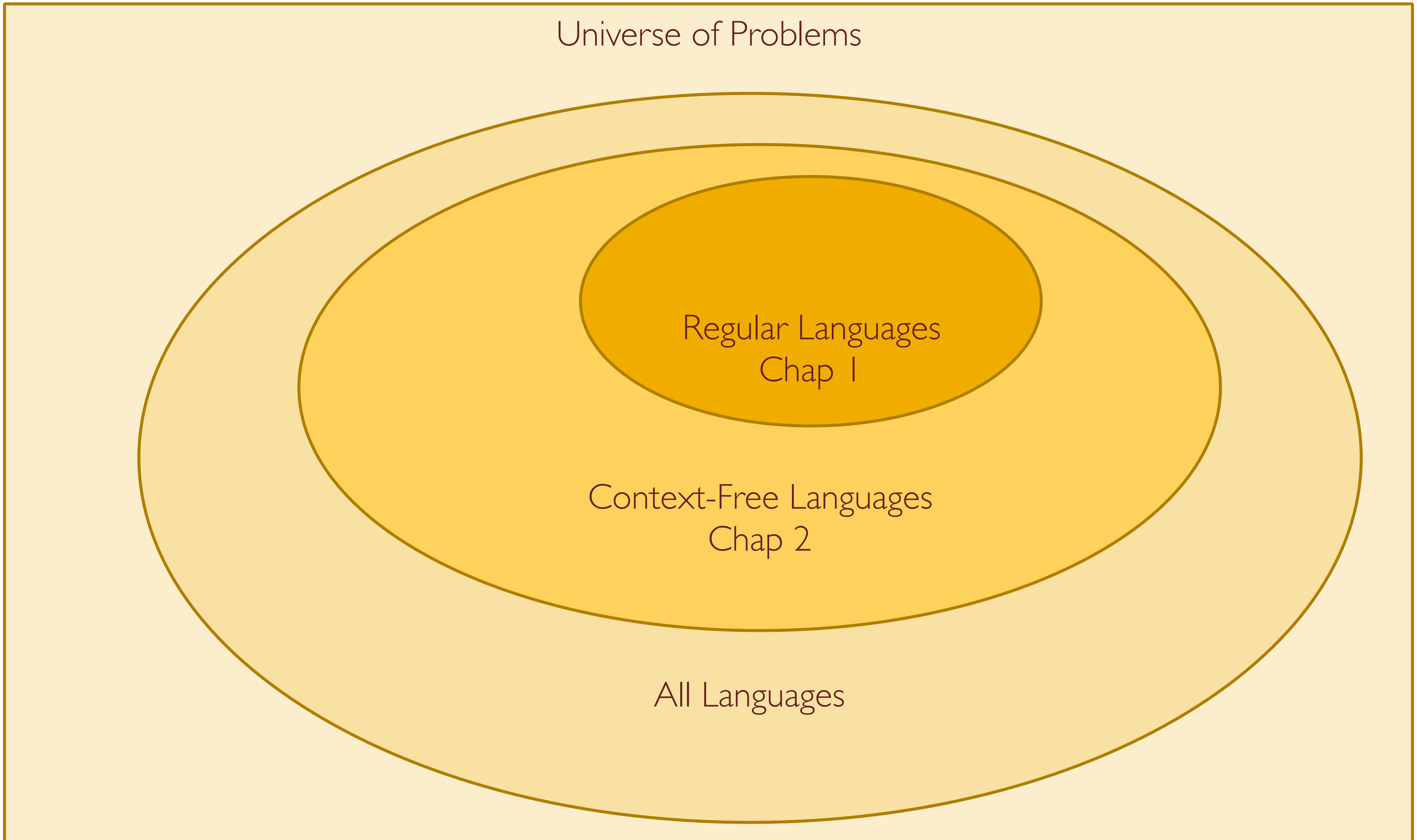
Computing Theory

COMP 147

The Space of Problems



Overview – Chapter 2

- Context-free languages:
  - CFLs include languages that were excluded from regular languages due to bounded memory

- Context-free Grammars
  - CFGs are a notation for describing CFLs
  - Languages definable by CFG ⇔ CFLs
  - Analogous to regex for regular languages
  - Recursive definitions

Overview – Chapter 2

- Pushdown Automata (PDA)
  - Automata that also have a stack
  - Stacks provide unbounded memory
  - Languages recognized by PDA = CFLs

- Pumping Lemma for CFLs
  - How do we show a language is not context-free?

General Grammars

- Grammars define the syntax (structure) of a language
    - Usually defined by rules that can generate legal strings (sentences) in the language
    - The set of strings that can be generated by the rules of the grammar is the language of the grammar

Context-Free Grammars

- 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.

Context-Free Grammars

- 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 a“productions”b involve only concatenation.

Example

CFG for $\{ 0^n 1^n \mid n \geq 1 \}$

- Productions:

  S -> 01

  S -> 0S1

- Basis: 01 is in the language.

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

English Grammar

$$
\begin{aligned}
\langle \text{SENTENCE} \rangle &\rightarrow \langle \text{NOUN-PHRASE} \rangle \langle \text{VERB-PHRASE} \rangle \\
\langle \text{NOUN-PHRASE} \rangle &\rightarrow \langle \text{CMPLX-NOUN} \rangle \mid \langle \text{CMPLX-NOUN} \rangle \langle \text{PREP-PHRASE} \rangle \\
\langle \text{VERB-PHRASE} \rangle &\rightarrow \langle \text{CMPLX-VERB} \rangle \mid \langle \text{CMPLX-VERB} \rangle \langle \text{PREP-PHRASE} \rangle \\
\langle \text{PREP-PHRASE} \rangle &\rightarrow \langle \text{PREP} \rangle \langle \text{CMPLX-NOUN} \rangle \\
\langle \text{CMPLX-NOUN} \rangle &\rightarrow \langle \text{ARTICLE} \rangle \langle \text{NOUN} \rangle \\
\langle \text{CMPLX-VERB} \rangle &\rightarrow \langle \text{VERB} \rangle \mid \langle \text{VERB} \rangle \langle \text{NOUN-PHRASE} \rangle \\
\langle \text{ARTICLE} \rangle &\rightarrow \texttt{a} \mid \texttt{the} \\
\langle \text{NOUN} \rangle &\rightarrow \texttt{boy} \mid \texttt{girl} \mid \texttt{flower} \\
\langle \text{VERB} \rangle &\rightarrow \texttt{touches} \mid \texttt{likes} \mid \texttt{sees} \\
\langle \text{PREP} \rangle &\rightarrow \texttt{with}
\end{aligned}
$$

Attempt to generate a sentence.

Is this grammar recursive?

Context-Free Grammars

- 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.

Context-Free Grammars

- A production or substitution rule 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

Context-free Grammar

A CFG over $\Sigma = \{ \#, 0, 1 \}$:

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

3 *substitution rules* (also called productions)

symbols left of → are *variables*

- 1st rule identifies the *start symbol*

symbols right of → are strings of *variables* or *terminals*

- Σ = set of terminals

Generating strings from a CFG

- A CFG over $\Sigma = \{ \#, 0, 1 \}$:

  $A \rightarrow 0A1$

  $A \rightarrow B$

  $B \rightarrow \#$

  - Start with start symbol
  - Loop:
    - Replace any variable with the RHS of a rule for that variable

Attempt to generate some strings.

Derivations

- 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

- We say $\alpha A\beta \Rightarrow \alpha\gamma\beta$ if $A \to \gamma$ is a production.

- Example:
  S -> 01;
  S -> 0S1.

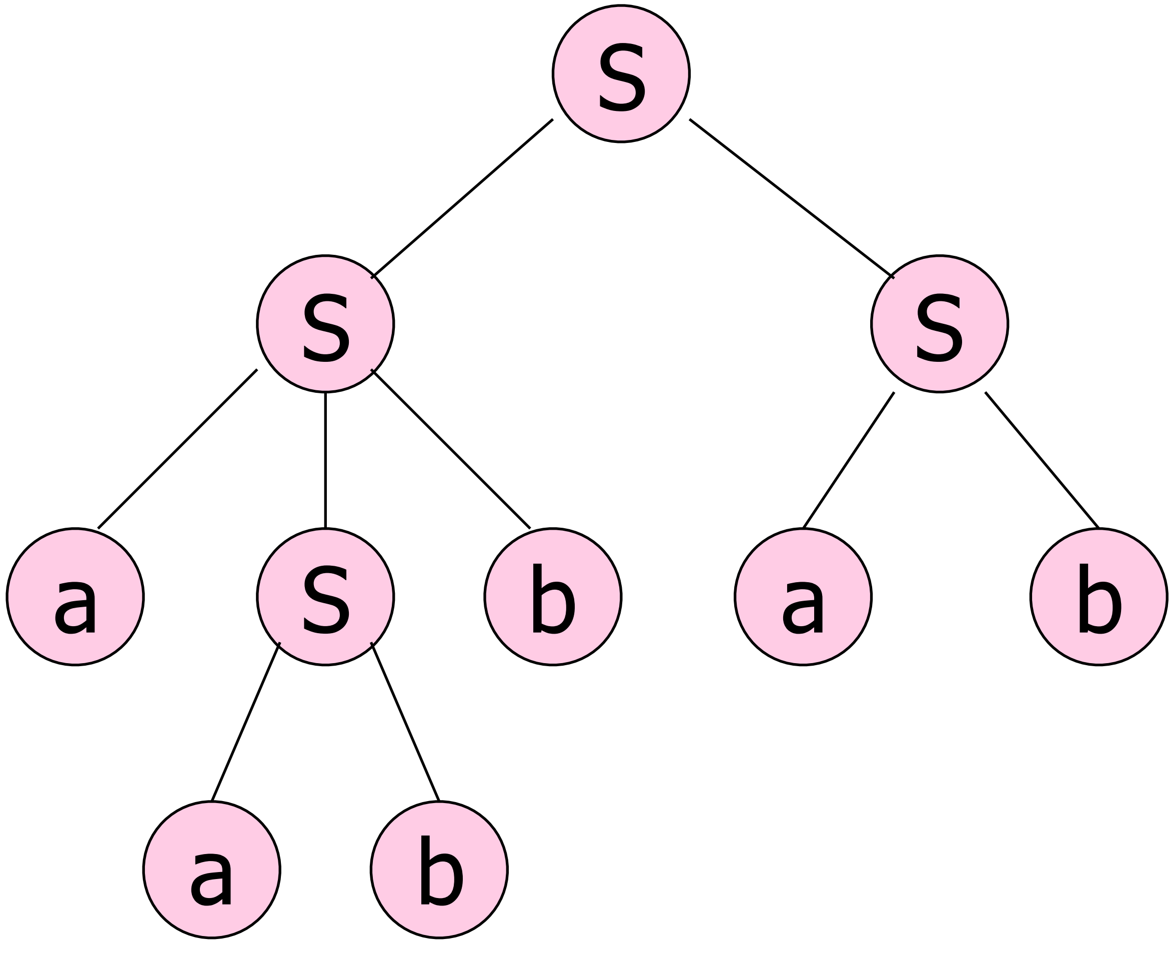
- S => 0S1 => 00S11 => 000111.

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 | aSb | ab



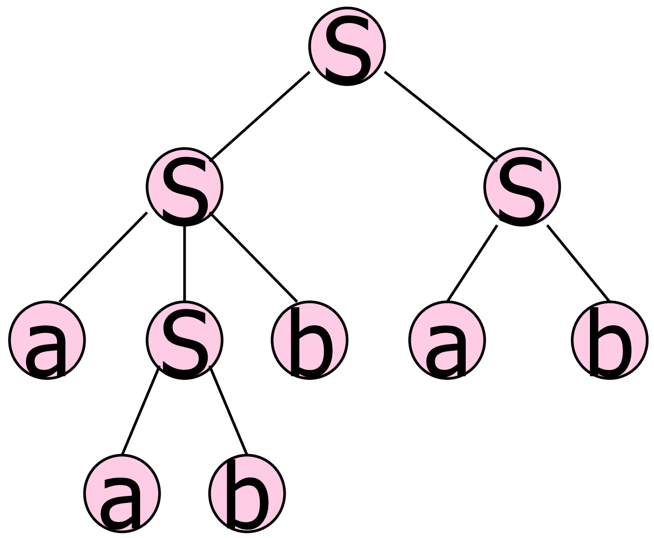
Parse Trees

Which of the following cannot appear as the label of a node aleaf or interior nodeb of a parse tree?

1. Variable S
2. Terminal a
3. Terminal b
4. Terminal String ab

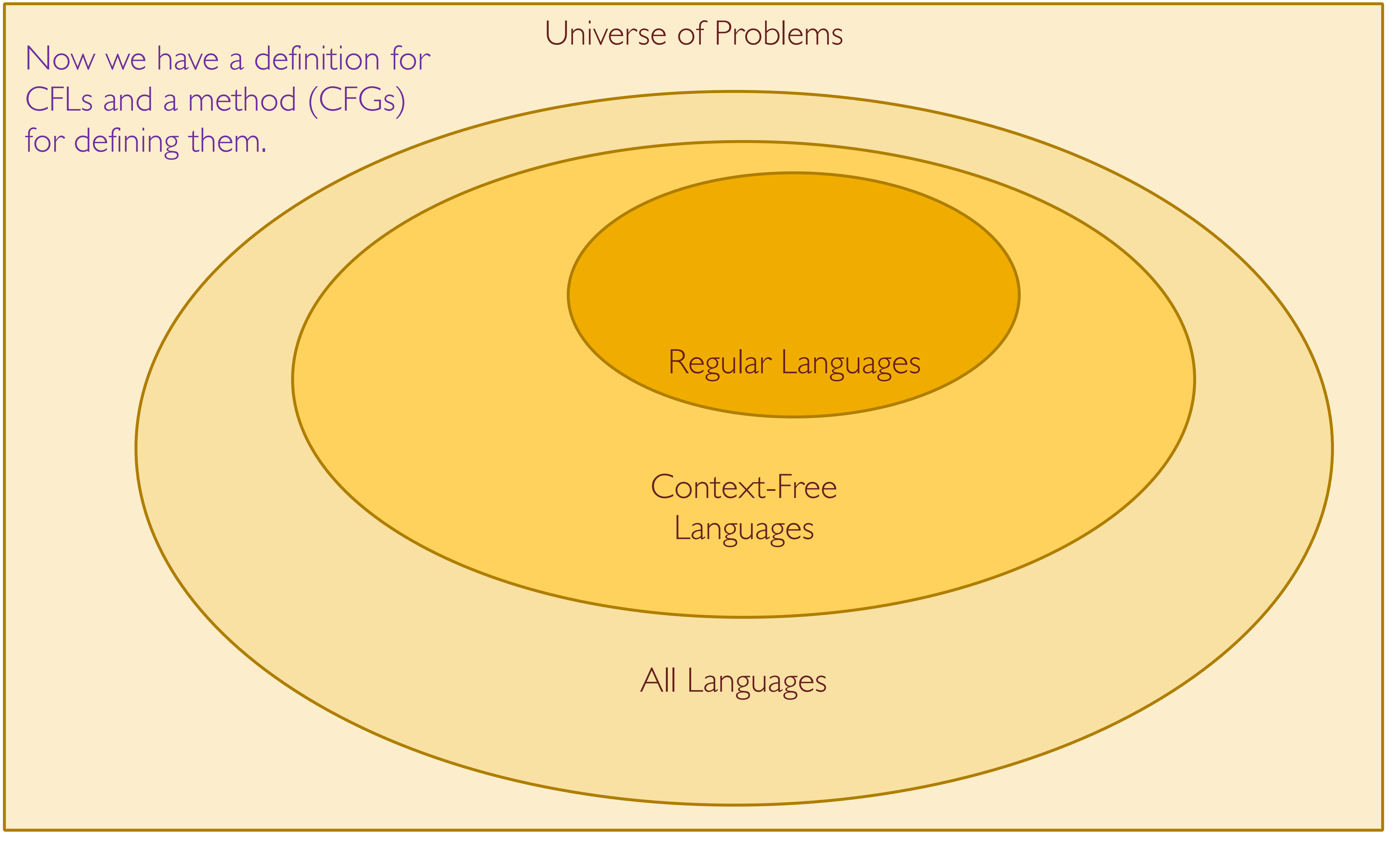
Yield of a Parse Tree

- 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 aabbab



The Space of Problems

Now we have a definition for CFLs and a method (CFGs) for defining them.



Last time

- Consider the following Grammar: S -> 0S1| 01.
- Give derivations and Parse Trees for string 0011

CFG: Formal Definition

DEFINITION 2.2

A ***context-free grammar*** is a 4-tuple (V, Σ, R, S), where

1. V is a finite set called the ***variables***,
2. Σ is a finite set, disjoint from V, called the ***terminals***,
3. R is a finite set of ***rules***, with each rule being a variable and a string of variables and terminals, and
4. $S \in V$ is the start variable.

Language of a Grammar

- If G is a CFG, then L(G), the language of G, is {w | S =>* w}.

- Example:
G has productions S -> **ε** and S -> 0S1.

- $L(G) = \{0^n1^n \mid n \geq 0\}$.

Example:
S->ab | aSb | SS

Then, which of the following strings is not in the language defined by this CFG?

a) aababbab

b) aaabbabaabbb

c) aaabaabbab

d) abababab

Last time

- Consider the following Grammar: S->aSb | ab | SS

- Derivation for aabbab

Example: Leftmost Derivations

- Leftmost derivation: forcing the leftmost variable
- Balanced-parentheses grammar:
  S -> SS | aSb | ab
- $S \Rightarrow_{lm} SS \Rightarrow_{lm} aSbS \Rightarrow_{lm} aabbS \Rightarrow_{lm} aabbab$
- Thus, $S \Rightarrow^*_{lm} aabbab$
- $S \Rightarrow SS \Rightarrow Sab \Rightarrow aSbab \Rightarrow aabbab$ is a derivation, but not a leftmost derivation.

Example: Rightmost Derivations

- Balanced-parentheses grammar:

  S -> SS | aSb | ab

- $S \Rightarrow_{rm} SS \Rightarrow_{rm} Sab \Rightarrow_{rm} aSbab \Rightarrow_{rm} aabbab$

- Thus, $S \Rightarrow^*_{rm} aabbab$

- S => SS => SSS => SabS => ababS => ababab is neither a rightmost nor a leftmost derivation.

Which of the following is a rightmost derivation of the grammar S -> SS | aSb | ab?

1. S=>SS=>SSS=>abSS=>abaSbS=>abaabbS=>abaabbab
2. S=>SS=>SSS=>SaSbS=>SaabbS=>Saabbab=>abaabbab
3. S=>SS=>SSS=>SSab=>SaSbab=>Saabbab=>abaabbab
4. S=>SS=>abS=>abSS=>abaSbS=>abaabbS=>abaabbab

Examples

- What is this grammar in english?
  S->aSa | bSb | ε

- Give a grammar for the language L
  L = { w | w has even number of 0’s}

- Give a grammar for language generate by the regular expression 00*11*

Examples-Solutions

- What is this grammar in english?
  S->aSa | bSb | ε
  - string are palindromes
- Give a grammar for the language L
  L = { w | w has even number of 0's}
  - S->1S | 0A | ε
    A->1A | 0S
  - (S represents strings with even 0's, A strings with odd 0's)
- Give a grammar for language generate by the regular expression 00*11*
  - S->AB
    A->0 | 0A
    B->1 | 1B

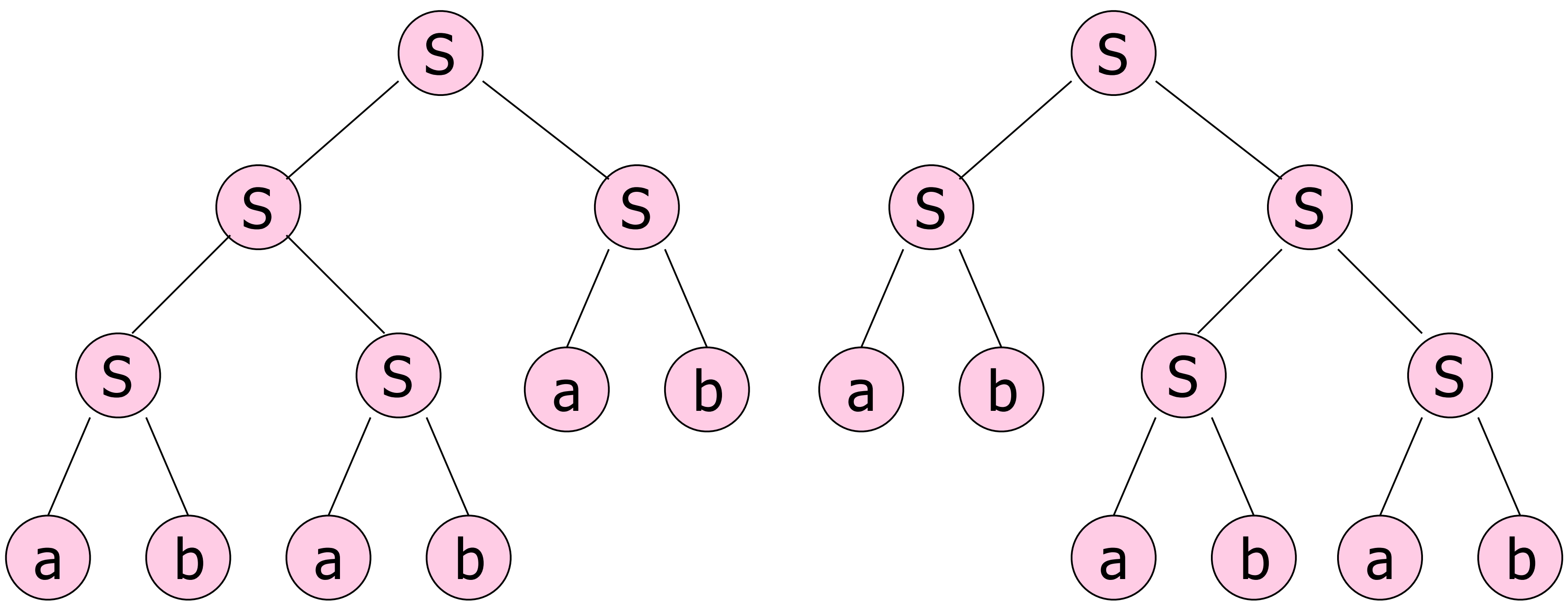
Ambiguous Grammars

- A CFG is ambiguous if there is a string in the language that is the yield of two or more parse trees (or two different leftmost (or rightmost) derivations)

- Example: S -> SS | aSb | ab

- Two parse trees for ababab on next slide.

Ambiguous Grammars



Example

- $S \rightarrow S + S \mid S * S \mid 1 \mid 2 \mid 3 \mid 4$
- Can you give 2 parse trees for 2 + 3 * 4?

Ambiguity is a Property of Grammars, not Languages

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

B, the start symbol, derives balanced strings.

- B -> aRB | ε
- R -> b | aRR

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

Example: Unambiguous Grammar

- B -> aRB | **ε**   R -> b | aRR
- 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 -> aRB if the next symbol is “a”; use **ε** if at the end.
  - If we need to expand R, use R -> b if the next symbol is “b” and aRR if it is “a”.

The Parsing Process

Remaining Input:

aabbab

Next symbol

Steps of leftmost derivation:

B

B -> aRB | ε   R -> b | aRR

The Parsing Process

Remaining Input:

abbab

Next symbol

Steps of leftmost derivation:

B

aRB

B -> aRB | ε

R -> b | aRR

The Parsing Process

Remaining Input:

bbab

Next symbol

Steps of leftmost derivation:

B

aRB

aaRRB

B -> aRB | ε

R -> b | aRR

The Parsing Process

Remaining Input:

bab

Next symbol

Steps of leftmost derivation:

B

aRB

aaRRB

aabRB

B -> aRB | ε

R -> b | aRR

The Parsing Process

Remaining Input:

ab

Next symbol

Steps of leftmost derivation:

B

aRB

aaRRB

aabRB

aabbB

B -> aRB | ε

R -> b | aRR

The Parsing Process

Remaining Input:

b

Next symbol

Steps of leftmost derivation:

B aabbaRB

aRB

aaRRB

aabRB

aabbB

B -> aRB | ε R -> b | aRR

The Parsing Process

Remaining Input:

Next
symbol

Steps of leftmost derivation:

B

aRB

aaRRB

aabRB

aabbB

aabbaRB

aabbabB

B -> aRB | ε

R -> b | aRR

The Parsing Process

Remaining Input:

Next symbol

Steps of leftmost derivation:

B
aRB
aaRRB
aabRB
aabbB
aabbaRB
aabbabB
aabbab

B -> aRB | ε

R -> b | aRR

LL(1) Grammars

- As an aside, a grammar such
  B -> aRB | ε R -> b | aRR,

- 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)

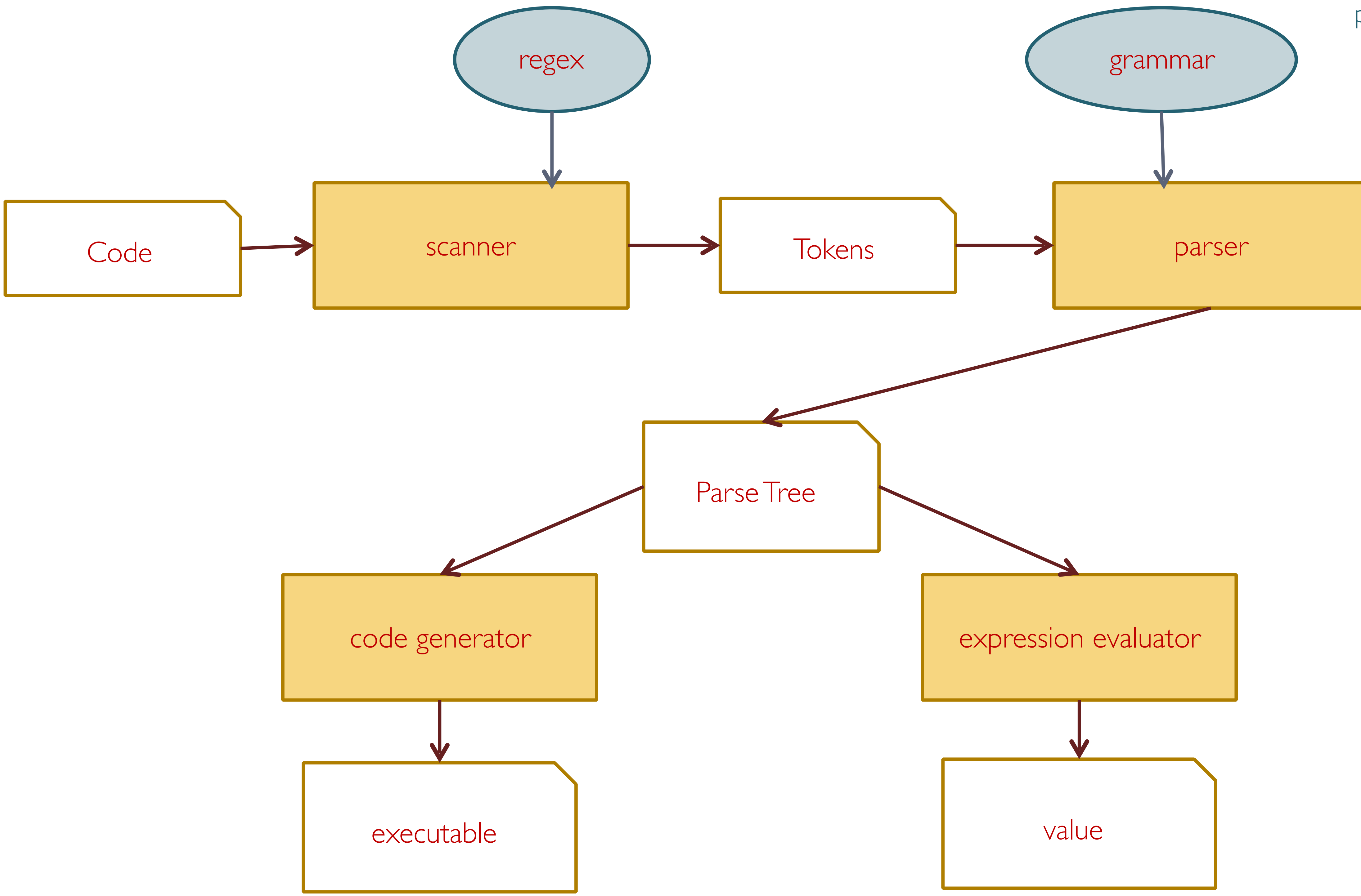
- Most programming languages have LL(1) grammars.
- LL(1) grammars are never ambiguous.

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.

Compilers

scanners and
parsers are defined
by languages



Chomsky Normal Form

- Convenient to have grammars in a simplified 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).

Chomsky Normal Form

- Motivation:
  - Every string of length n can be derived in 2n - 1 steps
  - Makes proof of pumping lemma for CFL easier
  - Makes description of parsing algorithms like CYK easier

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.
- Thus, S derives nothing, and the language is empty.

Example

In this CFG, which variable(s) (upper-case letters) do(es) not derive any terminal string?

$S \rightarrow ABC$; $A \rightarrow ab$; $C \rightarrow Bb$; $C \rightarrow aAb$; $B \rightarrow AB$

1. S
2. B
3. B, S
4. A, B, S

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)

- Algorithm: Remove from the grammar all symbols not discovered reachable from S and all productions that involve these symbols.

Example

In this CFG, S is the start symbol. Which symbols are unreachable?

$S \rightarrow AB$; $A \rightarrow bc$; $B \rightarrow aa$; $C \rightarrow De$; $D \rightarrow aa$

1. e, C, D
2. a, e, C, D
3. C, D
4. e, D

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:
  - Eliminate symbols that derive no terminal string.
  - 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.

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 $A \to X_1 \ldots X_n$ 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 -> A~~BC~~ | AB | ~~AC~~ | ~~BC~~ | A | B | ~~C~~

- A -> aA | a

- B -> bB | b

Note: C is now useless.

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).

Example

For the following CFG, find all pairs (A,B) such that $A \Rightarrow * B$ by a sequence of unit productions only:

$S \rightarrow AB|Aa;$
$A \rightarrow cD;$
$B \rightarrow aCb|C|A;$
$C \rightarrow D;$
$D \rightarrow a|b|c$

1)(B,C) (B,A) (C,D)
2) (B,C) (B,A) (B,D) (C,D)
3) (B,C) (B,A) (B,D) (C,D) (A,D)
4) (B,C) (B,A) (B,D) (C,D) (A,D)

Cleaning Up a Grammar

- Theorem: if L is a CFL, then there is a CFG for L – {**ε**} that has:
  - No useless symbols.
  - No **ε**-productions.
  - No unit productions.

- I.e., every body is either a single terminal or has length $\geq$ 2.

Cleaning Up a Grammar

- Proof: Start with a CFG for L.

- Perform the following steps in order:
  - Eliminate **ε**-productions.
  - Eliminate unit productions.
  - Eliminate variables that derive no terminal string.
  - 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:
  - A -> BC (body is two variables).
  - A -> a (body is a single terminal).
- Theorem: If L is a CFL, then L – {ε} has a CFG in CNF.

CNF Theorem
Every CFG -> CNF

- Step 1: "Clean" the grammar, so every body is either a single terminal or of length at least 2. (slide: cleaning up a grammar)

- 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.