CS 3100, Models of Computation, Spring 20, Lec18

Ganesh Gopalakrishnan
School of Computing
University of Utah
Salt Lake City, UT 84112

URL: bit.ly/3100s20Syllabus

SCHOOL OF COMPUTING
THE UNIVERSITY OF UTAH

Connections between regular and CFL

- All regular languages are CFL
  - Why?
    - Can you argue via : “given a DFA, I can obtain a PDA” ?
    - Can you argue via : ”given a DFA, I can obtain a CFL?

- The first is easy. (Obtain answer in class.)

Connections between regular and CFL

- All regular languages are CFL
  - Why?
    - Can you argue via : “given a DFA, I can obtain a PDA” ?
    - Can you argue via : ”given a DFA, I can obtain a CFL?
- For the second, let’s state a mechanical construction algorithm

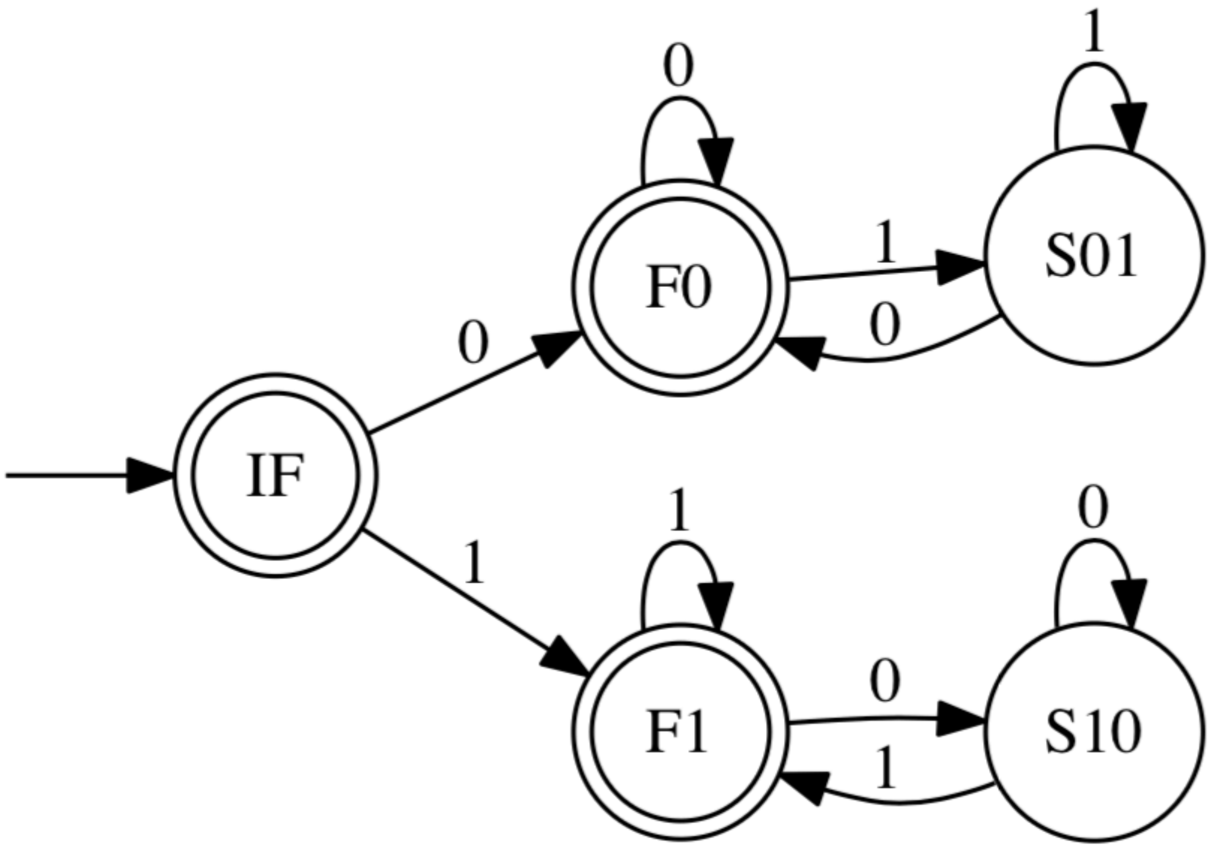
DFA to CFL

- Given a DFA with initial state I
- And final states F1 F2 …

- Obtain a set of productions
  - Let state “S” of the CFG correspond to (or model) the initial state I
  - For every move A : a -> B in the DFA, introduce a rule like that in the CFG
  - For every final state F1 F2 …, introduce a rule F1 → ‘’ F2 → ‘’

- Try an example now

Write a CFG for this DFA



Every DFA has an “easy” CFG one can obtained “just by staring at the DFA”

Hence all regular languages are also context-free !!!

Solution (obtaining a purely right-linear CFG)

"S" of the CFG models state "IF" of the DFA

S → '' (since IF is a final state)

S → 0 F0 (there are these transitions in the DFA)

S → 1 F1 (there are these transitions in the DFA)

F0 → '' (it is a final state)

F1 → '' (it is a final state)

…finish this construction…

Use the info from prev. slide to handle this CFG

What is S's language?

Can you write it as a regular expression?

S → a S b | b Y | Y a

Y → b Y | a Y | ‘’

Draw a parse-tree for ababb using this grammar

S → a S b | b Y | Y a

Y → b Y | a Y | ‘’

Calculator with Parse-tree Drawing

- This will be part of Asg4
- This Jove file draws parse-trees for you!
- Look for
  - First_Jove_Tutorial/ACMDSP/Calculator_with_Parse_Tree_Drawing.ipynb

- Look for First_Jove_Tutorial/ACMDSP/Drive_Chatty_Parser.ipynb
  - Will be in Asg4
  - Shows how Jove's internals work !!

Now we will study linearity

- There are purely left-linear and purely right-linear.
  - They are equivalent.
- Read about this conversion from the book (we will review it before the exam to the extent needed)
  - Read the portions around the "dog picture"

Obtaining Purely L. Lin. from Purely R. Lin.

Rotating pair of dogs trick to convert a
Purely right linear CFG
Into a Purely left linear CFG

Example:

S -> 0 A B   becomes

Sr -> Br Ar 0

Etc.

Check previous example.
See how I turned the purely right-linear
Into a purely left-linear CFG

Studying consistency and completeness

- Consistency – only good strings
- Completeness – all good strings

Exercise: Equal number of a’s and b’s

- Begin with template
- S -> …..a…..b….. Or S -> …..b….a…..
- Replacing all the … with S is an overkill
- Just having S -> a S b | b S a is insufficient (why? Ans: Incomplete!)
- Imagine just enough generality to cover all cases
  - We arrive at S -> S a S b | S b S a | “
  - Or it can be S -> a S b S | b S a S | “ (one of these)

Consistency: It is through structural induction

```
S ->  '' | aSbS | bSaS
```

When a string structure (in this case) is elaborated as above
i.e. S (“strings from s”) are derived via

- ‘’ – assert that this string is consistent : has equal a’s and b’s
- By induction hypothesis, examine all CFG rules

  - a followed by simpler strings s1 from S followed by b followed by simpler strings s2 from S [this is for the aSbS rule] forming string a s1 b s2
  - …or the bSaS rule similarly…

- By induction hypothesis, s1 and s2 are consistent
- Then the induction step says a s1 b s2 is consistent

Completeness Illustrated on the Grammar of Equal number of a’s and b’s

```
S -> '' | aSbS | bSaS
```

Completeness argues that ALL strings can be derived.

- Induction strategy:
  - Assume that all “substrings of a given string s can be derived”
  - Then show that the string s itself can be derived

Completeness Illustrated on the Grammar of Equal number of a’s and b’s

```
S ->  '' | aSbS | bSaS
```

Completeness argues that ALL strings can be derived.

- Induction strategy:
  - Assume that all “substrings of a given string s can be derived”
  - Then show that the string s itself can be derived

- Suppose we assume S -> a S b | b S a | ‘’
- Can we argue it is complete?

Completeness Illustrated on the Grammar of Equal number of a's and b's

```
S ->  '' | aSbS | bSaS
```

Completeness argues that ALL strings can be derived.

- Induction strategy:
  - Assume that all "substrings of a given string s can be derived"
  - Then show that the string s itself can be derived

- Suppose we assume S -> a S b | b S a | ''
- Can we argue it is complete?

If we consider a long string a…………..a

we immediately see that there is NO parse tree from "S"

hence incomplete!

Completeness Illustrated on the Grammar of Equal number of a's and b's

```
S ->   '' | aSbS | bSaS
```

Completeness argues that ALL strings can be derived.

- This tells us that there are FOUR cases to consider

a…………….a (consider this case now; all other cases have to be considered)

a…………….b

b…………….a

b………………b

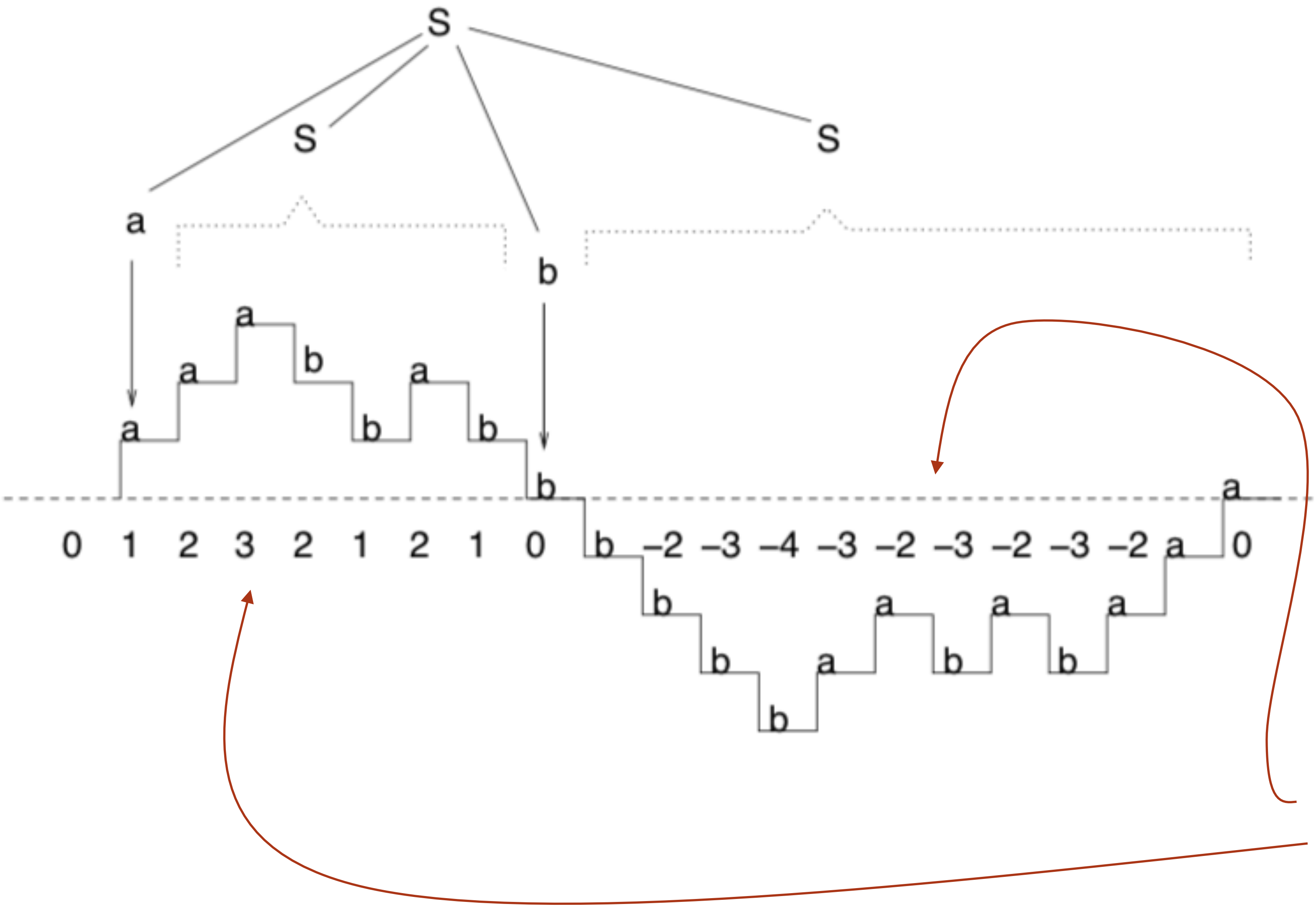
Arguing Completeness: Hill/Valley Plots

The “hill/valley plot” is just a way to visualize CFL completeness proofs involving counting arguments.

For other completeness proofs, we will need to invent other ways of visualizing the proof (there is no general approach).

Studying one approach thoroughly is all we aim for.

Arguing Completeness: Hill/Valley Plots



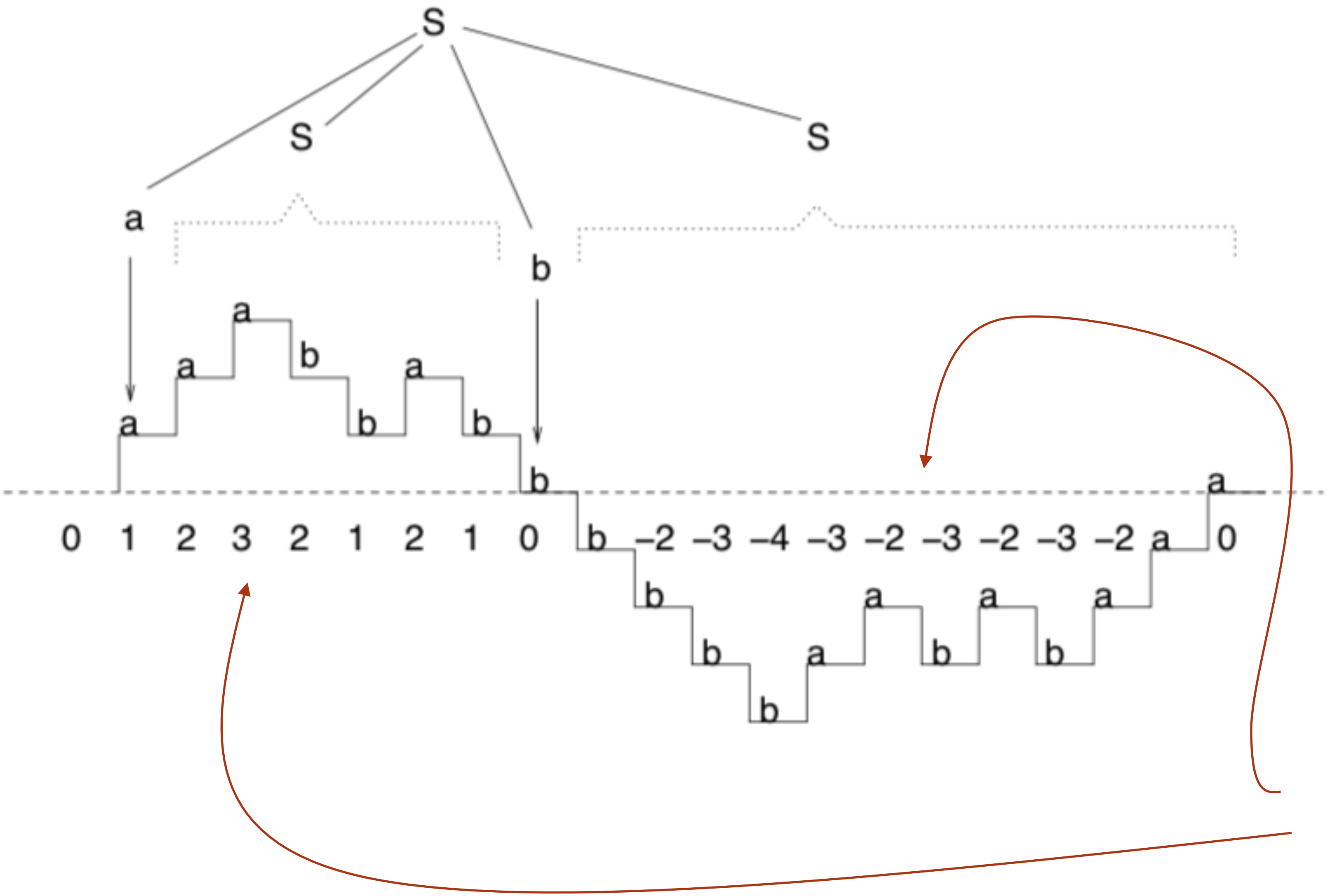
As soon as we draw a string that starts and ends with "a"

AND has equal # of 0's and 1's

We realize that it must achieve a count > 0 somewhere, then a count = 0, then achieve a count < 0, and the final "a" must restore the count back to 0.

So there are smaller strings that have equal counts!
Induct on them!

Arguing Completeness: Hill/Valley Plots



Induction argument:

The small strings in this picture are derivable from S

Then we can piece together a whole parse-tree that explains that the WHOLE string can also be derived!

Two Nontrivial Exercises

- Recall L_{ww} = { w w : w in Sigma^* } ?
- L_{ww} is not a CFL
- But the complement of L_{ww} is a CFL
  - Write its CFG (maybe part of Asg-4)

- Write a CFG for { w : w in {0,1}^* and #1(w) > #0(w) }
  - Part of Asg-4 most likely

Grammar Desiderata

- Grammars must be consistent
  - With respect to the language you want to capture
- Grammars must be complete
  - With respect to the language you want to capture
- Are there other desirable properties of grammars?
  - Yes! Grammars must be unambiguous.
- Reason grammars must be unambiguous:
  - Grammars are used to generate code, implement interpreters, etc.
  - Example: Our calculator grammar (let us take a look)

Grammars vs. Ambiguity

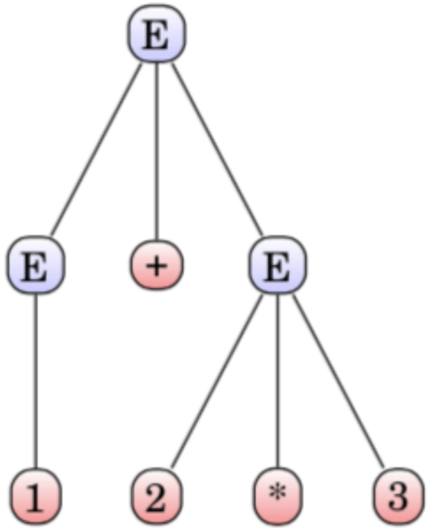
- A grammar G1 may be ambiguous
- Another grammar G2 such that L(G1) = L(G2) may be unambiguous
  - I.e. no string has two distinct parse trees

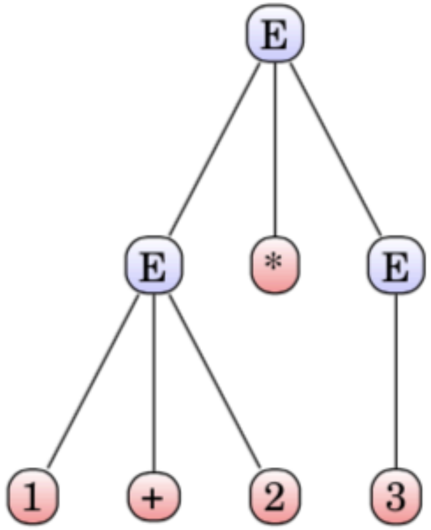
- While L(G1) = L(G2), there is only one parse-tree for L(G2)
- Parse trees determine how
  - A calculator evaluates
  - A compiler generates code
- Let us review the expression grammar (next slide)

Ambiguity and Disambiguation

```
E -> 1 | 2 | 3 | ~E | E+E | E*E | (E)
```

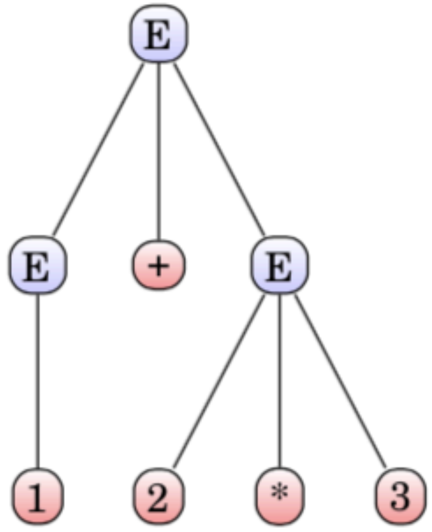
```
E -> E+T | T
T -> T*F | F
F -> 1 | 2 | 3 | ~F | (E)
```



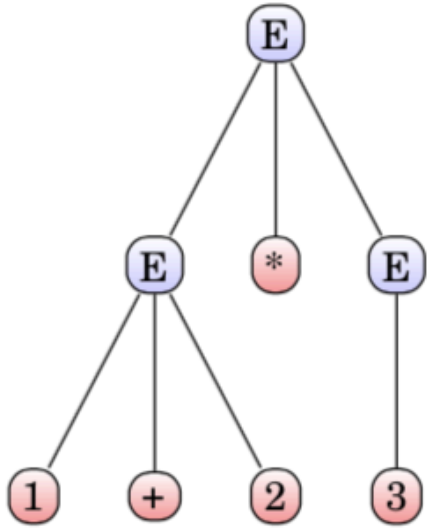


Ambiguity and disambiguation

```
E -> 1 | 2 | 3 | ~E | E+E | E*E | (E)
```




```
E -> E+T | T
T -> T*F | F
F -> 1 | 2 | 3 | ~F | (E)
```



Gist : by changing the grammar,

- The same set of strings are still derivable
- Ambiguity goes away !!
- The basic idea is to “layer the grammar”

Calculator Problem of Assignment

- In this assignment, you get to see, for the first time, how the theory of regular languages and context-free languages helps you build a calculator
- You must understand the calculator
- You must then modify it to have a new operator called ”!” or successor
- 3 + !4 = 3 + (4+1) = 8
- !(3+!4 - !!1) * !5 = ?

Notes on the Calculator Problem

- We relied on PLY to disambiguate for us
  - Defines associativity
  - Defines precedences
- But we can also modify the grammar and achieve this
  - More on that on Thursday next week
  - We will also study Jove’s parsers (we have 3 of them inside Jove)

Inherently Ambiguous CF Languages

$$L_{abORbc} = \{ a^i b^j c^k \ : \ (i = j) \, \text{or} \, (j = k) \}$$

No matter which CFG we try --- layering or otherwise --- ambiguity NEVER goes away !!!

The proof that the above language is inherently ambiguous is long, and is skipped.

But I can give you papers that cover it (if you wish).

Which are CFL and which aren't?

1. $L_{P0} = \{w \ : \ w \in \Sigma^*\}$
2. $L_{P1} = \{ww^R \ : \ w \in \Sigma^*\}$
3. $L_{P2} = \{waw^R \ : \ a \in (\{\varepsilon\} \cup \Sigma), \ w \in \Sigma^*\}$
4. $L_{eq01} = \{0^n 1^n \ : \ n \geq 0\}$
5. $L_{ww} = \{ww \ : \ w \in \Sigma^*\}$
6. $L_{w\#w} = \{w\#w \ : \ w \in \Sigma^*\}$, where # is a separator.
7. $L_{eq010} = \{0^n 1^n 0^n \ : \ n \geq 0\}$
8. $L_{eq012} = \{0^n 1^n 2^n \ : \ n \geq 0\}$

Getting to Pump CFGs: Part 1 of 4

```
S -> ( S ) | T | ''
T -> [ T ] | T T | ''
```

Getting to Pump CFGs: Part 2 of 4

```
S => (S) => (( T )) =>  (( [ T ] )) => (( [  ] ))
             ^                ^
             Occurrence-1     Occurrence-2
             Use T => [T]     Use T => ''
```

Getting to Pump CFGs: Part 3 of 4

```
S => (S) => (( T ))  => (( [ T ] )) => (( [[ T ]] )) => (( [[ ]] ))
               ^             ^               ^
               Occurrence-1  Occurrence-2    Here,
               Use T => [T]  Use T => [T]    use T => ''
```

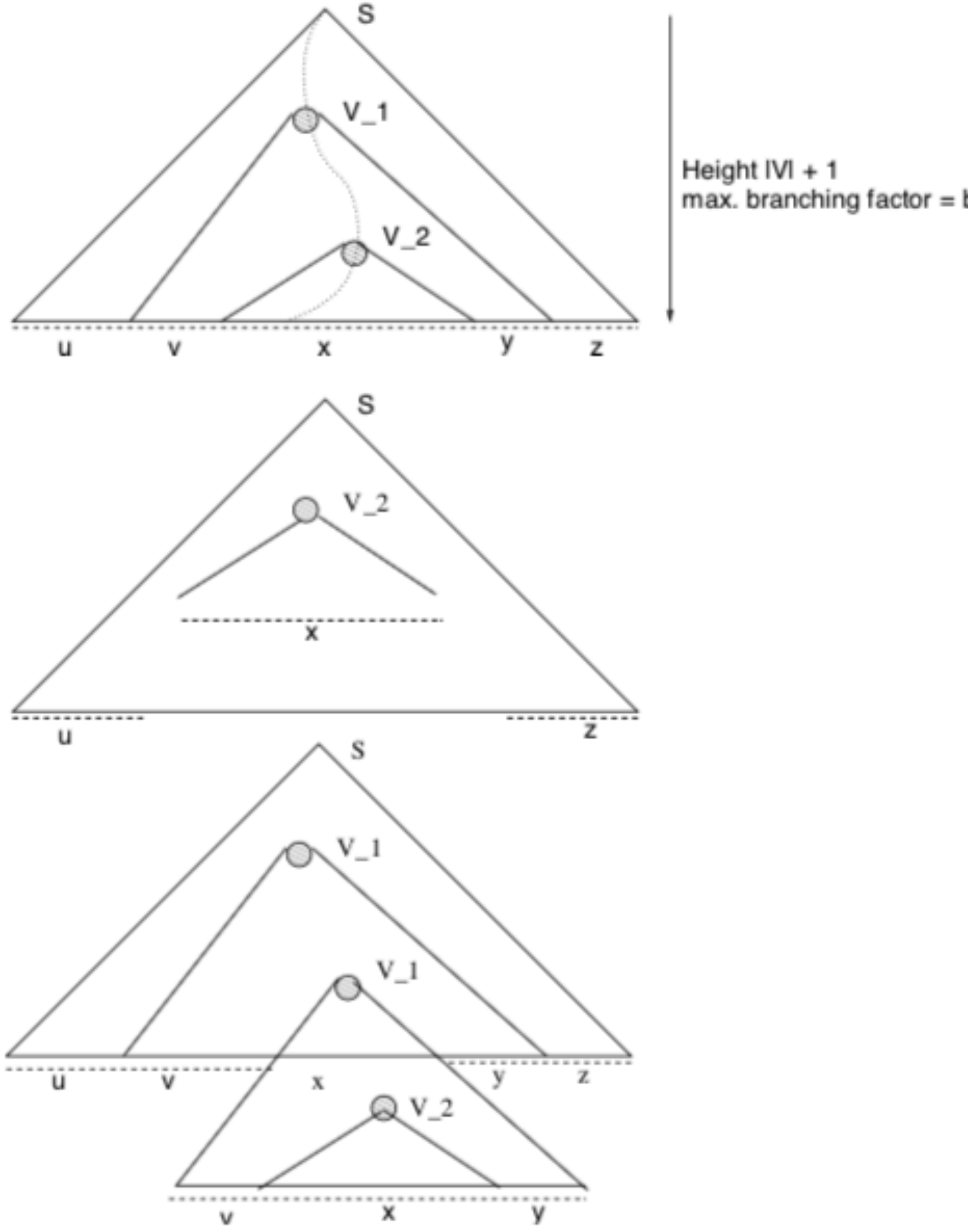
Getting to Pump CFGs: Part 4 of 4

```
S => (S) => (( T ))  => ((  ))
               ^
               Here, use T => ''
```

Summary of Example

```
Given that this         We infer that this                OR, this
derivation exists:      derivation exists:                derivation exists:
==================      ==================                ==================
S => (S)                S => (S)                          S => (S)
  => (( T ))              => (( T ))                        => (( T ))
  => (( [ T ] ))          => (( [ T ] ))                    => ((   ))
  => (( [   ] ))          => (( [[ T ]] ))
                          => (( [[[ T ]]] ))
                          => ...
                          => (( [[[[[[[[[ T ]]]]]]]]] ))
                          => (( [[[[[[[[[   ]]]]]]]]] ))
```

CFL PL in Pictures



The CFL PL finally! (pictures)

Theorem 11.9: Given any CFG $G = (N, \Sigma, P, S)$, there exists a number p such that given a string w in $L(G)$ such that $|w| \geq p$, we can split w into $w = uvxyz$ such that $|vy| > 0$, $|vxy| \leq p$, and for every $i \geq 0$, $uv^ixy^iz \in L(G)$.

The CFL PL finally! (words)

- Suppose L_{ww} were a CFL.
- Then the CFL Pumping Lemma would apply.
- Let p be the pumping length associated with a CFG of this language.
- Consider the string $0^p 1^p 0^p 1^p$ which is in L_{ww}.
- The segments v and y of the Pumping Lemma are contained within the first $0^p 1^p$ block, in the middle $1^p 0^p$ block or in the last $0^p 1^p$ block, and in each of these cases, it could also have fallen entirely within a 0^p block or a 1^p block.
- In each case, by pumping up or down, we will then obtain a string that is not within L_{ww}. $\square$

Exercise on CFL PL

Show that $\{ 0^{i^2} : i \geq 0 \}$ is not a CFL

Exercise on CFL PL from Sipser

Show that this language is not a CFL

{ w # x : w is a substring of x, where w in {a,b}* }
here # is a simple delimiter character

A substring is a contiguous sequence of strings embedded in another

A subsequence is more general – need not be contiguous