Context-Free Grammars

Formalism

Derivations

Backus-Naur Form

Left- and Rightmost Derivations

Informal Comments

- A *context-free grammar* is a notation for describing languages.
- It is more powerful than finite automata or RE's, but still cannot define all possible languages.
- Useful for nested structures, e.g., parentheses in programming languages.

Informal Comments – (2)

- Basic idea is to use "variables" to stand for sets of strings (i.e., languages).
- These variables are defined recursively, in terms of one another.
- Recursive rules ("productions") involve only concatenation.
- Alternative rules for a variable allow union.

Example: CFG for { 0^n1^n | $n \geq 1$}

- Productions:

  S -> 01

  S -> 0S1

- Basis: 01 is in the language.
- Induction: if w is in the language, then so is 0w1.

CFG Formalism

- *Terminals* = symbols of the alphabet of the language being defined.
- *Variables* = *nonterminals* = a finite set of other symbols, each of which represents a language.
- *Start symbol* = the variable whose language is the one being defined.

Productions

- A *production* has the form variable (*head*) -> string of variables and terminals (*body*).
- Convention:
  - A, B, C,… and also S are variables.
  - a, b, c,… are terminals.
  - …, X, Y, Z are either terminals or variables.
  - …, w, x, y, z are strings of terminals only.
  - α, β, γ,… are strings of terminals and/or variables.

Example: Formal CFG

- Here is a formal CFG for $\{ 0^n1^n \mid n \geq 1\}$.
- Terminals = {0, 1}.
- Variables = {S}.
- Start symbol = S.
- Productions =

  S -> 01

  S -> 0S1

Derivations – Intuition

- We *derive* strings in the language of a CFG by starting with the start symbol, and repeatedly replacing some variable A by the body of one of its productions.
  - That is, the "productions for A" are those that have head A.

Derivations – Formalism

- We say $\alpha A\beta$ => $\alpha\gamma\beta$ if A -> γ is a production.
- Example: S -> 01; S -> 0S1.
- S => 0S1 => 00S11 => 000111.

Iterated Derivation

- $=>^*$ means "zero or more derivation steps."
- Basis: $\alpha =>^* \alpha$ for any string α.
- Induction: if $\alpha =>^* \beta$ and $\beta => \gamma$, then $\alpha =>^* \gamma$.

Example: Iterated Derivation

- S -> 01; S -> 0S1.
- S => 0S1 => 00S11 => 000111.
- Thus S =>* S; S =>* 0S1;
  S =>* 00S11; S =>* 000111.

Sentential Forms

- Any string of variables and/or terminals derived from the start symbol is called a *sentential form*.
- Formally, α is a sentential form iff S =>* α.

Language of a Grammar

- If G is a CFG, then L(G), the *language of G*, is $\{w \mid S \Rightarrow^* w\}$.
- Example: G has productions S -> ε and S -> 0S1.
- $L(G) = \{0^n1^n \mid n \geq 0\}$.

Context-Free Languages

- A language that is defined by some CFG is called a *context-free language*.
- There are CFL's that are not regular languages, such as the example just given.
- But not all languages are CFL's.
- Intuitively: CFL's can count two things, not three.

BNF Notation

- Grammars for programming languages are often written in BNF (*Backus-Naur Form*).
- Variables are words in <...>; Example: <statement>.
- Terminals are often multicharacter strings indicated by boldface or underline; Example: **while** or <u>WHILE</u>.

BNF Notation – (2)

- Symbol ::= is often used for ->.
- Symbol | is used for "or."
  - A shorthand for a list of productions with the same left side.
- Example: S -> 0S1 | 01 is shorthand for S -> 0S1 and S -> 01.

BNF Notation – Kleene Closure

- Symbol … is used for “one or more.”
- Example: <digit> ::= 0|1|2|3|4|5|6|7|8|9
  <unsigned integer> ::= <digit>…
- Translation: Replace α… with a new variable A and productions A -> Aα | α.

Example: Kleene Closure

- Grammar for unsigned integers can be replaced by:

    U -> UD | D

    D -> 0|1|2|3|4|5|6|7|8|9

BNF Notation: Optional Elements

- Surround one or more symbols by [...] to make them optional.
- Example: <statement> ::= **if** <condition> **then** <statement> [; **else** <statement>]
- Translation: replace [α] by a new variable A with productions A -> α | ϵ.

Example: Optional Elements

- Grammar for if-then-else can be replaced by:

S -> iCtSA

A -> ;eS | ϵ

BNF Notation – Grouping

- Use {…} to surround a sequence of symbols that need to be treated as a unit.
  - Typically, they are followed by a … for "one or more."
- Example: <statement list> ::= <statement> [{;<statement>}…]

Translation: Grouping

- Create a new variable A for $\{\alpha\}$.
- One production for A: A -> α.
- Use A in place of $\{\alpha\}$.

Example: Grouping

L -> S [{;S}...]

- Replace by L -> S [A...]   A -> ;S
  - A stands for {;S}.
- Then by L -> SB   B -> A... | ϵ   A -> ;S
  - B stands for [A...] (zero or more A's).
- Finally by L -> SB   B -> C | ϵ
  C -> AC | A   A -> ;S
  - C stands for A... .

Leftmost and Rightmost Derivations

- Derivations allow us to replace any of the variables in a string.
  - Leads to many different derivations of the same string.
- By forcing the leftmost variable (or alternatively, the rightmost variable) to be replaced, we avoid these "distinctions without a difference."

Leftmost Derivations

- Say $wA\alpha \Rightarrow_{lm} w\beta\alpha$ if w is a string of terminals only and $A \rightarrow \beta$ is a production.
- Also, $\alpha \Rightarrow^*_{lm} \beta$ if α becomes β by a sequence of 0 or more $\Rightarrow_{lm}$ steps.

Example: Leftmost Derivations

- Balanced-parentheses grammmar:

  S -> SS | (S) | ()

- $S \Rightarrow_{lm} SS \Rightarrow_{lm} (S)S \Rightarrow_{lm} (())S \Rightarrow_{lm} (())()$
- Thus, $S \Rightarrow^*_{lm} (())()$
- S => SS => S() => (S)() => (())() is a derivation, but not a leftmost derivation.

Rightmost Derivations

- Say $\alpha Aw \Rightarrow_{rm} \alpha\beta w$ if w is a string of terminals only and A -> β is a production.
- Also, $\alpha \Rightarrow^*_{rm} \beta$ if α becomes β by a sequence of 0 or more $\Rightarrow_{rm}$ steps.

Example: Rightmost Derivations

- Balanced-parentheses grammmar:

  S -> SS | (S) | ()

- $S \Rightarrow_{rm} SS \Rightarrow_{rm} S() \Rightarrow_{rm} (S)() \Rightarrow_{rm} (())()$
- Thus, $S \Rightarrow^*_{rm} (())()$
- S => SS => SSS => S()S => ()()S => ()()() is neither a rightmost nor a leftmost derivation.

Parse Trees

Definitions

Relationship to Left- and Rightmost Derivations

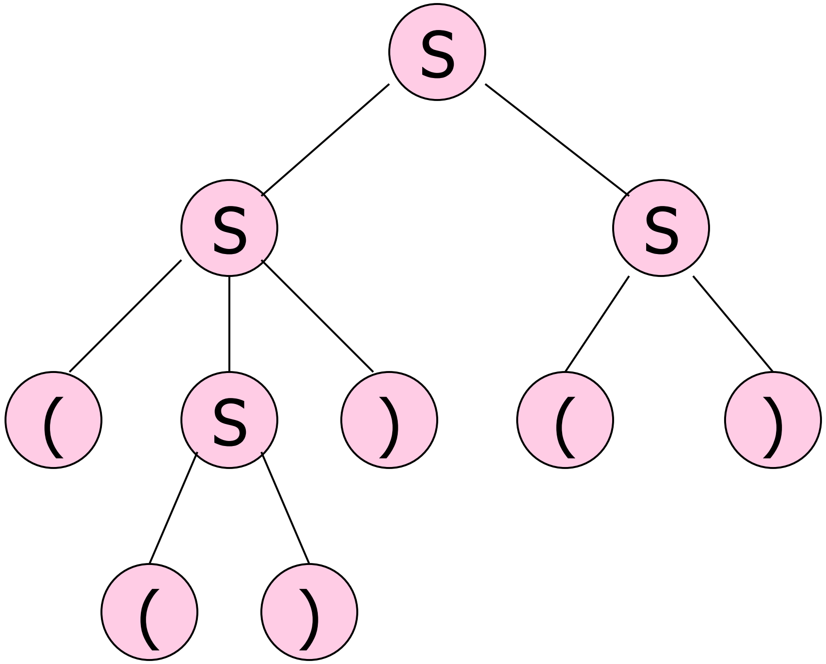
Ambiguity in Grammars

Parse Trees

- *Parse trees* are trees labeled by symbols of a particular CFG.
- Leaves: labeled by a terminal or ϵ.
- Interior nodes: labeled by a variable.
  - Children are labeled by the body of a production for the parent.
- Root: must be labeled by the start symbol.

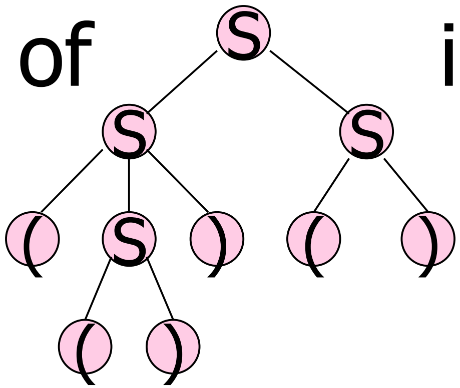
Example: Parse Tree

S -> SS | (S) | ()



Yield of a Parse Tree

- The concatenation of the labels of the leaves in left-to-right order
  - That is, in the order of a preorder traversal.

  is called the *yield* of the parse tree.
- Example: yield of 

 is (())()

Generalization of Parse Trees

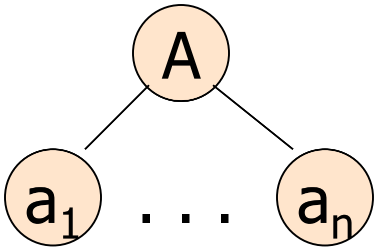
- We sometimes talk about trees that are not exactly parse trees, but only because the root is labeled by some variable A that is not the start symbol.
- Call these *parse trees with root A*.

Parse Trees, Leftmost and Rightmost Derivations

- Trees, leftmost, and rightmost derivations correspond.
- We'll prove:
  1. If there is a parse tree with root labeled A and yield w, then $A =>^*_{lm} w$.
  2. If $A =>^*_{lm} w$, then there is a parse tree with root A and yield w.

Proof – Part 1

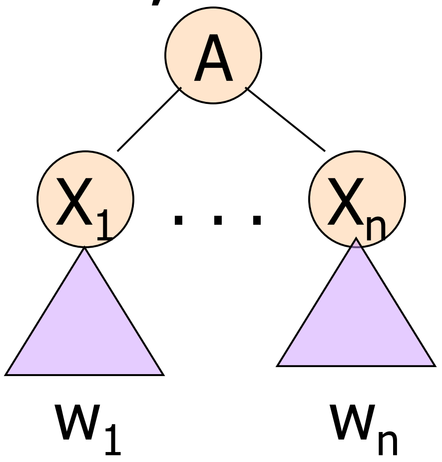
- Induction on the *height* (length of the longest path from the root) of the tree.
- Basis: height 1. Tree looks like



- $A \rightarrow a_1 \ldots a_n$ must be a production.
- Thus, $A \Rightarrow^*_{lm} a_1 \ldots a_n$.

Part 1 – Induction

- Assume (1) for trees of height < h, and let this tree have height h:
- By IH, $X_i \Rightarrow^*_{lm} w_i$.
  - Note: if X_i is a terminal, then $X_i = w_i$.
- Thus, $A \Rightarrow_{lm} X_1 \ldots X_n \Rightarrow^*_{lm} w_1 X_2 \ldots X_n \Rightarrow^*_{lm} w_1 w_2 X_3 \ldots X_n \Rightarrow^*_{lm} \ldots \Rightarrow^*_{lm} w_1 \ldots w_n$.

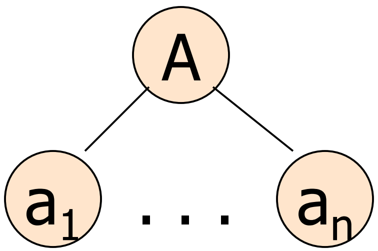


Proof: Part 2

- Given a leftmost derivation of a terminal string, we need to prove the existence of a parse tree.
- The proof is an induction on the length of the derivation.

Part 2 – Basis

- If $A \Rightarrow^*_{lm} a_1 \ldots a_n$ by a one-step derivation, then there must be a parse tree

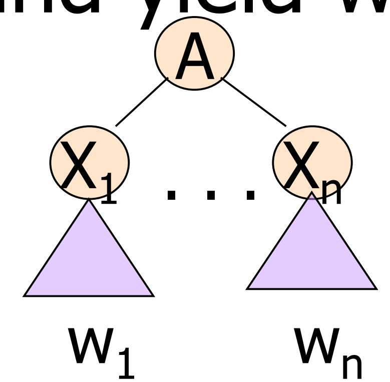


Part 2 – Induction

- Assume (2) for derivations of fewer than k > 1 steps, and let $A =>*_{lm} w$ be a k-step derivation.
- First step is $A =>_{lm} X_1...X_n$.
- Key point: w can be divided so the first portion is derived from X_1, the next is derived from X_2, and so on.
  - If X_i is a terminal, then $w_i = X_i$.

Induction – (2)

- That is, $X_i =>*_{lm} w_i$ for all i such that X_i is a variable.
  - And the derivation takes fewer than k steps.
- By the IH, if X_i is a variable, then there is a parse tree with root X_i and yield w_i.
- Thus, there is a parse tree



Parse Trees and Rightmost Derivations

- The ideas are essentially the mirror image of the proof for leftmost derivations.
- Left to the imagination.

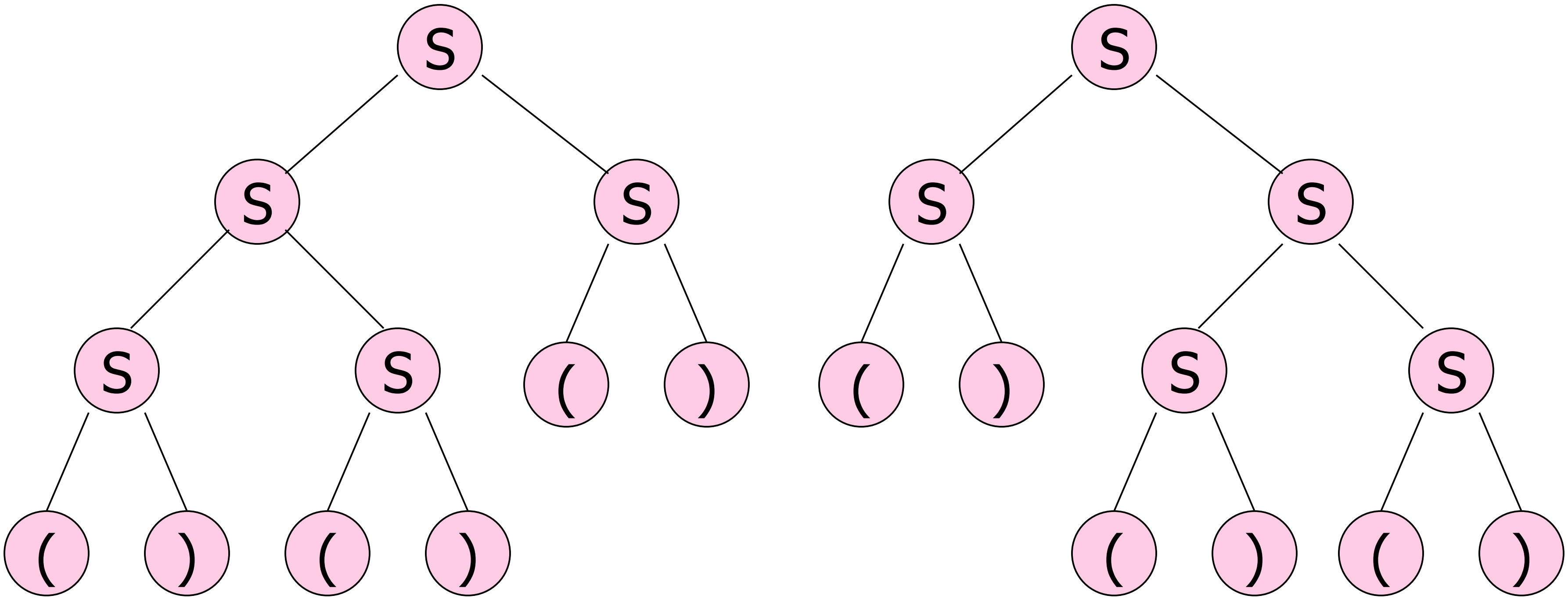
Parse Trees and Any Derivation

- The proof that you can obtain a parse tree from a leftmost derivation doesn't really depend on "leftmost."
- First step still has to be A => $X_1 \ldots X_n$.
- And w still can be divided so the first portion is derived from X_1, the next is derived from X_2, and so on.

Ambiguous Grammars

- A CFG is *ambiguous* if there is a string in the language that is the yield of two or more parse trees.
- Example: S -> SS | (S) | ()
- Two parse trees for ()()() on next slide.

Example – Continued



Ambiguity, Left- and Rightmost Derivations

- If there are two different parse trees, they must produce two different leftmost derivations by the construction given in the proof.
- Conversely, two different leftmost derivations produce different parse trees by the other part of the proof.
- Likewise for rightmost derivations.

Ambiguity, etc. – (2)

- Thus, equivalent definitions of "ambiguous grammar" are:
  1. There is a string in the language that has two different leftmost derivations.
  2. There is a string in the language that has two different rightmost derivations.

Ambiguity is a Property of Grammars, not Languages

- For the balanced-parentheses language, here is another CFG, which is unambiguous.

  B -> (RB | ε

  R ->) | (RR

B, the start symbol, derives balanced strings.

R generates certain strings that have one more right paren than left.

Example: Unambiguous Grammar

B -> (RB | ϵ   R ->) | (RR

- Construct a unique leftmost derivation for a given balanced string of parentheses by scanning the string from left to right.
  - If we need to expand B, then use B -> (RB if the next symbol is "("; use ϵ if at the end.
  - If we need to expand R, use R ->) if the next symbol is ")" and (RR if it is "(".

The Parsing Process

Remaining Input:

(())()

↑ Next symbol

Steps of leftmost derivation:

B

B -> (RB | ϵ   R ->) | (RR

The Parsing Process

Remaining Input:

())()

Next symbol

Steps of leftmost derivation:

B

(RB

B -> (RB | ε   R ->) | (RR

The Parsing Process

Remaining Input:

))()

↑ Next symbol

Steps of leftmost derivation:

B

(RB

((RRB

B -> (RB | ϵ   R ->) | (RR

The Parsing Process

Remaining Input:

)()

Next symbol

Steps of leftmost derivation:

B

(RB

((RRB

(()RB

B -> (RB | ϵ   R ->) | (RR

The Parsing Process

Remaining Input:

()

↑ Next symbol

Steps of leftmost derivation:

B

(RB

((RRB

(()RB

(())B

B -> (RB | ϵ R ->) | (RR

The Parsing Process

Remaining Input:

)

↑

Next symbol

Steps of leftmost derivation:

B

(RB

((RRB

(()RB

(())B

(())(RB

B -> (RB | ϵ   R ->) | (RR

The Parsing Process

Remaining Input:

↑

Next symbol

Steps of leftmost derivation:

B

(RB

((RRB

(()RB

(())B

(())(RB

(())()B

B -> (RB | ϵ   R ->) | (RR

The Parsing Process

Remaining Input:

Next symbol

Steps of leftmost derivation:

B

(RB

((RRB

(()RB

(())B

(())(RB

(())()B

(())()

B -> (RB | ϵ   R ->) | (RR

LL(1) Grammars

- As an aside, a grammar such B -> (RB | ϵ R ->) | (RR, where you can always figure out the production to use in a leftmost derivation by scanning the given string left-to-right and looking only at the next one symbol is called LL(1).
  - "Leftmost derivation, left-to-right scan, one symbol of lookahead."

LL(1) Grammars – (2)

- Most programming languages have LL(1) grammars.
- LL(1) grammars are never ambiguous.

Inherent Ambiguity

- It would be nice if for every ambiguous grammar, there were some way to "fix" the ambiguity, as we did for the balanced-parentheses grammar.
- Unfortunately, certain CFL's are *inherently ambiguous*, meaning that every grammar for the language is ambiguous.

Example: Inherent Ambiguity

- The language $\{0^i1^j2^k \mid i = j \text{ or } j = k\}$ is inherently ambiguous.
- Intuitively, at least some of the strings of the form $0^n1^n2^n$ must be generated by two different parse trees, one based on checking the 0's and 1's, the other based on checking the 1's and 2's.

One Possible Ambiguous Grammar

S -> AB | CD

A -> 0A1 | 01 — A generates equal 0's and 1's

B -> 2B | 2 — B generates any number of 2's

C -> 0C | 0 — C generates any number of 0's

D -> 1D2 | 12 — D generates equal 1's and 2's

And there are two derivations of every string with equal numbers of 0's, 1's, and 2's. E.g.:

S => AB => 01B =>012

S => CD => 0D => 012

Normal Forms for CFG's

Eliminating Useless Variables

Removing Epsilon

Removing Unit Productions

Chomsky Normal Form

Variables That Derive Nothing

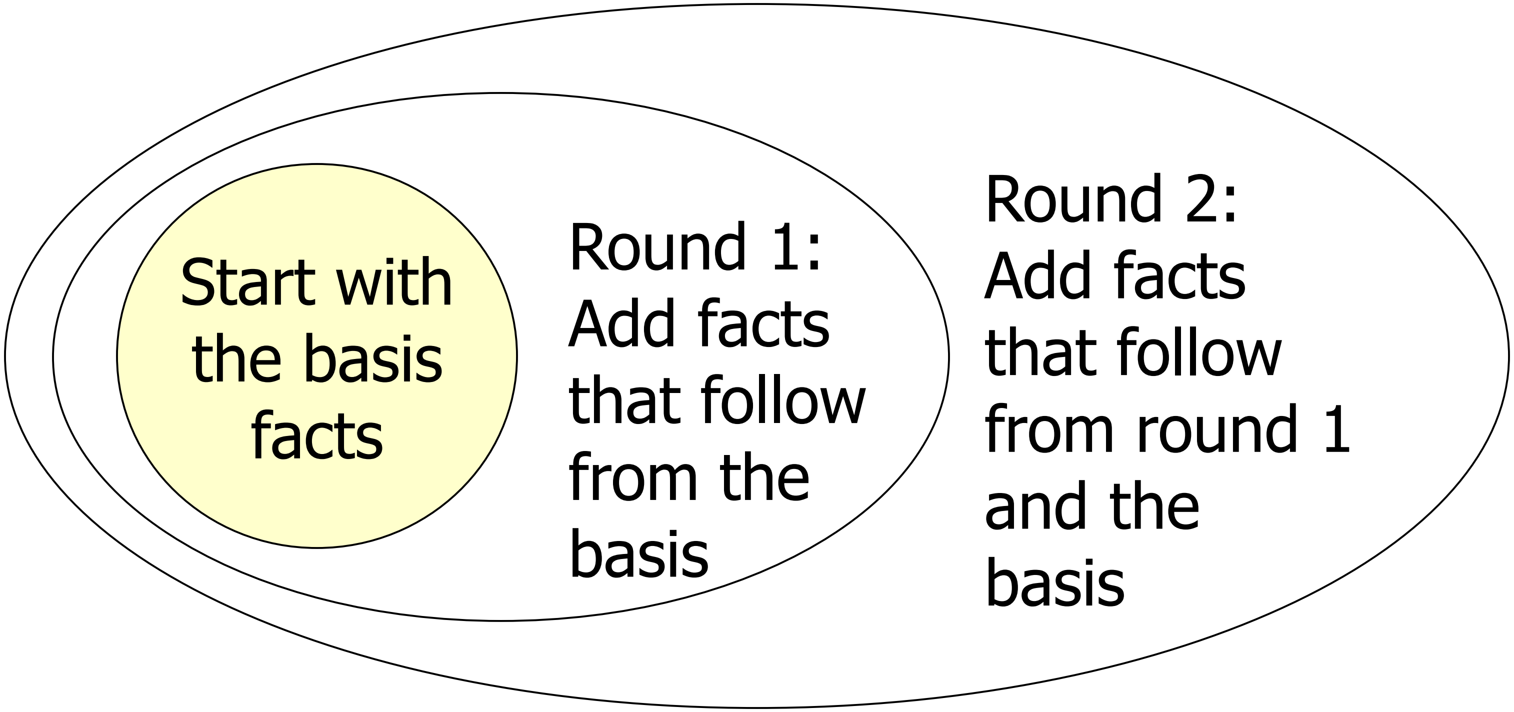
- Consider: S -> AB, A -> aA | a, B -> AB
- Although A derives all strings of a's, B derives no terminal strings.
  - Why? The only production for B leaves a B in the sentential form.
- Thus, S derives nothing, and the language is empty.

Discovery Algorithms

- 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.

Picture of Discovery

And so on ...



Testing Whether a Variable Derives Some Terminal String

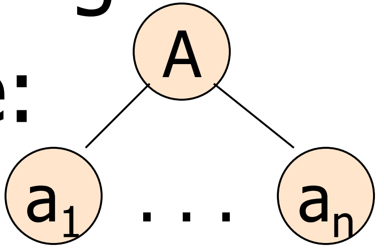
- Basis: If there is a production A -> w, where w has no variables, then A derives a terminal string.
- Induction: If there is a production A -> α, where α consists only of terminals and variables known to derive a terminal string, then A derives a terminal string.

Testing – (2)

- Eventually, we can find no more variables.
- 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

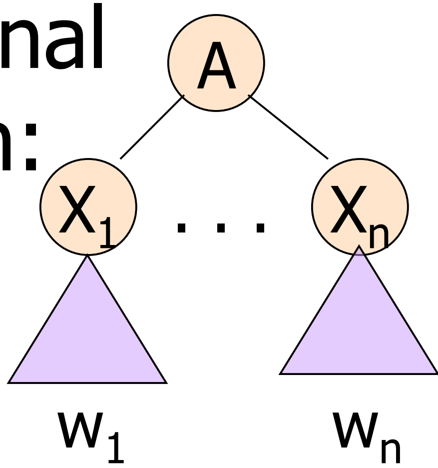
- The proof is an induction on the height of the least-height parse tree by which a variable A derives a terminal string.
- Basis: Height = 1. Tree looks like:



- 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 Eliminate Variables That Derive Nothing

1. Discover all variables that derive terminal strings.
2. For all other variables, remove all productions in which they appear in either the head or body.

Example: Eliminate Variables

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

Unreachable Symbols

- Another way a terminal or variable deserves to be eliminated is if it cannot appear in any derivation from the start symbol.
- Basis: We can reach S (the start symbol).
- Induction: if we can reach A, and there is a production A -> α, then we can reach all symbols of α.

Unreachable Symbols – (2)

- Easy inductions in both directions show that when we can discover no more symbols, then we have all and only the symbols that appear in derivations from S.
- Algorithm: Remove from the grammar all symbols not discovered reachable from S and all productions that involve these symbols.

Eliminating Useless Symbols

- A symbol is *useful* if it appears in some derivation of some terminal string from the start symbol.
- Otherwise, it is *useless*.
  Eliminate all useless symbols by:
  1. Eliminate symbols that derive no terminal string.
  2. Eliminate unreachable symbols.

Example: Useless Symbols – (2)

S -> AB, A -> C, C -> c, B -> bB

- If we eliminated unreachable symbols first, we would find everything is reachable.
- A, C, and c would never get eliminated.

Why It Works

- After step (1), every symbol remaining derives some terminal string.
- After step (2) the only symbols remaining are all derivable from S.
- In addition, they still derive a terminal string, because such a derivation can only involve symbols reachable from S.

Epsilon Productions

- We can almost avoid using productions of the form A -> ϵ (called *ϵ-productions*).
  - The problem is that ϵ cannot be in the language of any grammar that has no ϵ–productions.
- Theorem: If L is a CFL, then L-{ϵ} has a CFG with no ϵ-productions.

Nullable Symbols

- To eliminate ϵ-productions, we first need to discover the *nullable symbols* = variables A such that A =>* ϵ.
- Basis: If there is a production A -> ϵ, then A is nullable.
- Induction: If there is a production A -> α, and all symbols of α are nullable, then A is nullable.

Example: Nullable Symbols

S -> AB, A -> aA | ϵ, B -> bB | A

- Basis: A is nullable because of A -> ϵ.
- Induction: B is nullable because of B -> A.
- Then, S is nullable because of S -> AB.

Eliminating ϵ-Productions

- Key idea: turn each production $A \rightarrow X_1 \ldots X_n$ into a family of productions.
- For each subset of nullable X's, there is one production with those eliminated from the right side "in advance."
  - 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.

Example: Eliminating ϵ-Productions

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

- 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.

Why it Works

- Prove that for all variables A:
  1. If $w \neq \epsilon$ and $A =>^*_{old} w$, then $A =>^*_{new} w$.
  2. If $A =>^*_{new} w$ then $w \neq \epsilon$ and $A =>^*_{old} w$.
- Then, letting A be the start symbol proves that $L(new) = L(old) - \{\epsilon\}$.
- (1) is an induction on the number of steps by which A derives w in the old grammar.

Proof of 1 – Basis

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

Proof of 1 – Induction

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

Induction – Continued

- By the IH, if $w_i \neq \epsilon$, then X_i $=>*_{new}$ 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 $w_i \neq \epsilon$.
  - Note: they all can't be ϵ, because $w \neq \epsilon$.
- Follow a use of this production by the derivations X_i $=>*_{new}$ w_i to show that A derives w in the new grammar.

Unit Productions

- A *unit production* is one whose body consists of exactly one variable.
- These productions can be eliminated.
- 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.

Unit Productions – (2)

- Find all pairs (A, B) such that A =>* B by a sequence of unit productions only.
- Basis: Surely (A, A).
- Induction: If we have found (A, B), and B -> C is a unit production, then add (A, C).

Proof That We Find Exactly the Right Pairs

- By induction on the order in which pairs (A, B) are found, we can show A =>* B by unit productions.
- Conversely, by induction on the number of steps in the derivation by unit productions of A =>* B, we can show that the pair (A, B) is discovered.

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.

Cleaning Up a Grammar

- Theorem: if L is a CFL, then there is a CFG for L – {ϵ} that has:
  1. No useless symbols.
  2. No ϵ-productions.
  3. No unit productions.
- I.e., every body is either a single terminal or has length $\geq$ 2.

Cleaning Up – (2)

- Proof: Start with a CFG for L.
- Perform the following steps in order:
  1. Eliminate ϵ-productions.
  2. Eliminate unit productions.
  3. Eliminate variables that derive no terminal string.
  4. Eliminate variables not reached from the start symbol.

Must be first. Can create unit productions or useless variables.

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.

Proof of CNF Theorem

- Step 1: "Clean" the grammar, so every body is either a single terminal or of length at least 2.
- Step 2: For each body $\neq$ a single terminal, make the right side all variables.
  - For each terminal *a* create new variable A_a and production A_a -> a.
  - Replace *a* by A_a in bodies of length $\geq$ 2.

Example: Step 2

- Consider production A -> BcDe.
- We need variables A_c and A_e. with productions A_c -> c and A_e -> e.
  - Note: you create at most one variable for each terminal, and use it everywhere it is needed.
- Replace A -> BcDe by A -> BA_cDA_e.

CNF Proof – Continued

- Step 3: Break right sides longer than 2 into a chain of productions with right sides of two variables.
- Example: A -> BCDE is replaced by A -> BF, F -> CG, and G -> DE.
  - F and G must be used nowhere else.

Example of Step 3 – Continued

- Recall A -> BCDE is replaced by A -> BF, F -> CG, and G -> DE.
- In the new grammar, A => BF => BCG => BCDE.
- More importantly: Once we choose to replace A by BF, we must continue to BCG and BCDE.
  - Because F and G have only one production.