CS 3313
Foundations of Computing:

Simplification of Context Free Grammars

http://gw-cs3313-2021.github.io

Context Free Grammars

- A context free grammar is a grammar G=(V,T,P,S) where all production rules are of the form: $V \rightarrow (V \cup T)^*$
  - Production rules have exactly one variable on the left and a string consisting of variables and terminals on the right.

Recall

- Derivations: $\alpha A\beta \Rightarrow \alpha\gamma\beta$ if $A \rightarrow \gamma$ is a production, $\Rightarrow^*$
  - Leftmost derivation: leftmost variable replaced at each step
  - Rightmost derivation: rightmost variable replaced at each step
- Derivation or Parse Trees
  - S is root node, Variables are internal nodes, children are RHS of prod
  - Yield are leaves concatenated left to right
- Equivalence of Parse Trees and Derivations
  - Parse tree has an equivalent leftmost (/rightmost) derivation
- If G is a CFG, then L(G), the *language of G*, is

  $L(G) = \{w \mid S \Rightarrow^* w \text{ and } w \text{ is a string over set } T\}$.
- Ambiguity: A grammar is ambiguous if it has two distinct parse trees (leftmost/rightmost derivations)
  - Language is ambiguous if there is no unambiguous grammar for it

Next….the quest for ”automation”!

- We would like to answer the question “Does G derive string w” *i.e.,* Is w generated by the grammar?
  - Ex: Given the grammar of Python and a program in Python, does the program satisfy all the rules of the grammar.
- Design an algorithm that takes as input the grammar G and string w, and outputs the parse tree for w or returns “syntax error”
- How do we proceed ?
  - Grammars seem to be built “arbitrarily”
    - Algorithm should handle all possible representations
    - Complicates the algorithm

Simplification and Parsing

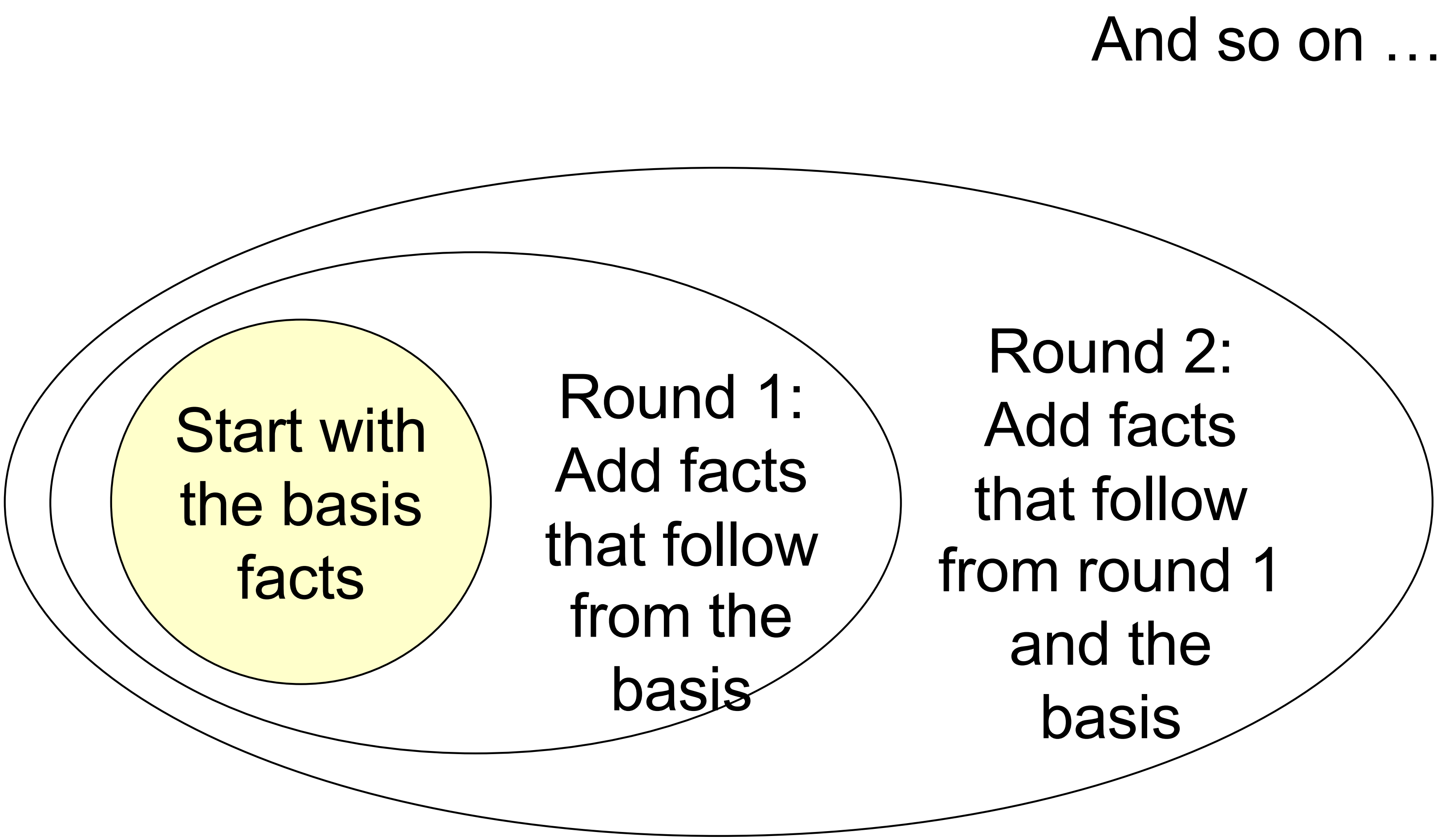
- 1. Simplification rules: transform a grammar such that:
  - Resulting grammar generates the same language
  - and has “more efficient” production rules in a specific format
- 2. Normal Forms: express all CFGs using a standard “format” for how the production rules are specified
  - Definition of CFGs places no restrictions on RHS of production
  - It is convenient (for parsing algorithms) to restrict to a standard form
    - Chomsky Normal Form (CNF) or Greiback Normal Form (GNF)
- 3. Parsing Algorithm: Design a parsing algorithm that takes a grammar in a standard form (CNF) to check if string w is generated by grammar G.

Comment: a useful algorithm design technique

- There is a family of algorithms that work *inductively*.
- They start discovering some facts that are obvious
  - the basis
- They discover more facts from what they already have discovered
  - induction
- Eventually, nothing more can be discovered, and we are done…..were called *discovery algorithms*

- Observation: decision algorithms for Reg. Lang as well as NFA to DFA (RE to NFA) used this process

Picture of Discovery



Simplification Rules: Why

- Exhaustive membership (i.e., parsing) algorithm:
  - Input string *w* of length *n*.
  - Starting with S, explore all productions for worst case *n* derivations and determine if it derives string *w* of length *n*.
  - How may steps for each of the n derivations:
    - Size of V (set of variables)
    - Size of P (set of productions)

- Observation: if we can remove variables and productions that do not play a part in deriving terminal strings, then we can improve run-time of the algorithm.

CFG Simplification: What is it ?

For a CFG G=(V,T,P,S)

- G has variables/productions that are "*useless*"
  - Variables that cannot derive a terminal string
    - Ex: $B \rightarrow AB$
  - G has variables that do not appear in any sentential form
    - Variables that cannot be "reached" from start S
- G has *λ– productions* but language does not contain λ
- We can have *unit productions* that create a chain of derivations without contributing at each step to a terminal derivation
  - Ex: $A \rightarrow B$. B. $\rightarrow C$. $C \rightarrow ab$

CFG Simplification Process

- Lemma 2.1: We derive an equivalent grammar by removing variables that do not derive a terminal string
- Lemma 2.2: We can derive an equivalent grammar by removing variables that do not appear in a sentential form
- Theorem 2.1: Combine Lemmas 2.1,2.2 to remove useless variables/productions
- Theorem 2.3: we can derive an equivalent grammar without λ– productions and get an equivalent grammar
- Theorem 2.2: we can derive an equivalent grammar without Unit Productions
- Note: We would like the proofs to result in procedures
  - using iterative algorithms

A Useful Substitution Rule

- Lemma 2.0: If A and B are distinct variables, a production of the form $A \rightarrow uBv$ can be replaced by a set of productions in which B is substituted by all strings B derives in one step.
- Consider the grammar

  $V = \{ A, B \}$, $T = \{ a, b, c \}$, and productions

  $A \rightarrow a \mid aaA \mid abBc \quad B \rightarrow abbA \mid b$

- We can replace $A \rightarrow abBc$ with two productions that replace B (in red), obtaining an equivalent grammar with productions

  $A \rightarrow a \mid aaA \mid ababbAc \mid abbc$

  $B \rightarrow abbA \mid b$

Useless Variables and Useless Productions

- A variable is *useful* if it occurs in the derivation of at least one string in the language
- A variable is *useless* if:
  - 1. No terminal strings can be derived from the variable
  - 2. The variable symbol cannot be reached from S
  - otherwise, the variable and any productions in which it appears is considered *useless*

- Ex: In the grammar below, (1) C does not derive a terminal string and (2) B can never be reached from the start symbol S

  $S \rightarrow A \mid AC$

  $A \rightarrow aA \mid \lambda$

  $B \rightarrow bA$

  $C \rightarrow AC$

Lemma 2.1: Removing variables that do not derive terminal strings

- Lemma 2.1: Given a CFG G=(V,T,P,S) we can find an equivalent grammar G_1 =(V', T, P', S) such that for each A in V', there is some w in T* such that A =>* w
  - Every variable in G_1 derives a terminal string
  - $L(G_1) = L(G)$

- We want to construct a proof that leads itself to a (discovery) algorithm
- Proof by induction.

Testing Whether a Variable Derives Some Terminal String: Proof

- The inductive proof serves as a *proof of correctness* for the algorithm
- Basis: If there is a production $A \rightarrow w$, where w has no variables, then A derives a terminal string.
- Induction: If there is a production $A \rightarrow \alpha$, where α consists only of terminals and variables known to derive a terminal string, then A derives a terminal string.
  - Eventually, we can find no more variables.

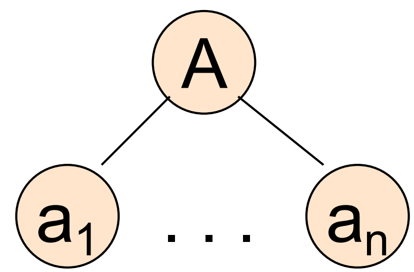
Proof

- An easy induction on the order in which variables are discovered shows that each one truly derives a terminal string.
- Conversely, any variable that derives a terminal string will be discovered by this algorithm.

Proof of Converse

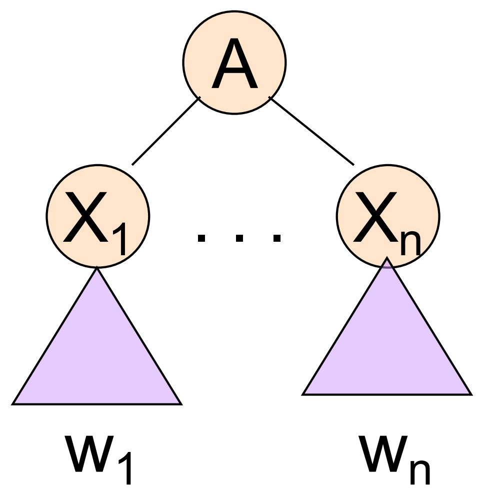
- The proof is an induction on the height of the least-height parse tree by which a variable A derives a terminal string.
- Basis: Height = 1. Tree looks like:
- Then the basis of the algorithm

  tells us that A will be discovered.



Induction for Converse

- Assume IH for parse trees of height < h, and suppose A derives a terminal string via a parse tree of height h:
- By IH, those X_i's that are
  variables are discovered.
- Thus, A will also be discovered, because it has a right side of terminals and/or discovered variables.



Algorithm to remove variables that do not derive terminal strings: Lemma 2.1

Input: $G = (V,T,P,S)$

1. $V_{init} := \emptyset$ /* initialize V_{init} to empty set
2. $V' = \{ A \mid A \rightarrow w$ is a production for w in $T^* \}$
3. While $V_{init} <> V'$
4. $V_{init} = V'$
5. $V' = V_{init} \cup \{A \mid A \rightarrow \alpha$ for some α in $(V_{init} \cup T)^* \}$
6. endwhile
7. $P' = \{ X \rightarrow \alpha \mid X \in V'$ and $\alpha \in (V' \cup T)^* \}$

Example: Lemma 2.1 Algorithm to Eliminate Variables that do not derive terminal strings

S -> AB | C, A -> aA | a, B -> bB, C -> c

- Basis: A and C are discovered because of A -> a and C -> c.
- Induction: S is discovered because of S -> C.
- Nothing else can be discovered.
- Result: S -> C, A -> aA | a, C -> c

Lemma 2.2: Removing variables or terminals that are not reachable from S

- Lemma 2.2: Given a CFG G=(V,T,P,S) we can find an equivalent grammar G_1 =(V', T', P', S) such that for each A in V' U T', there exist α, β in (V' U T')* for which S =>* αA β
  - Every variable or terminal in G_1 appears in a sentential form
  - i.e., is reachable from S through a series of derivations

- We want to construct a proof that leads itself to a (discovery) algorithm
  - A proof by induction on length of derivation or use reachability graph.

Algorithm to remove variables that are not reachable from S: Lemma 2.2

Input: G = (V,T,P,S); Output G'=(V',T',P',S) with all reachable

1. $V_{init} = \emptyset$
2. $V' = \{ S \}$
3. While V_{init} <> V' /* repeat loop until you cannot add more
4. $\quad V_{init} = V'$
5. $\quad V' = V_{init} \cup \{X \mid A \rightarrow \alpha,\ A \in V_{init}$ and X appears in $\alpha \}$
6. $\quad$ endwhile
7. $P' = \{ X \rightarrow \alpha \mid X \in V'$ and $\alpha \in (V' \cup T)^* \}$

Simpler approach:
Construct graph, with edge from X to Y where X is LHS of production and Y is on RHS of production
V' = all nodes reachable from S

Theorem 2.1: Removing useless symbols/productions

- Theorem 2.1: Every nonempty context free language is generated by a CFG G with no useless symbols.

- Proof:

1. Apply Lemma 2.1 and
2. then apply Lemma 2.2
3. Prove by contradiction.

Application of the Procedure for Removing Useless Productions

- Consider the grammar:

  $S \rightarrow aS \mid A \mid C$

  $A \rightarrow a$ $\quad B \rightarrow aa$ $\quad C \rightarrow aCb$

- In step 1 (Lemma 2.1), variables A, B, and S are added to V_1

since C is useless, it is eliminated in step 3, resulting in the grammar with productions

  $S \rightarrow aS \mid A$

  $A \rightarrow a$

  $B \rightarrow aa$

- In step 2 (Lemma 2.2), B is identified as unreachable from S, resulting in the grammar with productions

  $S \rightarrow aS \mid A$

  $A \rightarrow a$

λ-Productions

- A production with λ on the right side is called a *λ-production*
- A variable (symbol) A is called *nullable* if there is a sequence of derivations through which A produces λ
  - $A \Rightarrow^* \lambda$
- If a grammar generates a language not containing λ, any λ-productions can be removed
- Example: S_1 is nullable

  $S \rightarrow aS_1b$

  $S_1 \rightarrow aS_1b \mid \lambda$
- Since the language is λ-free, we have the equivalent grammar

  $S \rightarrow aS_1b \mid ab$

  $S_1 \rightarrow aS_1b \mid ab$

Example: Nullable Variables

S -> AB, A -> aA | λ, B -> bB | A

- Basis: A is nullable because of $A \rightarrow \lambda$
- Induction: B is nullable because of $B \rightarrow A$.
- Then, S is nullable because of $S \rightarrow AB$.

Theorem 2.2: Removing λ productions

- **Theorem 2.2**: If L = L(G) for some CFG G=(V,T,P,S) then L-{λ } is generated by a CFG G’ with no useless symbols or λ-productions.

1. First iteratively find *nullable* variables
2. Next replace RHS of production with nullable symbols replaced by λ
3. Then apply algorithms to remove useless symbols (Thm. 2.1)

- Key Idea: turn each production A -> $X_1 \ldots X_n$ into a set of productions
  - Except, if all X’s are nullable (or the body was empty to begin with), do not make a production with ε as the right side

Theorem 2.2: Proof

Formal proof by induction

Prove that for all variables A:

1. If $w \neq \lambda$ and $A \Rightarrow^*_{old} w$, then $A \Rightarrow^*_{new} w$.
2. If $A \Rightarrow^*_{new} w$ then $w \neq \lambda$ and $A \Rightarrow^*_{old} w$.

- Then, letting A be the start symbol proves that

  $L(new) = L(old) - \{\lambda\}$.

- (1) is an induction on the number of steps by which A derives *w* in the old grammar.

Proof – Basis

- If the old derivation is one step, then $A \rightarrow w$ must be a production.
- Since $w \neq \lambda$, this production also appears in the new grammar.
- Thus, $A \Rightarrow_{new} w$.

Proof: Inductive Step

- Let $A =>^*_{old} w$ be a k-step derivation, and assume the IH for derivations of fewer than k steps.
- Let the first step be $A =>_{old} X_1 \ldots X_n$.
- Then w can be broken into $w = w_1 \ldots w_n$, where $X_i =>^*_{old} w_i$, for all i, in fewer than k steps.

Induction – Continued

- By the IH, if $w_i \neq \lambda$, then $X_i \Rightarrow^*_{new} w_i$.
- Also, the new grammar has a production with A on the left, and just those X_i's on the right such that $w_i \neq \lambda$.
  - Note: they all can't be λ , because $w \neq \lambda$
- Follow a use of this production by the derivations

  $X_i \Rightarrow^*_{new} w_i$ to show that A derives *w* in the new grammar.

Theorem 2.2: Algorithm to remove λ productions

1. $V_N = \emptyset$ /* these are nullable variables
2. $V_{init} = \{ A \mid A \to \lambda \}$ /* add A if RHS of prod is λ
3. While $V_{init} \text{<>} V_N$ /* repeat loop to add A where $A \Rightarrow^*$
4. $V_{init} = V_N$
5. $V_N = V_{init} \cup \{A \mid A \to \alpha$ and α is in $V_{init}^* \}$
6. endwhile
7. Remove all λ productions from P
8. For each production in P, $A \to \alpha$,

   For each $X \in \alpha$ and $X \in V_N$ add productions in which nullable symbols are replaced by λ but not all are replaced by λ

Application of the Procedure for Removing λ-Productions

- Consider the grammar :

  $S \to ABaC$

  $A \to BC \qquad B \to b \mid \lambda$

  $C \to D \mid \lambda \qquad D \to d$

- In step 2, variables B, C are added to V_N
- In while loop, variable A is added to V_N
- In step 7, λ-productions are eliminated
- In step 8, productions are added by replacing nullable symbols with λ all possible combinations, resulting in

  $S \to ABaC \mid BaC \mid AaC \mid Aba \mid aC \mid Aa \mid Ba \mid a$

  $A \to B \mid C \mid BC$

  $B \to b \qquad C \to D \qquad D \to d$

Example: Eliminating λ-Productions

S -> ABC,   A -> aA | λ,   B -> bB | λ ,   C -> λ

- A, B, C, and S are all nullable.
- New grammar:

  S -> ABC | AB | AC | BC | A | B | C

  A -> aA | a

  B -> bB | b

Note: C is now useless.
Eliminate its productions.

Unit-Productions

- A production of the form A $\rightarrow$ B (where A and B are variables) is called a *unit-production*
- Unit-productions add unneeded complexity to a grammar and can usually be removed by simple substitution
- Theorem 2.2 states that any context-free grammar <u>without λ-productions</u> has an equivalent grammar without unit-productions
  - The procedure for eliminating unit-productions assumes that all λ-productions have been previously removed

Theorem 2.3: Removing Unit productions

- Theorem 2.3: Every context free language without the empty string is generated by a CFG G=(V,T,P,S) with no useless productions,λ productions, or unit productions.
- Proof:
  - First remove unit productions and then apply Theorem 2.2 and 2.1
- Key idea: If A =>* B by a series of unit productions, and B -> α is a non-unit-production, then add production A -> α.
  - Then, drop all unit productions.
- Formal proof by induction on length of derivation.

Algorithm for Removing Unit Productions

1. Draw a dependency graph with an edge from A to B corresponding to every $A \rightarrow B$ production in the grammar
2. Construct a new grammar that includes all the productions from the original grammar, except for the unit-productions
3. Whenever there is a path from A to B in the dependency graph, replace $A \rightarrow B$ with $A \rightarrow \alpha$ using the substitution rule from Lemma 2.0 (but using only the non-unit productions $B \rightarrow \alpha$ in the new grammar)

Application of the Procedure for Removing Unit-Productions

- Consider the grammar:

  $S \to Aa \mid B$

  $A \to a \mid bc \mid B$

  $B \to A \mid bb$

  The dependency graph contains paths from S to A, S to B, B to A, and A to B

- After removing unit-productions and adding the new productions (in red), the resulting grammar is

  $S \to Aa \mid a \mid bc \mid bb$

  $A \to a \mid bc \mid bb$

  $B \to a \mid bc \mid bb$

Proof The the Unit-Production-Elimination Algorithm Works

- Basic idea: there is a leftmost derivation $A \Rightarrow^*_{lm} w$ in the new grammar if and only if there is such a derivation in the old.
- A sequence of unit productions and a non-unit production is collapsed into a single production of the new grammar.

Theorem 2.3

1. Apply Theorem 2.2 (Algorithm to remove λ productions)
2. Apply Algorithm 2.3 to remove Unit Productions
3. Apply Theorem 2.1 (algorithms to remove useless symbols)
   - Observe that removing λ productions may introduce unit productions, hence the ordering.

- We can henceforth assume any CFG G can be “simplified” into an equivalent grammar G’ which has no useless symbols or productions, no λ-productions and no unit productions.

Putting it all together: Cleaning Up a Grammar

- **Theorem 2.4:** if L is a CFL, then there is a CFG for L – {λ} that has:
  1. No useless variables (and productions).
  2. No λ-productions.
  3. No unit productions.
- *Theorem 2.4 implies: every string on RHS of prodution is either a single terminal or has length $\geq$ 2.*

Cleaning Up CFGs

- Proof: Start with a CFG for L.
- Perform the following steps in order:

1. Eliminate λ-productions. (Theorem 2.3)
2. Eliminate unit productions. (Theorem 2.2)
3. Eliminate variables that derive no terminal string. (Lemma 2.1)
4. Eliminate variables not reached from the start symbol. (Lemma 2.2)

Must be first. This step can create unit productions or useless variables.

Next: Procedure to transform any CFG to Chomsky Normal Form

- A CFG is said to be in *Chomsky Normal Form* if every production is of one of these two forms:
  1. A -> BC (body is two variables).
  2. A -> a (body is a single terminal).
- Theorem: If L is a CFL, then L – {ϵ} has a CFG in CNF.
  - *Note: Theorem 2.4 implies every string on RHS of prodution is either a single terminal or has length $\geq$ 2.*
    - *This is our starting point when converting to CNF form*

- Question: property of parse trees for CNF grammars ?

Time to test out the algorithms: Inclass Exercises

- Given grammar G=(V,T,P,S), find an equivalent grammar G' with no unit productions, λ productions or useless variables/productions.
  - i.e, clean up the grammar