at beginning and check at end.

$$(q_0, \varepsilon, \varepsilon) \mapsto (q, \$)$$
$$(r, \varepsilon, \$) \mapsto (q_f, \varepsilon)$$

Language acceptance with PDAs

A PDA **accepts** an input string

If there is a computation that starts

- in the start state
- with reading head at the beginning of string
- with the stack empty

and ends

- in a final state
- with all the input consumed

A PDA computation becomes “blocked” (i.e. “dies”) if

- no transition matches **both** the input and stack

Formal definition of a PDA

$M = (Q, \Sigma, \Gamma, \delta, q_0, F)$

Q = states

Σ = input alphabet

Γ = stack alphabet

δ = transition function

$$Q \times (\Sigma \cup \{\varepsilon\}) \times (\Gamma \cup \{\varepsilon\}) \to \mathcal{P}(Q \times (\Gamma \cup \{\varepsilon\}))$$

q_0 = start state

F = final states

Computation by a PDA

- M **accepts** w if we can write $w = w_1 \cdots w_m$,
  where each $w_i \in \Sigma \cup \{\varepsilon\}$,
  and there is a sequence of states $r_0, \ldots, r_m$
  and **stack strings** $s_0, \ldots, s_m \in \Gamma^*$ that satisfy
  1. $r_0 = q_0$ and $s_0 = \varepsilon$.
  2. For each i, $(r_{i+1}, \gamma') \in \delta(r_i, w_{i+1}, \gamma)$ where $s_i = \gamma t$ and $s_{i+1} = \gamma' t$ for some $\gamma, \gamma' \in \Gamma \cup \{\varepsilon\}$ and $t \in \Gamma^*$.
  3. $r_m \in F$.
- $L(M) = \{w \in \Sigma^* : M \text{ accepts } w\}$.

PDA for $\{w \in \{a, b\}^* : \#_a(w) = \#_b(w)\}$

PDA for $\{w \in \{a,b\}^* : \#_a(w) = \#_b(w)\}$

Strategy:

- Keep $|\#_a(w) - \#_b(w)| = n$ on stack in form of $1^n\$$.
- Keep the **sign** of $\#_a(w) - \#_b(w)$ in the state:

  $+$ or $0 \Rightarrow$ state q_+

  $-$ or $0 \Rightarrow$ state q_-

Equivalence of CFGs and PDAs

Thm: The class of languages recognized by PDAs is the CFLs.

I. For every CFG G,
there is a PDA M
with $L(M) = L(G)$

II. For every PDA M,
there is a CFG G
with $L(G) = L(M)$

Proof that every CFL is accepted by some PDA

Let $G = (V, \Sigma, R, S)$

We'll allow a generalized sort of PDA that can push **strings** onto stack.

E.g., $(q, a, b) \mapsto (r, cd)$

Proof that every CFL is accepted by some PDA

Let $G = (V, \Sigma, R, S)$

We'll allow a generalized sort of PDA that can push **strings** onto stack.

E.g., $(q, a, b) \mapsto (r, cd)$

The corresponding PDA has just 3 states:

$q_{start} \sim$ start state

$q_{loop} \sim$ "main loop" state

$q_{accept} \sim$ final state

Stack alphabet $= V \cup \Sigma \cup \{\$\}$

CFL $\Rightarrow$ PDA, Continued: The Transitions of the PDA

Transitions:

- $\delta(q_{start}, \varepsilon, \varepsilon) = \{(q_{loop}, S\$)\}$

  "Start by putting $S\$$ on the stack, & go to q_{loop}"

- $\delta(q_{loop}, \varepsilon, A) = \{(q_{loop}, w)\}$ for each rule $A \to w$

  "Remove a variable from the top of the stack and replace it with a corresponding righthand side"

- $\delta(q_{loop}, \sigma, \sigma) = \{(q_{loop}, \varepsilon)\}$ for each $\sigma \in \Sigma$

  "Pop a terminal symbol from the stack if it matches the next input symbol"

- $\delta(q_{loop}, \varepsilon, \$) = \{(q_{accept}, \varepsilon)\}$.

  "Go to accept state if stack contains only $\$$."

Example

- Consider grammar G with rules $\{S \to aSb, S \to \varepsilon\}$
  (so $L(G) = \{a^n b^n : n \geq 0\}$)
- Construct PDA

  $M = (\{q_{start}, q_{loop}, q_{accept}\}, \{a, b\}, \{a, b, S, \$\}, \delta, q_{start}, \{q_{accept}\})$

  Transition Function δ:
- Derivation $S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow aabb$

  Corresponding Computation:

The Dual Bottom-Up CFG $\to$ PDA Construction

- $\delta(q_{\text{start}}, \varepsilon, \varepsilon) = \{(q_{\text{loop}}, \$)\}$

  "Start by putting $\$$ on the stack, & go to q_{loop}"

- $\delta(q_{\text{loop}}, \sigma, \varepsilon) = \{(q_{\text{loop}}, \sigma)\}$ for each $\sigma \in \Sigma$

  "Shift input symbols onto the stack"

- $\delta(q_{\text{loop}}, \varepsilon, w^{\mathcal{R}}) = \{(q_{\text{loop}}, A) : A \to w)$ is a rule of $G\}$

  "Reduce right-hand sides on the stack to corresponding left-hand sides"

- $\delta(q_{\text{loop}}, \varepsilon, S\$) = \{q_{\text{accept}}, \varepsilon)\}$

  "Accept if the stack consists just of S above the bottom-marker"

Proof that for every PDA M there is a CFG G such that $L(M) = L(G)$

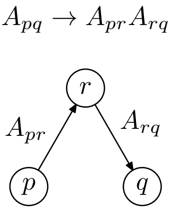
- First modify PDA M so that
  - Single accept state.
  - All accepting computations end with empty stack.
  - In every step, push a symbol or pop a symbol but not both.

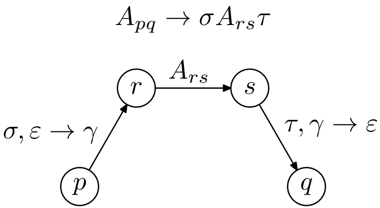
Design of the grammar G equivalent to PDA M

- Variables: A_{pq} for every two states p, q of M
- Goal: A_{pq} generates all strings that can take M from p to q, beginning and ending with an empty stack.
- Rules:
  - For all states p, q, r, $A_{pq} \to A_{pr} A_{rq}$
  - For states p, q, r, s and $\sigma, \tau \in \Sigma$,
    $A_{pq} \to \sigma A_{rs} \tau$ if there is a stack symbol γ
    such that $\delta(p, \sigma, \varepsilon)$ contains (r, γ)
    and $\delta(s, \tau, \gamma)$ contains (q, ε)
  - For every state p, $A_{pp} \to \varepsilon$
- Start variable: $A_{q_{start} q_{accept}}$

Visualizing the Construction

How to generate all possible strings that could be recognized moving from state p with an empty stack to q with an empty stack? Two cases:





1. If the stack is also empty in some middle state r,
   trace the path from $p \to r$ then $r \to q$
2. Else if $p \to r$ pushes γ on the stack and $s \to q$ pops it back off,
   generate $\sigma A_{rs} \tau$.

Proof Sketch: the Grammar is Equivalent to the PDA

Claim: $A_{pq} \overset{*}{\Rightarrow} w$ if and only if w can take M from p to q, beginning & ending w/empty stack

$\Rightarrow$ Proof by induction on length of derivation

$\Leftarrow$ Proof by induction on length of computation

- Computation of length 0 (base case): Use $A_{pp} \rightarrow \varepsilon$
- Stack empties sometime in middle of computation:
  Use $A_{pq} \rightarrow A_{pr} A_{rq}$
- Stack does not empty in middle of computation:
  Use $A_{pq} \rightarrow \sigma A_{rs} \tau$

Context-free Grammars

STOP: End cgl.

Context-free Grammars

Reading: Sipser §2.1 (except Chomsky Normal Form).

Context-free Grammars

- Originated as abstract model for:
  - Structure of natural languages (Chomsky)
  - Syntactic specification of programming languages (Backus-Naur Form)

Context-free Grammars

- Originated as abstract model for:
  - Structure of natural languages (Chomsky)
  - Syntactic specification of programming languages (Backus-Naur Form)
- A context-free grammar is a set of **generative rules** for strings

  e.g.

$$G \quad = \quad \begin{array}{l} S \to aSb \\ S \to \varepsilon \end{array}$$

- A **derivation** looks like:

$$S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow aabb$$

$$L(G) = \{\varepsilon, ab, aabb, \ldots\} = \{a^n b^n : n \geq 0\}$$

Equivalent Formalisms

1. Backus-Naur Form (aka BNF, Backus Normal Form)

   due to John Backus and Peter Naur

   $$\begin{array}{lll}\langle\text{term}\rangle ::= \langle\text{factor}\rangle & \mid & \langle\text{factor}\rangle \text{ * } \langle\text{term}\rangle \\ & \mid & \langle\text{factor}\rangle \text{ / } \langle\text{term}\rangle\end{array}$$

   "|" means "or" in the metalanguage = same left-hand side

Equivalent Formalisms

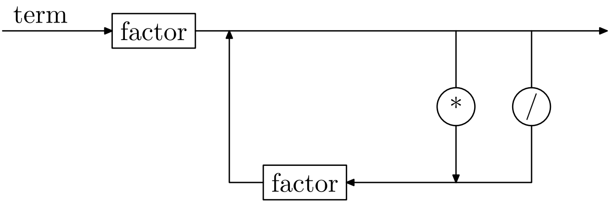
1. Backus-Naur Form (aka BNF, Backus Normal Form)

   due to John Backus and Peter Naur

   $$\begin{array}{rcl} \langle\text{term}\rangle ::= \langle\text{factor}\rangle & \mid & \langle\text{factor}\rangle \text{ * } \langle\text{term}\rangle \\ & \mid & \langle\text{factor}\rangle \text{ / } \langle\text{term}\rangle \end{array}$$

   "|" means "or" in the metalanguage = same left-hand side

2. "Railroad Diagrams"



Formal Definitions for CFGs

- A CFG $G = (V, \Sigma, R, S)$

  V = Finite set of **variables** (or **nonterminals**)

  Σ = The alphabet, a finite set of **terminals** ($V \cap \Sigma = \emptyset$).

  R = A finite set of **rules**, each of the form $A \to w$
  for $A \in V$ and $w \in (V \cup \Sigma)^*$.

  S = The **start variable**, a member of V

  e.g. $(\{S\}, \{a, b\}, \{S \to aSb, S \to \varepsilon\}, S)$

Formal Definitions for CFGs

- A CFG $G = (V, \Sigma, R, S)$

  V = Finite set of **variables** (or **nonterminals**)

  Σ = The alphabet, a finite set of **terminals** ($V \cap \Sigma = \emptyset$).

  R = A finite set of **rules**, each of the form $A \to w$
  for $A \in V$ and $w \in (V \cup \Sigma)^*$.

  S = The **start variable**, a member of V

  e.g. $(\{S\}, \{a, b\}, \{S \to aSb, S \to \varepsilon\}, S)$

- **Derivations:** For $\alpha, \beta \in (V \cup \Sigma)^*$ (strings of terminals and nonterminals),

  $\alpha \Rightarrow_G \beta$ if $\alpha = uAv, \beta = uwv$ for some $u, v \in (V \cup \Sigma)^*$ and rule $A \to w$.

  $\alpha \overset{*}{\Rightarrow}_G \beta$ ("α **yields** β") if there is a sequence $\alpha_0, \ldots, \alpha_k$ for $k \geq 0$ such that
  $\alpha_0 = \alpha$, $\alpha_k = \beta$, and $\alpha_{i-1} \Rightarrow_G \alpha_i$ for each $i = 1, \ldots, k$.

- $L(G) = \{w \in \Sigma^* : S \overset{*}{\Rightarrow}_G w\}$ (strings of terminals only!)

More examples of CFGs

- Arithmetic Expressions

G_1:

$$E \to x \mid y \mid E * E \mid E + E \mid (E)$$

G_2:

$$\begin{aligned} E &\to T \mid E + T \\ T &\to T * F \mid F \\ F &\to (E) \mid x \mid y \end{aligned}$$

Q: Which is preferable? Why?

More examples of CFGs

- $L = \{x \in \{(,)\}^*$: parentheses in x are properly 'balanced'$\}$.

- $L = \{x \in \{a, b\}^* : x$ has the same # of a's and b's$\}$.

Parse Trees

Derivations in a CFG can be represented by parse trees.

Examples:

Each parse tree corresponds to many derivations, but has unique **leftmost derivation**.

Parsing

Parsing: Given $x \in L(G)$, produce a parse tree for x. (Used to 'interpret' x. Compilers parse, rather than merely recognize, so they can assign semantics to expressions in the source language.)

Ambiguity: A grammar is **ambiguous** if some string has two parse trees.

Example:

Context-free Grammars and Automata

What is the fourth term in the analogy:

Regular Languages : Finite Automata
as
Context-free Languages : ???

Regular Grammars

Hint: There is a special kind of CFGs, the **regular grammars**, that generate exactly the regular languages.

A CFG is **(right-)regular** if any occurrence of a nonterminal on the RHS of a rule is as the rightmost symbol.

Turning a DFA into an equivalent Regular Grammar

- Variables are states.
- Transition $\delta(P, \sigma) = R$ $\quad P \xrightarrow{\sigma} R$
  becomes $P \to \sigma R$
- If P is accepting, add rule $P \to \varepsilon$
  Example: $\{x : x$ has an even # of a's and an even # of b's$\}$

Other Direction: Omitted.

CFL Closure Properties and Non–Context-Free Languages

Reading: Sipser §2.3.

Closure Properties of CFLs

- **Thm:** The CFLs are closed under
  - Union
  - Concatenation
  - Kleene *
  - Intersection with a regular language

Intersection of a CFL and a regular language is CF

Pf sketch: Let L_1 be CF and L_2 be regular

$L_1 = L(M_1)$, M_1 a PDA

$L_2 = L(M_2)$, M_2 a DFA

$Q_1 =$ state set of M_1

$Q_2 =$ state set of M_2

Construct a PDA with state set $Q_1 \times Q_2$ which keeps track of computation of both M_1 and M_2 on input.

Q: Why doesn't this argument work if M_1 and M_2 are both PDAs?

In fact, the intersection of two CFLs is not necessarily CF.

And the complement of a CFL is not necessarily CF (Asst 5).

Q: How to prove that languages are not context free?

Pumping Lemma for CFLs

Lemma: If L is context-free, then there is a number p (the **pumping length**) such that any $s \in L$ of length at least p can be divided into $s = uvxyz$, where

1. $uv^ixy^iz \in L$ for every $i \geq 0$,
2. $v \neq \varepsilon$ or $y \neq \varepsilon$, and
3. $|vxy| \leq p$.

Using the Pumping Lemma to Prove a language non–context-free

$\{a^n b^n c^n : n \geq 0\}$ is not CF.

$aaaaaaaaaaaaaaaa$	$bbbbbbbbbbbbbbbb$	$cccccccccccccccc$

What are v, y?

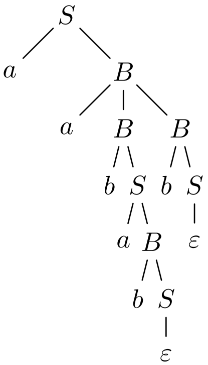
- Contain 2 kinds of symbols
- Contain only one kind of symbol

$\Rightarrow$ **Corollary:** CFLs not closed under intersection (why?)

Is the intersection of 2 CFLs or the complement of a CFL **sometimes** a CFL?

Recall: Parse Trees

$$\begin{array}{lclcl} S & \to & aB & \mid & bA \quad \mid \quad \varepsilon \\ A & \to & aS & \mid & bAA \\ B & \to & bS & \mid & aBB \end{array}$$

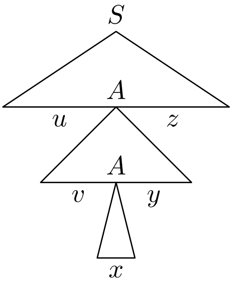


Parse tree for $aababb$, the "yield" of the tree

Height = max length path from S to a terminal symbol = 6 in above example

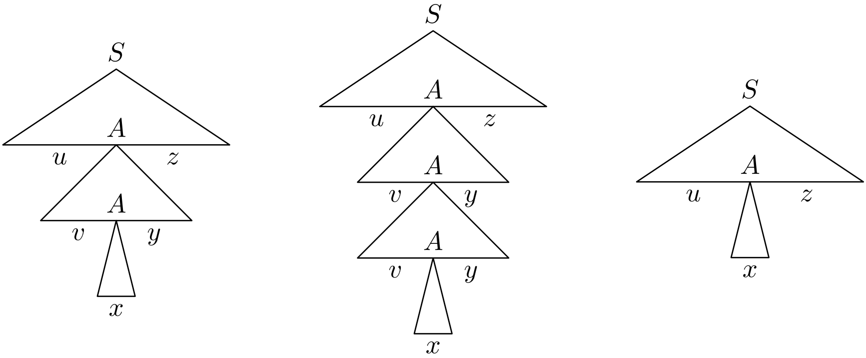
Proof of Pumping Lemma

Show that there exists a p such that any string s of length $\geq p$ has a parse tree of the form:



Proof of Pumping Lemma

Show that there exists a p such that any string s of length $\geq p$ has a parse tree of the form:



Finding “Repetition” in a big parse tree

- Since RHS of rules have bounded length, long strings must have tall parse trees
- A tall parse tree must have a path with a repeated nonterminal
- Let $p = b^m + 1$, where:

  $b =$ max length of RHS of a rule

  $m =$ # of variables
- Suppose T is the smallest parse tree for a string $s \in L$ of length at least p. Then

  Let $h =$ height of T. Then $b^h \geq p = b^m + 1$,

  $\Rightarrow h > m$,

  $\Rightarrow$ Path of length h in T has a repeated variable.

Final annoying details

- **Q:** Why is v or y nonempty?
- **Q:** How to ensure $|vxy| \leq p$?

Context-Free Recognition

Reading: Sipser §2.1 (Chomsky Normal Form).

Context-Free Recognition

- **Goal:** Given CFG G and string w to determine if $w \in L(G)$
- **First attempt:** Construct a PDA M from G and run M on w.
- **Brute-Force Method:**

  Check all parse trees of height up to some upper limit depending on G and $|w|$

  Exponentially costly
- **Better:**
  1. Transform G into Chomsky normal form (CNF) (once for G)
  2. Apply a special algorithm for CNF grammars

     (once for each w)

Chomsky Normal Form

Def: A grammar is in **Chomsky normal form** if

- the only possible rule with ε as the RHS is $S \to \varepsilon$
  (Of course, this rule occurs iff $\varepsilon \in L(G)$)
- Every other rule is of the form
  1. $X \to YZ$
     where X, Y, Z are variables
  2. $X \to \sigma$
     where X is a variable and σ is a single terminal symbol

Transforming a CFG into Chomsky Normal Form

Definitions:

- ε**-rule:** one of the form $X \to \varepsilon$
- **Long Rule:** one of the form $X \to \alpha$ where $|\alpha| > 2$
- **Unit Rule:** one of the form $X \to Y$
  where $X, Y \in V$
- **Terminal-Generating Rule:** one of the form $X \to \alpha$
  where $\alpha \notin V^*$ and $|\alpha| > 1$ (α has at least one terminal)

Eliminate non-Chomsky-Normal-Form Rules in Order:

1. All ε-rules, except maybe $S \to \varepsilon$
2. All unit rules
3. All long rules
4. All terminal-generating rules

Note: while eliminating rules of type j, we make sure not to reintroduce rules of type $i < j$.

Eliminating ε-Rules

0. Ensure start variable does not appear on the RHS of any rule (by adding new start variable with rule $S \to S_{old}$ if necessary).
1. To eliminate ε-rules, repeatedly do the following:
   a. Pick a ε-rule $Y \to \varepsilon$ and remove it.
   b. Given a rule $X \to \alpha$, where α contains n occurrences of Y, replace it with 2^n rules in which $0, \ldots, n$ occurrences are replaced by ε. (Do not add $X \to \varepsilon$ if previously removed.)

   e.g.

$$X \to aYZbY \qquad \Rightarrow$$

   (Why does this terminate?)

Eliminating ε-Rules

0. Ensure start variable does not appear on the RHS of any rule (by adding new start variable with rule $S \to S_{old}$ if necessary).
1. To eliminate ε-rules, repeatedly do the following:
   a. Pick a ε-rule $Y \to \varepsilon$ and remove it.
   b. Given a rule $X \to \alpha$, where α contains n occurrences of Y, replace it with 2^n rules in which $0, \ldots, n$ occurrences are replaced by ε. (Do not add $X \to \varepsilon$ if previously removed.)

   e.g.

$$X \to aYZbY \qquad \Rightarrow \qquad \begin{aligned} X &\to aYZbY \\ X &\to aZbY \\ X &\to aYZb \\ X &\to aZb \end{aligned}$$

(Why does this terminate?)

Eliminating Unit and Long Rules

2. To eliminate unit rules, repeatedly do the following:
   a. Pick a unit rule $A \to B$ and remove it.
   b. For every rule $B \to u$, add rule $A \to u$ unless this is a unit rule that was previously removed.
3. To eliminate long rules, repeatedly do the following:
   a. Remove a long rule $A \to u_1 u_2 \cdots u_k$, where each $u_i \in V \cup \Sigma$ and $k \geq 3$.
   b. Replace with rules $A \to u_1 A_1, A_1 \to u_2 A_2, \ldots, A_{k-2} \to u_{k-1} u_k$, where $A_1, \ldots, A_{k-2}$ are newly introduced variables used only in these rules.

Eliminating Terminal-Generating Rules

4. To eliminate terminal-generating rules:
   a. For each terminal a introduce a new nonterminal A.
   b. Add the rules $A \to a$
   c. "Capitalize" existing rules, e.g.
      replace $X \to aY$
      with $X \to AY$

Example of Transformation to Chomsky Normal Form

Starting grammar:

$$S \to XX$$
$$X \to aXb \mid \varepsilon$$

Benefit of CNF for Deciding if $w \in L(G)$

- **Observation:** If $S \Rightarrow XY \Rightarrow^* w$, then $w = uv$, $X \Rightarrow^* u$, $Y \Rightarrow^* v$ where u, v are *strictly shorter* than w.
- **Divide and Conquer:** can decide whether S yields w by recursively determining which variables yield substrings of w.
- **Dynamic Programming:** record answers to all subproblems to avoid repeating work.

Determining $w \in L(G)$, for G in CNF

Let $w = a_1 \cdots a_n$, $a_i \in \Sigma$.
Determine sets $S_{ij} (1 \leq i \leq j \leq n)$:

$$S_{ij} = \{X : X \stackrel{*}{\Rightarrow} a_i \cdots a_j, X \text{ variable of } G\}$$

	a_1					
S_{11}		a_2				
S_{12}	S_{22}		a_3			
S_{13}	S_{23}	S_{33}				
	S_{24}	S_{34}	S_{44}			
						a_n
S_{1n}					S_{nn}	

$w \in L(G)$ iff start symbol $\in S_{1n}$

Filling in the Matrix

- Calculate S_{ij} by induction on $j - i$
  - $(j - i = 0)$

    $$S_{ii} = \{X : X \to a_i \text{ is a rule of } G\}$$

  - $(j - i > 0)$

    $$\begin{aligned} X \in S_{ij} \text{ iff } &\exists \text{ rule } X \to YZ \\ &\exists k : i \leq k < j \\ \text{such that } &Y \in S_{ik} \\ &Z \in S_{k+1,j} \end{aligned}$$

e.g. $w = abaabb$

The Chomsky Normal Form Parsing Algorithm

for $i \leftarrow 1$ to n do

$S_{ii} = \{X : X \to a_i \text{ is a rule } \}$

for $d \leftarrow 1$ to $n-1$ do

for $i \leftarrow 1$ to $n-d$ do

$$S_{i,i+d} \leftarrow \bigcup_{j=i}^{i+d-1} \left\{ \begin{array}{l} X : X \to YZ \text{ is a rule,} \\ Y \in S_{ij}, Z \in S_{j+1,i+d} \end{array} \right\}$$

Complexity: $\mathcal{O}(n^3)$.

Of what does this triply nested loop remind you?

Of what does this triply nested loop remind you?

- Matrix Multiplication
- In fact, better matrix multiplication algorithms yield (asymptotically) better general context free parsing algorithms
- Strassen's algorithm requires $\mathcal{O}(n^{2.81})$ instead of $\mathcal{O}(n^3)$ multiplications

Summary of Context-Free Recognition

- CFL to PDA reduction yields nondeterministic automaton
- By use of Chomsky Normal Form and dynamic programming, there is a general $\mathcal{O}(n^3)$ non-stack-based algorithm
- The deterministic CFLs are the languages recognizable by deterministic PDAs
- E.g. $\{wcw^R : w \in \{a, b\}^*\}$ is a deterministic CFL but $\{ww^R : w \in \{a, b\}^*\}$ (even palindromes) is not
- Methods used in compilers are deterministic stack-based algorithms, requiring that the source language be deterministic CF or a special type of deterministic CF (LR(k), etc.)

Beyond Context-Free

- A **Context-Sensitive Grammar** allows rules of the form $\alpha \to \beta$, where α and β are strings and $|\alpha| \leq |\beta|$, so long as α contains at least one nonterminal.
- The possibility of using rules such as $aB \to aDE$ makes the grammar "sensitive to context"
- Is there an algorithm for determining whether $w \in L(G)$ where G is a CSG?
- But the field moved, and now we also move, from syntactic structures to computational difficulty