UNIT - IV

Normal Forms for Context- Free Grammars

Eliminating useless symbols, Eliminating €-Productions. Chomsky Normal form Griebech Normal form.

Pumping Lemma for Context-Free Languages

Statement of pumping lemma, Applications Closure

Properties of Context-Free Languages

Closure properties of CFL’s, Decision Properties of CFL's

Turing Machines

Introduction to Turing Machine, Formal Description, Instantaneous description, The language of a Turing machine

Normal forms for Context-Free Grammars

- In linguistics and computer science, a context-free grammar (CFG) is a formal grammar in which every production rule is of the form

  V → w

  where V is a "non-terminal symbol" and w is a "string" consisting of terminals and/or non-terminals.

- The term "context-free" expresses the fact that the non-terminal V can always be replaced by w, regardless of the context in which it occurs.

- A formal language is context-free if there is a context-free grammar that generates it.

- Context-free grammars are powerful enough to describe the syntax of most programming languages; in fact, the syntax of most programming languages is specified using context-free grammars.

- On the other hand, context-free grammars are simple enough to allow the construction of efficient parsing algorithms which, for a given string, determine whether and how it can be generated from the grammar.

- Not all formal languages are context-free.

- A well-known counter example is $\{ a^n b^n c^n : n >= 0 \}$ the set of strings containing some number of a's, followed by the same number of b's and the same number of c's.

- Just as any formal grammar, a context-free grammar G can be defined as a 4-tuple:

- $G = (V_t, V_n, P, S)$ where

- V_t is a finite set of terminals

- V_n is a finite set of non-terminals

- P is a finite set of production rules

- S is an element of V_n, the distinguished starting non-terminal.

- elements of *P* are of the form

$$V_n \rightarrow (V_t \cup V_n)^*$$

- A language L is said to be a Context-Free-Language (CFL) if its grammar is Context-Free. More precisely, it is a language whose words, sentences and phrases are made of symbols and words from a Context-Free-Grammar.
- Usually, CFL is of the form L=L(G).

Example 1

- A simple context-free grammar is given as:

$$S \rightarrow a\,S\,b \mid \varepsilon$$

- where | is used to separate multiple options for the same non-terminal, and ε stands for the empty string. This grammar generates the language $\{ a^n b^n : n >= 0 \}$, which is not regular.

Regular languages

- A **regular language** is a formal language (i.e., a possibly infinite set of finite sequences of symbols from a finite alphabet) that satisfies the following equivalent properties:
  - it can be accepted by a deterministic finite state machine
  - it can be accepted by a nondeterministic finite state machine
  - it can be accepted by an alternating finite automaton
  - it can be described by a regular expression it can be generated by a regular grammar
  - it can be generated by a prefix grammar

Regular languages

The collection of regular languages over an alphabet Σ is defined recursively as follows:

- the empty language Ø is a regular language. the empty string language { ε } is a regular language.
- For each *a* ϵ Σ, the singleton language { *a* } is a regular language.
- If *A* and *B* are regular languages, then *A* ∩ *B* (union), *A* ○ *B* (concatenation), and *A** (Kleene star) are regular languages.
- No other languages over Σ are regular.

Finite languages

Finite languages are:

A specific subset within the class of regular languages is the finite languages - those containing only a finite number of words. These are obviously regular as one can create a regular expression that is the union of every word in the language, and thus are regular.

Example 2

- A context-free grammar for the language consisting of all strings over {a,b} which contain a different number of a's to b's is
  - $S \rightarrow U \mid V$
  - $U \rightarrow TaU \mid TaT$
  - $V \rightarrow TbV \mid TbT$
  - $T \rightarrow aTbT \mid bTaT \mid \varepsilon$

- Here, T can generate all strings with the same number of a's as b's, U generates all strings with more a's than b's and V generates all strings with fewer a's than b's.

Example 3

Another example of a context-free language is $\{b^n a^m b^{2n} : n \geq 0, m \geq 0\}$
This is not a regular language, but it is context free as it can be generated by the following context-free grammar:

- $S \rightarrow b\ S\ bb \mid A$
- $A \rightarrow a\ A \mid \varepsilon$

Normal forms

- Every context-free grammar that does not generate the empty string can be transformed into an equivalent one in Chomsky normal form or Greibach normal form. "Equivalent" here means that the two grammars generate the same language.

- Because of the especially simple form of production rules in Chomsky Normal Form grammars, this normal form has both theoretical and practical implications.

- For instance, given a context-free grammar, one can use the Chomsky Normal Form to construct a polynomial-time algorithm which decides whether a given string is in the language represented by that grammar or not (the CYK algorithm).

Properties of context-free languages

- An alternative and equivalent definition of contextfree languages employs non-deterministic pushdown automata: a language is context-free if and only if it can be accepted by such an automaton.
- A language can also be modeled as a set of all sequences of terminals which are accepted by the grammar. This model is helpful in understanding set operations on languages.
- The union and concatenation of two context-free languages is context-free, but the intersection need not be.
- The reverse of a context-free language is contextfree, but the complement need not be.

Properties of context-free languages

- Every regular language is context-free because it can be described by a regular grammar.
- The intersection of a context-free language and a regular language is always context-free.
- There exist context-sensitive languages which are not context-free.
- To prove that a given language is not context-free, one may employ the pumping lemma for contextfree languages.
- The problem of determining if a context-sensitive grammar describes a context-free language is undecidable.

Normal forms for Context-Free Grammars

The goal is to show that every CFL (without ε) is generated by a CFG in which all productions are of the form A □□BC or A □□a, where A, B, C are variables, and a is a terminal.

Normal forms for Context-Free Grammars

□□A number of simplifications is inevitable:

1. The elimination of useless symbols, “variables or terminals that do not appear in any derivation of a terminal string from the start symbol”.
2. The elimination of ε-productions, those of the form A □□ε for some variable A.
3. The elimination of unit productions, those of the form A □□B for variables A and B.

Eliminating useless symbols

A symbol X is useful for Grammar G = {V, T, P, S}, if there is some derivation of the form S ═>* a X b ═>* w , where w ϵ T*

- X ϵ V or X ϵ T
- The sentential form of a X b might be the first or last derivation
- If X is not useful, then X is useless
- Characteristics of useful symbols (for instance X):

1. X is generating if X ═>* w for some terminal string w. Every terminal is generating since w can be that terminal itself, which is derived by 0 steps.
2. X is reachable if there is a derivation S ═>* a X b for some a and b.
   A symbol which is useful is surely to be both generating and reachable.

Eliminating the symbols which are not generating first followed by eliminating the symbols which are not reachable from the remaining grammar, this will generate a grammar consisting of only useful symbols.

- If we have the following grammar:

$$\begin{aligned} S &\to AB \mid a \\ A &\to b \end{aligned}$$

- Notice that a and b generate themselves "terminals", S generates a, and A generates b. B is not generating.

Eliminating ε-productions

- After eliminating B:

$$\begin{aligned} S &\to a \\ A &\to b \end{aligned}$$

- Notice that only S and a are reachable after eliminating the non-generating B.
- A is not reachable; so it should be eliminated. □□The result :

$$S \to a$$

- This production itself is a grammar that has the same result, which is {a}, as the original grammar.

Computing the generating and reachable symbols

- Basis: Every Symbol of T is obviously generating; it generates itself.
- Induction: If we have a production A → a, and every symbol of a is already known to be generating, then A is generating; because it generates all and only generating symbols, even if a = ε ; since all variables that have ε as a production body are generating.
- Theorem: The previous algorithm finds all and only the Generating symbols of G

Computing the generating and reachable symbols

- Basis : For a grammar G = {V, T, P, S} S is surely reachable.
- Induction: If we discovered that some variable A is reachable, then for all productions with A in the head (first part of the expression), all the symbols of the bodies (second part of the expression) of those productions are also reachable.
- Theorem: The above algorithm finds all and only the Reachable symbols of G

Eliminating useless symbols

- So far, the first step, which is the elimination of useless symbols is concluded.

- Now, for the second part, which is the elimination of ε-productions.
- The strategy is to have the following:

  if L is CFG, then L – {ε} is also CFG
- This is done through discovering the nullable variables. A variable for instance A, is nullable if: A =>* ε .
- Whenever A appears in a production body, A might or might not derive ε
- Basis: If A □□ε is a production of G, then A is nullable
- Induction: If there is a production

  B □□$C_1 C_2 \ldots C_k$ such that each C is a variable and each C is nullable, then B is nullable
- Theorem: For any grammar G, the only nullable symbols are the variables that derive ε in previous algorithm □□Proof:

  <u>for one step</u> : A □□ε must be a production, then this implies that A is discovered as nullable (as in basis).

  <u>for N > 1 steps:</u> the first step is A □□$C_1 C_2 \ldots C_k$ □□ε , each C_i derives ε by a sequence < N steps.

  By the induction, each C_i is discovered by the algorithm to be nullable. So by the inductive step, A is eventually found to be nullable.
- If a grammar G_1 is constructed by the elimination of ε-productions " using the previous method " of grammar G, then

$$\mathbf{L(G_1) = L(G) - \{\varepsilon\}}$$

Eliminating unit productions

- The last part concerns the eliminating of unit productions
- Any production of the form A □□B , where A and B are variables, is called a unit production.
- These production introduce extra steps in the derivations that obviously are not needed in there.
- Basis: (A, A) is a unit pair of any variable A, if A =>* A by 0 steps.
- Induction: Let's (A, B) be a unit pair, and let B □□C is a production, where A, B, and C are variables, then we can conclude that (A, C) is also a unit pair.

- Theorem: The previous algorithm (basis and induction) finds exactly all the unit pairs for any grammar G.

Example 7.12

$$\begin{array}{lcl} I & \rightarrow & a \mid b \mid Ia \mid Ib \mid I0 \mid I1 \\ F & \rightarrow & I \mid (E) \\ T & \rightarrow & F \mid T * F \\ E & \rightarrow & T \mid E + T \end{array}$$

Pair	Productions
(E,E)	$E \rightarrow E + T$
(E,T)	$E \rightarrow T * F$
(E,F)	$E \rightarrow (E)$
(E,I)	$E \rightarrow a \mid b \mid Ia \mid Ib \mid I0 \mid I1$
(T,T)	$T \rightarrow T * F$
(T,F)	$T \rightarrow (E)$
(T,I)	$T \rightarrow a \mid b \mid Ia \mid Ib \mid I0 \mid I1$
(F,F)	$F \rightarrow (E)$
(F,I)	$F \rightarrow a \mid b \mid Ia \mid Ib \mid I0 \mid I1$
(I,I)	$I \rightarrow a \mid b \mid Ia \mid Ib \mid I0 \mid I1$

After eliminating the unit productions, the generated grammar is:

$$\begin{array}{l} E \rightarrow E + T \mid T * F \mid (E) \mid a \mid b \mid Ia \mid Ib \mid I0 \mid I1 \\ T \rightarrow T * F \mid (E) \mid a \mid b \mid Ia \mid Ib \mid I0 \mid I1 \\ F \rightarrow (E) \mid a \mid b \mid Ia \mid Ib \mid I0 \mid I1 \\ I \rightarrow a \mid b \mid Ia \mid Ib \mid I0 \mid I1 \end{array}$$

This grammar has no unit productions and still generates the same expressions as the previous one.

Chomsky Normal Form

Conclusion of all three elimination stages:

Theorem: If G is a CFG which generates a language that consists of at least one string along with ε, then there is another CFG G_1 such that:

L{G_1} = L{G} – {ε} , "no ε-productions", and G_1 has neither unit productions nor useless symbols

- Proof: Start by performing the elimination of ε-productions. Then perform the elimination of unit productions, so the resulting grammar won't introduce any ε-productions since the new bodies are still identical to some bodies of the old grammar. Finally, perform the elimination of useless symbols, and since this eliminates productions and symbols, it will never reintroduce any ε-productions nor unit productions

Every nonempty CFL without ε has grammar G in which all productions are in one of the following forms:

- A □□BC , where A, B, and C are variables or
- A □□a , where A is a variable and a is a terminal

- Also G doesn't contain any useless symbols

- A grammar complying to these forms is called a Chomsky Normal Form (CNF).
- The construction of CNF is performed through:

1. Arrangement of all bodies of length 2 or more to contain only variables.

2. Breaking bodies of length 3 or more into a cascade productions, where each one has a body consisting of 2 variables.

Example 7.15

$$\begin{aligned}
I &\rightarrow a \mid b \mid Ia \mid Ib \mid I0 \mid I1\\
F &\rightarrow I \mid (E)\\
T &\rightarrow F \mid T * F\\
E &\rightarrow T \mid E + T
\end{aligned}$$

- First: we introduce new variables to represent terminals:

$$\begin{array}{llll}
A \rightarrow a & B \rightarrow b & Z \rightarrow 0 & O \rightarrow 1\\
P \rightarrow + & M \rightarrow * & L \rightarrow (& R \rightarrow)
\end{array}$$

- Second: We make all bodies either a single terminal or multiple variables:

$$\begin{aligned}
E &\rightarrow EPT \mid TMF \mid LER \mid a \mid b \mid IA \mid IB \mid IZ \mid IO\\
T &\rightarrow TMF \mid LER \mid a \mid b \mid IA \mid IB \mid IZ \mid IO\\
F &\rightarrow LER \mid a \mid b \mid IA \mid IB \mid IZ \mid IO\\
I &\rightarrow a \mid b \mid IA \mid IB \mid IZ \mid IO\\
A &\rightarrow a\\
B &\rightarrow b\\
Z &\rightarrow 0\\
O &\rightarrow 1\\
P &\rightarrow +\\
M &\rightarrow *\\
L &\rightarrow (\\
R &\rightarrow)
\end{aligned}$$

- Last step: we make all bodies either a single terminal or two variables:

$$
\begin{array}{lcl}
E & \rightarrow & EC_1 \mid TC_2 \mid LC_3 \mid a \mid b \mid IA \mid IB \mid IZ \mid IO \\
T & \rightarrow & TC_2 \mid LC_3 \mid a \mid b \mid IA \mid IB \mid IZ \mid IO \\
F & \rightarrow & LC_3 \mid a \mid b \mid IA \mid IB \mid IZ \mid IO \\
I & \rightarrow & a \mid b \mid IA \mid IB \mid IZ \mid IO \\
A & \rightarrow & a \\
B & \rightarrow & b \\
Z & \rightarrow & 0 \\
O & \rightarrow & 1 \\
P & \rightarrow & + \\
M & \rightarrow & * \\
L & \rightarrow & (\\
R & \rightarrow &) \\
C_1 & \rightarrow & PT \\
C_2 & \rightarrow & MF \\
C_3 & \rightarrow & ER
\end{array}
$$

PROPERTIES OF CONTEXT-FREE LANGUAGES

Decision Properties
Closure Properties

Summary of Decision Properties

- As usual, when we talk about “a CFL” we really mean “a representation for the CFL, e.g., a CFG or a PDA accepting by final state or empty stack.
- There are algorithms to decide if:

1. String w is in CFL L.
2. CFL L is empty.
3. CFL L is infinite.

Decision Properties

- Many questions that can be decided for regular sets cannot be decided for CFL’s.
- Example: Are two CFL’s the same? Example: Are two CFL’s disjoint?
- How would you do that for regular languages?
- Need theory of Turing machines and decidability to prove no algorithm exists.

Testing Emptiness

- We already did this.
- We learned to eliminate variables that generate no terminal string.
- If the start symbol is one of these, then the CFL is empty; otherwise not.

Testing Membership

- Want to know if string w is in L(G).
- Assume G is in CNF.
- Or convert the given grammar to CNF.
- w = ε is a special case, solved the start symbol is nullable.
- Algorithm (*CYK*) is a good dynamic programming and (n^3), where n = |w|. by testing if example of runs in time O

CYK Algorithm

- Let w = $a_1 \ldots a_n$.
- We construct an n-by-n triangular array of sets of variables.
- X_{ij} = {variables A | A =>* $a_i \ldots a_j$}. Induction on j–i+1.
  - The length of the derived string.
- Finally, ask if S is in X_{1n}.

- Induction: X = {A | there is a production A -> BC and an integer k, with i ≤ k < j, such that B is in X and C is in $X_{k+1,j}$.

Example: CYK Algorithm

Grammar: S -> AB, A -> BC | a, B -> AC | b, C -> a | b String w = ababa

X_{12}={B,S} X_{23}={A} X_{34}={B,S} X_{45}={A}

X_{11}={A,C} X_{22}={B,C} X_{33}={A,C} X_{44}={B,C} X_{55}={A,C}

Example: CYK Algorithm

Grammar: S -> AB, A -> BC | a, B -> AC | b, C -> a | b String w = ababa

X_{13}={}

X_{12}={B,S}

X_{11}={A,C}

Yields

X_{23}={A}

X_{22}={B,C}

nothing

X_{34}={B,S}

X_{33}={A,C}

X_{45}={A}

X_{44}={B,C} X_{55}={A,C}

Example: CYK Algorithm
Grammar: S -> AB, A -> BC | a, B -> AC | b, C -> a | b String w = ababa

X_{13}={A}

X_{12}={B,S}

X_{11}={A,C}

X_{24}={B,S}

X_{23}={A}

X_{22}={B,C}

X_{35}={A}

X_{34}={B,S}

X_{33}={A,C}

X_{45}={A}

X_{44}={B,C} X_{55}={A,C}

Example: CYK Algorithm

Grammar: S -> AB, A -> BC | a, B -> AC | b, C -> a | b String w = ababa

X_{14}={B,S}
X_{13}={A}
X_{12}={B,S}
X_{11}={A,C}
X_{24}={B,S}
X_{23}={A}
X_{22}={B,C}
X_{35}={A}
X_{34}={B,S}
X_{33}={A,C}
X_{45}={A}

X_{44}={B,C} X_{55}={A,C}

Example: CYK Algorithm
Grammar:

X_{15}={A}
S -> AB, A -> BC | a, B -> AC | b, C -> a | b String w = ababa

X_{14}={B,S}
X_{13}={A}
X_{12}={B,S}
X_{11}={A,C}

X_{25}={A}
X_{24}={B,S}
X_{23}={A}
X_{22}={B,C}
X_{35}={A}
X_{34}={B,S}
X_{33}={A,C}

X_{45}={A}
X_{44}={B,C} X_{55}={A,C}

Testing Infiniteness

- The idea is essentially the same as for regular languages.
- Use the pumping lemma constant n.
- If there is a string in the language of length between n and 2n-1, then the language is infinite; otherwise not.
- Let’s work this out in class.

CLOSURE PROPERTIES OF CFL’S:

- CFL’s are closed under union, concatenation, and Kleene closure.
- Also, under and inverse
reversal, homomorphisms homomorphisms.
- But not under intersection or difference.

Closure of CFL’s Under Union

- Let L and M be CFL’s with grammars G and H, respectively.
- Assume G and H have no variables in common.
  - Names of variables do not affect the language.
- Let S and and H.
S_2 be the start symbols of G

Closure Under Union – (2)

- Form a new grammar for L M by combining all the symbols and productions of G and H.
- Then, add a new start symbol S. Add productions S -> S_1 | S_2.

Closure Under Union – (3)

In the new grammar, all derivations start with S.

The first step replaces S by either S_2.
S_1 or

In the first case, the result must be a string in L(G) = L, and in the second case a string in L(H) = M.

Closure of CFL's Under Concatenation

Let L and M be CFL's with grammars G and H, respectively.

Assume G and H have no variables in common.

Let S and and H. S_2 be the start symbols of G

Closure Under Concatenation – (2)

Form a new grammar for LM by starting with all symbols and productions of G and H.

Add a new start symbol S.

Add production S -> S_1S_2.

Every derivation from S results in a string in L followed by one in M.

Closure Under Star

Let L have grammar G, with start symbol S_1. Form a new grammar for L* by introducing to G a new start symbol S and the 1

A rightmost derivation from S generates a sequence of zero or more S's, each of which generates some string in L.

Closure of CFL's Under Reversal

If L is a CFL with grammar G, form a grammar for L^R by reversing the right side of every production.

Example: Let G have S -> 0S1 | 01.

The reversal of L(G) has grammar S -> 1S0 | 10.

Closure of CFL's Under Homomorphism

Let L be a CFL with grammar G.

Let h be a homomorphism on the terminal symbols of G.

Construct a grammar for h(L) by replacing each terminal symbol *a* by h(a).

Example: Closure Under Homomorphism

G has productions S -> 0S1 | 01. h is defined by h(0) = ab, h(1) = ε. h(L(G)) has the grammar with
productions S -> abS | ab.

Closure of CFL's Under Inverse Homomorphism

Here, grammars don't help us.

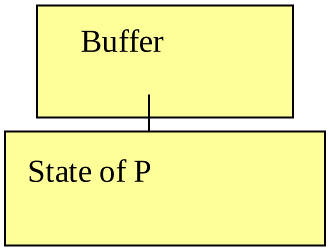
But a PDA construction serves nicely.

Intuition: Let L = L(P) for some PDA P. Construct PDA P' to accept h^{-1}(L).

P' simulates P, but keeps, as one component of a two-component state a buffer that holds the result of applying h to one input symbol.

Architecture of P'

Input: 0 0 1 1 h(0)



Read first remaining symbol in buffer as if it were input to P.

Stack of P

Formal Construction of P'

- States are pairs [q, b], where:
  1. q is a state of P.
  2. b is a suffix of h(a) for some symbol *a*.
     - Thus, only a finite number of possible values for b.
- Stack symbols of P' are those of P. Start state of P' is $[q_0, \varepsilon]$.

Construction of P' – (2)

Input symbols of P' are the symbols to which h applies.
Final states of P' are the states [q, ε] such that q is a final state of P.

Transitions of P'

1. δ'([q, ε], a, X) = {([q, any input symbol *a* of symbol X.
h(a)], X)} for
P' and any stack

1. When the buffer is empty, P’ can reload it.

2. δ’([q, bw], ε, X) contains ([p, w],) if δ(q, b, X) contains (p,), where b is either an input symbol of P or ε.

1. Simulate P from the buffer.

Proving Correctness of P’

We need to show that L(P’) = h^{-1}(L(P)). Key argument: P’ makes the transition

$(q_0, y, Z_0) \vdash^*_t (q, ε,)$, h(w) = yx, and x is a suffix of the last symbol of w.

Proof in both directions is an induction on the number of moves made.

Nonclosure Under Intersection

Unlike the regular languages, the class of CFL’s is not closed under .

We know not a CFL that L = {$0^n1^n2^n$ | n ≥ 1} is (use the pumping lemma).

However, L_2 = {$0^n1^n2^i$ | n ≥ 1, i ≥ 1} is. CFG: S -> AB, A -> 0A1 | 01, B -> 2B | 2.

So is L_3 = {$0^i1^n2^n$ | n ≥ 1, i ≥ 1}.

But L_1 = L_2 L_3.

Nonclosure Under Difference

We can prove something more general:

 Any class of languages that is closed under difference is closed under intersection.

Proof: L M = L – (L – M).

Thus, if CFL’s were closed under difference, they would be closed under intersection, but they are not.

Intersection with a Regular Language

Intersection of two CFL’s need not be context free.

But the intersection of a CFL with a regular language is always a CFL.

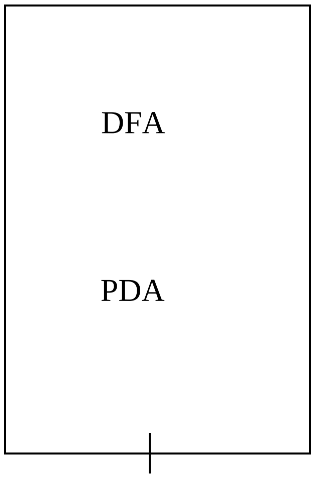
Proof involves running a DFA in parallel with a PDA, and noting that the combination is a PDA.

 PDA’s accept by final state.

DFA and PDA in Parallel

Input

Accept if both accept



state of one PDA

Stack

Formal Construction

Let the

Let the

DFA A have transition

PDA P have transition

function δ_A

. function δ_P.

States of combined PDA are [q,p], where q is a state of A and p a state of P.

δ([q,p], a, X) contains ([δ (q,a),r],) if δ_P(p, a, X) contains (r,).

 Note a could be , in which case δ_A(q,a) = q.

Formal Construction – (2)

Accepting states of combined PDA are those [q,p] such that q is an accepting state of A and p is an accepting state of P.

([q,p], ,) if and only if δ (q ,w) = q and in P: (p_0, w, Z_0)⊦*(p,