CS 3100, Models of Computation, Spring 20, Lec16

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

URL: bit.ly/3100s20Syllabus

SCHOOL OF COMPUTING
THE UNIVERSITY OF UTAH

Motivations for studying DFA and PDA

- DFA
  - Lexers (scanners)
  - Typestate encoding
  - Fast text search, malware filtration, etc…
- PDA
  - Program static analysis
  - Weighted pushdown systems
    - in PPoPP 2020 that I just attended in San Diego

Principles and Practice of Parallel Programming 2020

Talk in PPoPP 2020 on FSM !!

09:35 - 10:25: Main Conference - Scaling (Mediterranean Ballroom)
Chair(s): Zhijia Zhao UC Riverside

09:35 - 10:00 *Talk* ☆ Using Sample-Based Time Series Data for Automated Diagnosis of Scalability Losses in Parallel Programs
Lai Wei Rice University, John Mellor-Crummey Rice University

10:00 - 10:25 *Talk* ☆ Scaling out Speculative Execution of Finite-State Machines with Parallel Merge
Yang Xia The Ohio State University, Peng Jiang The University of Iowa, Gagan Agrawal The Ohio State University

A way to process very long strings using parallel processing

This is a hard problem because FSM are sequential

Idea! Chunk the string … and speculatively process the tail pieces
Then find a way to connect the head with the tail!

The idea was spurred by this paper from 2014
Data-Parallel Finite-State Machines

Todd Mytkowicz, Madanlal Musuvathi, and Wolfram Schulte

Microsoft Research

Research published in ASPLOS 2014

Has spurred considerable research since then

(even in PPoPP 2020, San Diego, that I just attended)

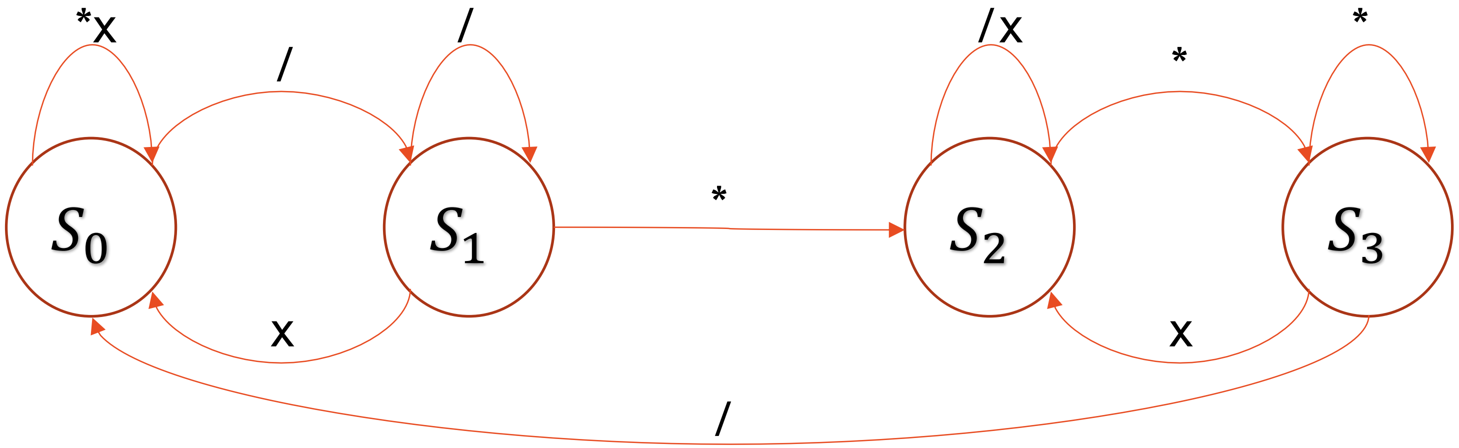
New method to break data dependencies

- Preserves program semantics
- Does not use speculation
- Generalizes to other domains, but this talk focuses on FSM

FSMs contain an important class of algorithms

- Unstructured text (e.g., regex matching or lexing)
- Natural language processing (e.g., Speech Recognition)
- Dictionary based decoding (e.g., Huffman decoding)
- Text encoding / decoding (e.g., UTF8)

Want parallel versions to all these problems, particularly in the context of large amounts of data



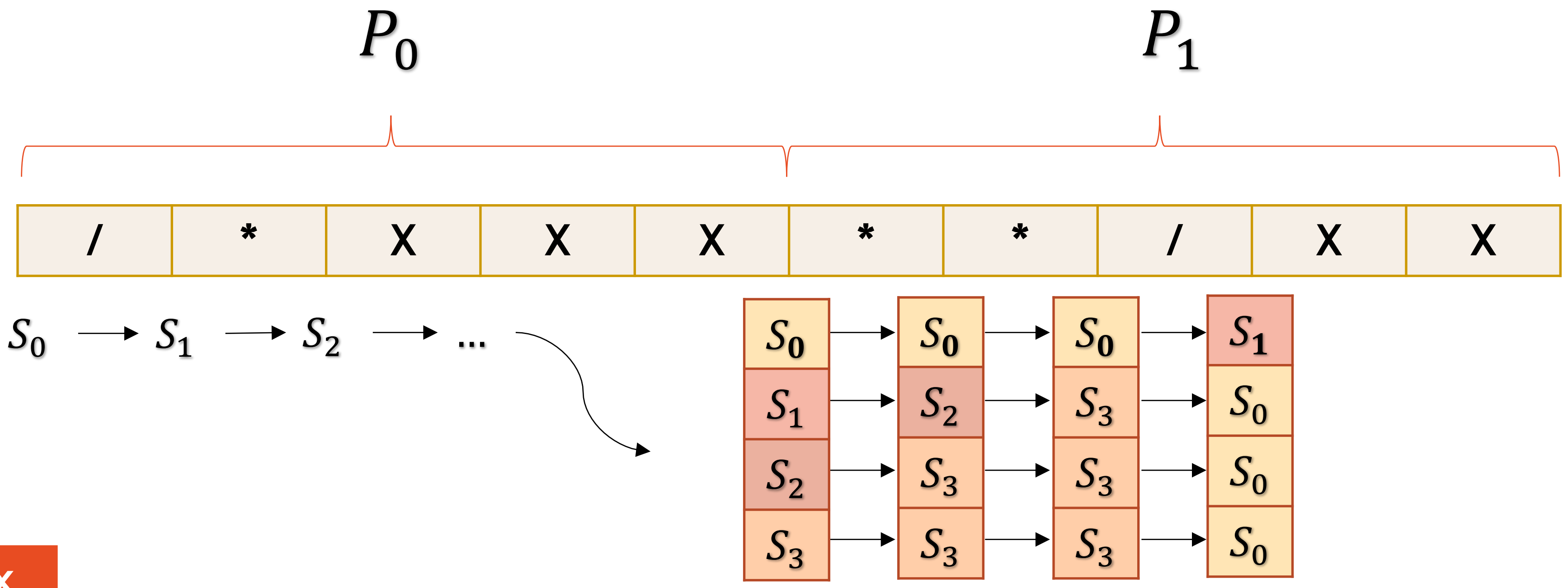
T	/	*	x
S_0	S_1	S_0	S_0
S_1	S_1	S_2	S_0
S_2	S_2	S_3	S_2
S_3	S_0	S_3	S_2

```
state = S0;
foreach(input in)
  state = T[in][state];
```

Data Dependence limits ILP, SIMD, and multicore parallelism

Demo UTF-8 Encoding

Breaking data dependences with enumeration

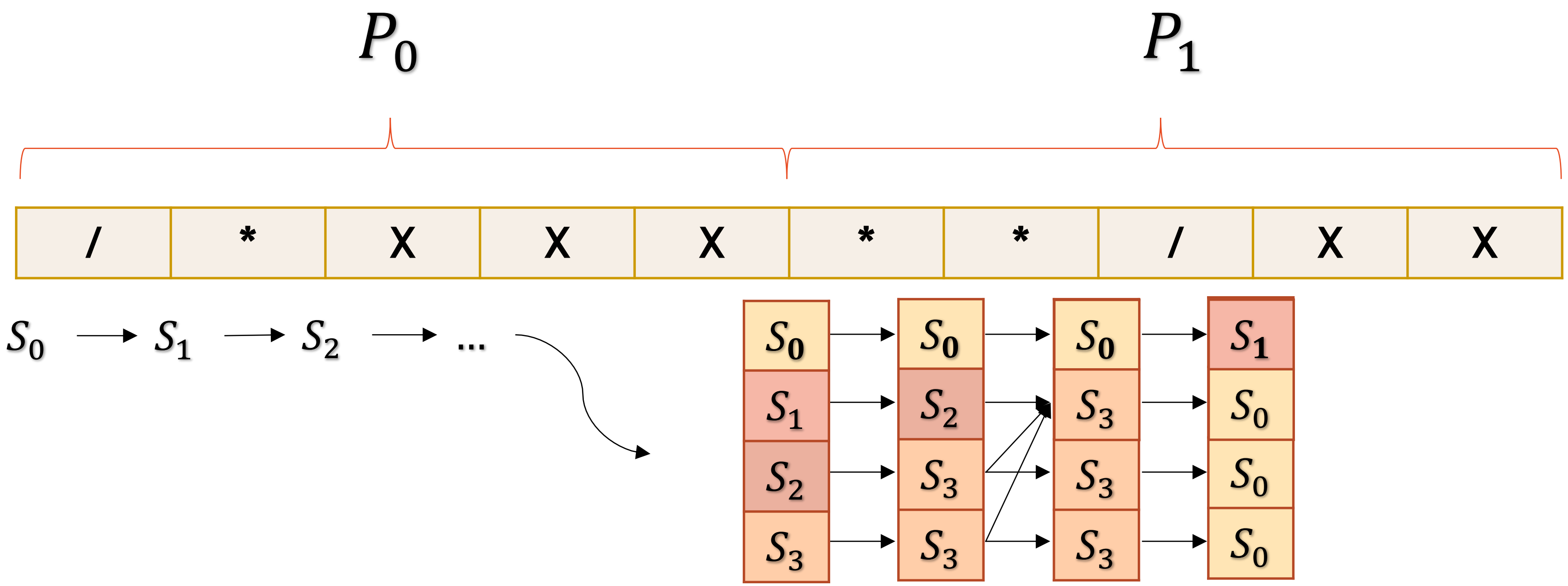


T	/	*	x
S_0	S_1	S_0	S_0
S_1	S_1	S_2	S_0
S_2	S_2	S_3	S_2
S_3	S_0	S_3	S_2

Enumeration breaks data dependences but how do we make it scale?

- Overhead is proportional to # of states

Intuition: Exploit convergence in enumeration



After 2 characters of input, FSM *converges* to 2 unique states

- Overhead is proportional to # of unique states

Back to what we are studying now

- Today:
  - CFG and CFLs
- Two key topics
  - Consistency
    - “Do not do anything bad”

Context-free Grammars (CFG)

A context-free grammar is a four-tuple (N, Σ, S, P), where

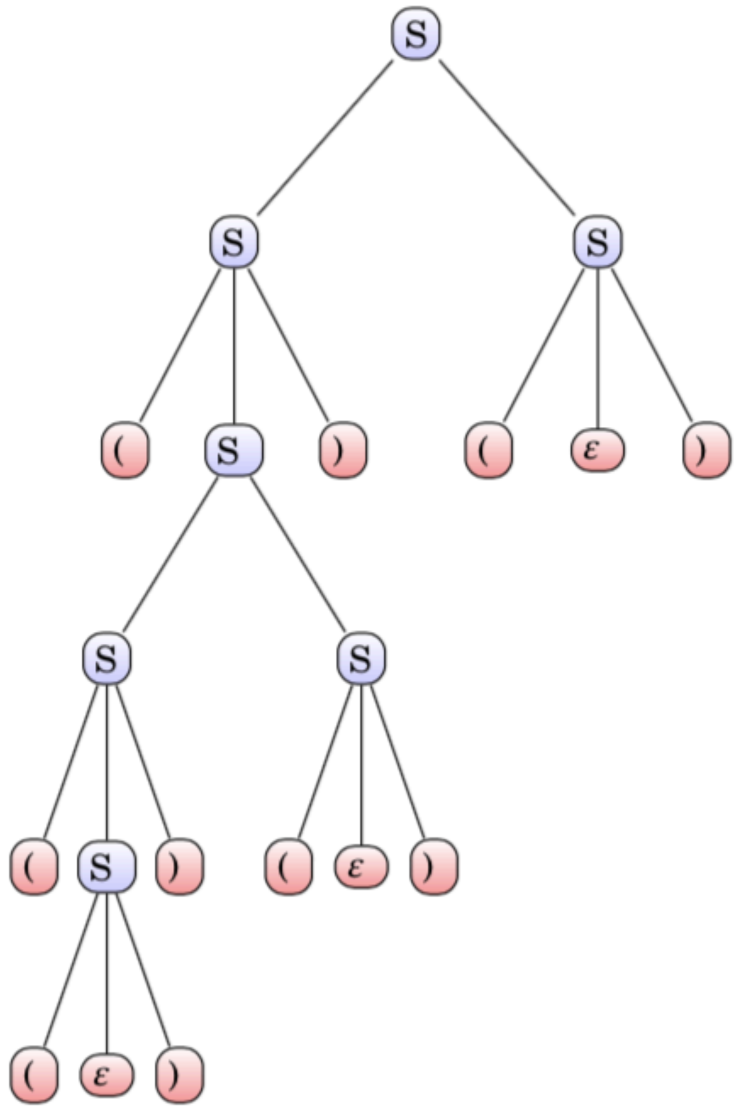
- N is a set of **nonterminals**. In L_{Dyck}, S is the only nonterminal.
- Σ is a set of **terminals**. In L_{Dyck}, the terminals are (and). The name "terminals" suggests places when the recursion of the context-free production rules *terminates*. ε itself can be viewed as a terminal, although strictly speaking, it is not. When we define P below, we will allow the right-hand sides of production rules to contain $\{(N \cup \Sigma)^*$. From that point of view, ε (ASCII '') is an *empty string of terminals*.
- S is the **start symbol** which is one of the nonterminals. In our example, the start symbol is S.
- P is a set of **production rules** which are of the form:
  - $N \rightarrow \{(N \cup \Sigma)^*$, and read "N derives a string of other N and Σ items." Such strings are called **sentential forms**. A terminal-only string is called a **sentence**.

The language of a CFG

The language of a context-free grammar (CFG) G is exactly all those sentences that can be obtained by constructing a *derivation sequence*. In mathematical notation, $L(G) = \{ w : S \Rightarrow^* w \}$. The language of a CFG is called a **context-free language.**

Context-free Grammars: Derivation Sequences

S => SS => (S)S => (SS)S => ((S)S)S =>
(((S))S)S => (((”))S)S => ((())S)S => ((())(S))S =>
((())(”))S => ((())())S => ((())())(S) => ((())())(”) =>
((())())()



Goals of Grammar Design

- We want <u>consistent</u> grammars
  - Grammars that do not yield strings outside the language of interest
  - E.g. S -> (S) | (is inconsistent for the grammar of matched parentheses
  - E.g. for Odd number of 0's, we don't want this grammar
    - OddZ -> 0 | 0 OddZ 0 | ‘’
  - Also,
    - OddZ -> 1 | 0 0 OddZ | 0   is inconsistent

- We want <u>complete</u> grammars
  - Grammars that yield all the strings in the language of interest
  - E.g. for "equal a's and b's," we don't want this grammar [why ?]
    - S -> SabS | SbaS | ‘’

- We want grammars that are <u>non-redundant</u>
  - Complete but with no redundancies
  - E.g. for "equal a's and b's" we like to avoid this grammar [why ?]
    - S -> aSbS | bSaS | SS | ‘’

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 non-ambiguous
- All these properties are difficult to show for large grammars…
  - Being aware of these notions, we can at least “keep an eye”

Summary: Properties of good grammars

- 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
- Grammars must be non-redundant
  - By being redundant, we can add to the ambiguity
    - The redundant production rules can be used to generate redundant parser trees
- Grammars must also be non-ambiguous

Arguing Consistency: Induction on CFG

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

Basis case:

Take the first rule S -> ‘’
Observe that ‘’ is derivable and that has equal a’s and b’s

Inductive cases:

Take each rule, e.g. S – aSbS
By inductive hypotheses, the strings derivable by the RHS `S’ are shorter than the whole string derivable from aSbS, AND contain equal # of a’s and b’s

Arguing Consistency: Induction on CFG

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

Basis case:

Take the first rule S -> ‘’
Observe that ‘’ is derivable and that has equal a’s and b’s

Inductive cases:

Take each rule, e.g. S – aSbS

Based on induction hypothesis in S -> aSbS ,
we are simply adding one more a and one more b --
hence MAINTAINING consistency

Arguing Consistency: Testing your knowledge

Is this grammar consistent for generating all palindromes over {a,b} ?

G1: S -> ‘’

G2: S -> a S a

G3 : S -> ” | a S a | b S b

G4 : ?

(Hint: The above are all consistent but not complete…)

(Strive to make G4 complete … i.e. be able to generate all pals.)

Completeness: Equal number of a's and b's

Completeness means "ALL the correct strings (according to the definition of the language) will be obtainable via the given CFG

Example: Show that this grammar is complete with respect to the intended language " equal number of a's and b's "

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

Completeness: Equal number of a’s and b’s

Basic idea

- By induction hypotheses, assume that all strings in “equal a’s and b’s of length N-2 are generatable via the given productions

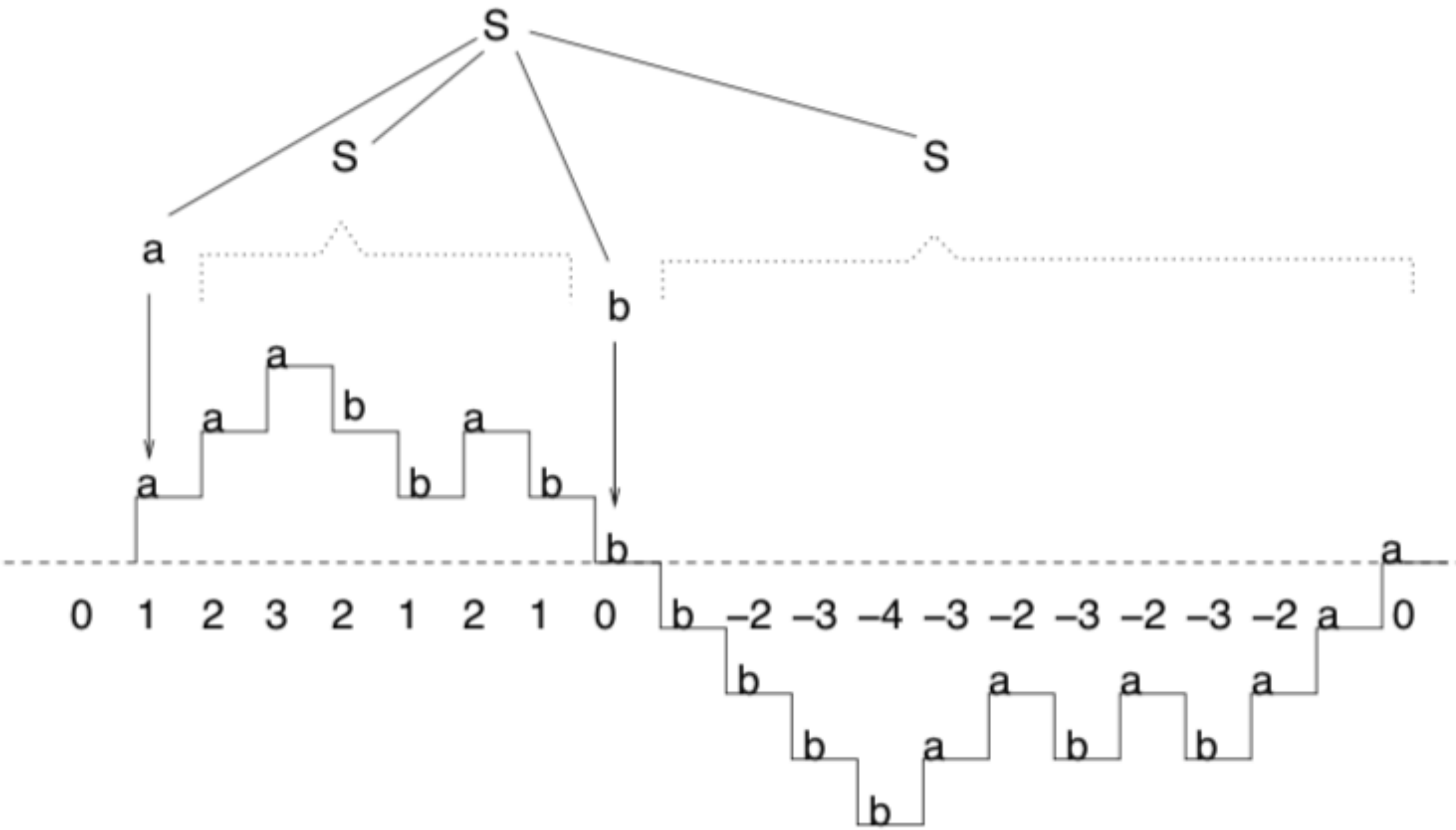
* Argue that all strings of length N are generatable (induction step)

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

Arguing Completeness (in general)

- Find a way to decompose strings in the given language into "shorter strings" via one less derivation step
- That is, we used the next smaller parse tree to get that string
  - There will be many cases here
- Show that the next longer string can be obtained via the given grammar
  - That is, we can build the shorter strings and THEN add one more production rule to finish building the bigger parse tree

Arguing Completeness: Hill/Valley Plots

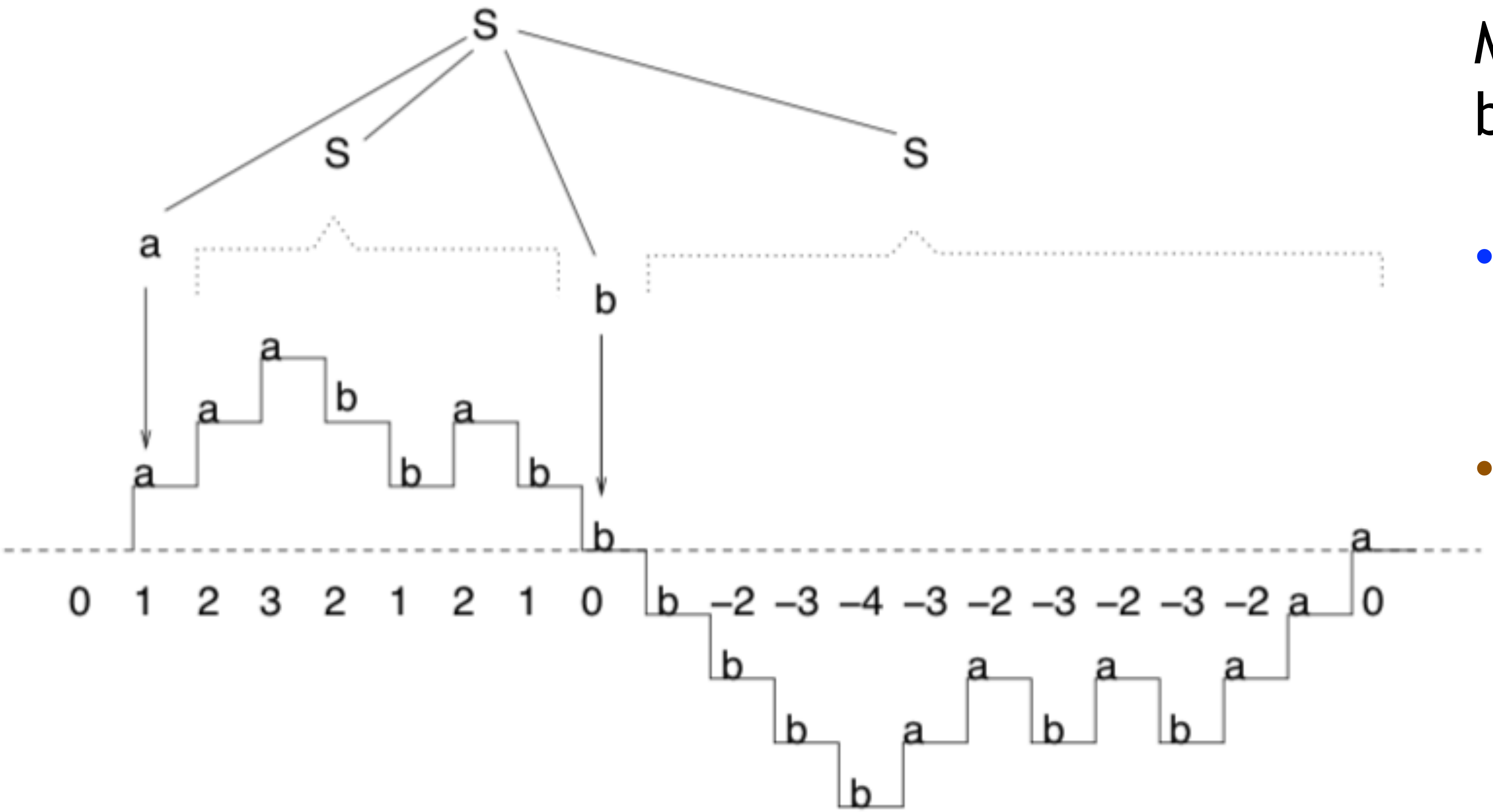


All “equal a’s and b’s strings” look like this in general

Draw it as if “a” takes the plot going up (tally of a’s goes up)

Draw “b”s as if bringing the tally down

Arguing Completeness: Hill/Valley Plots



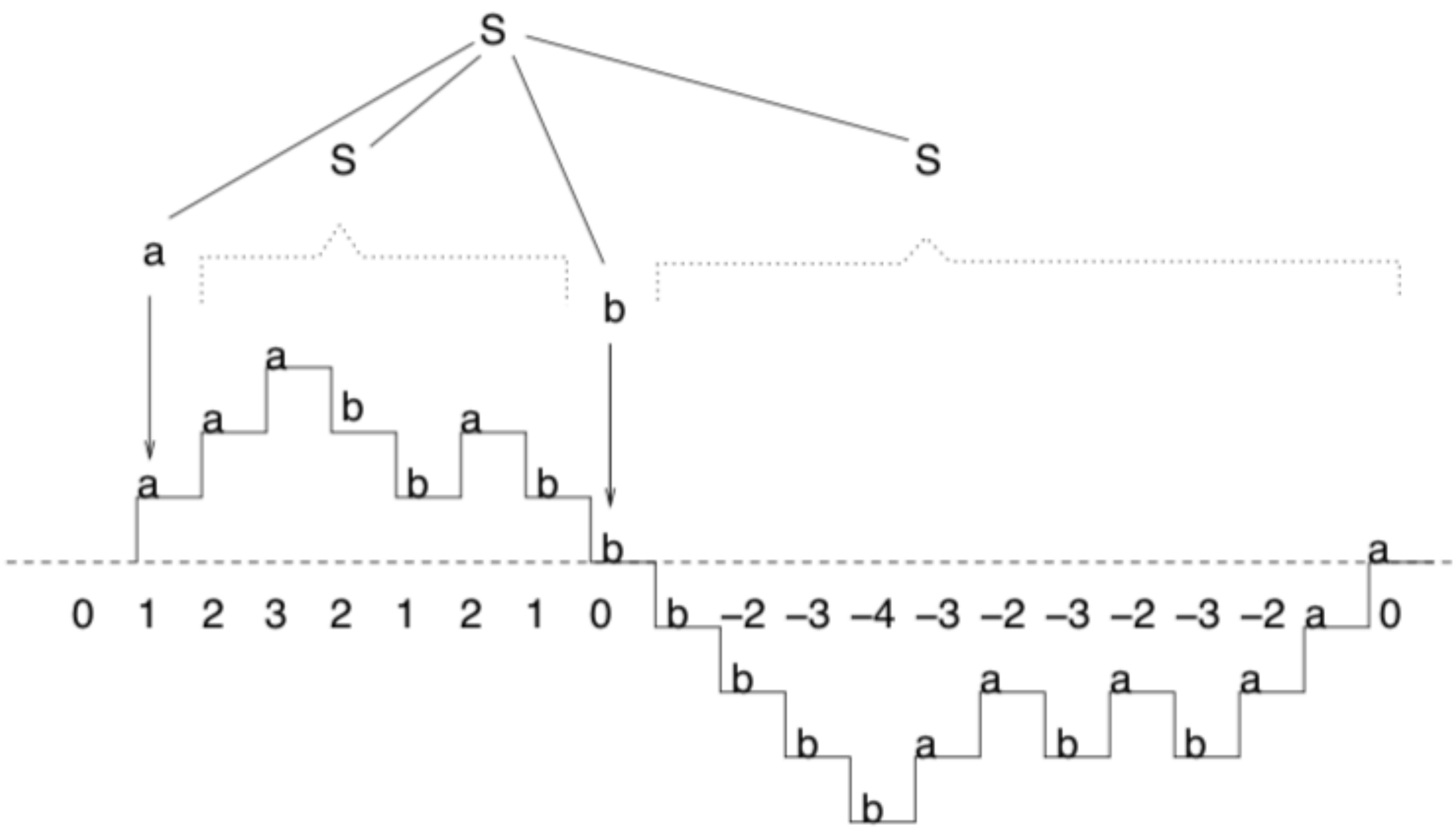
Many cases in equal a's and b's strings

- Starts with a and ends with a (shown to the left)

- Three more cases exist:
  - Starts with b and ends with b
  - Starts with a and ends with b
  - Starts with b and ends with a

- ARGUE FOR ALL CASES!

Arguing Completeness: Hill/Valley Plots



Take one of these cases.

Start and end with a.

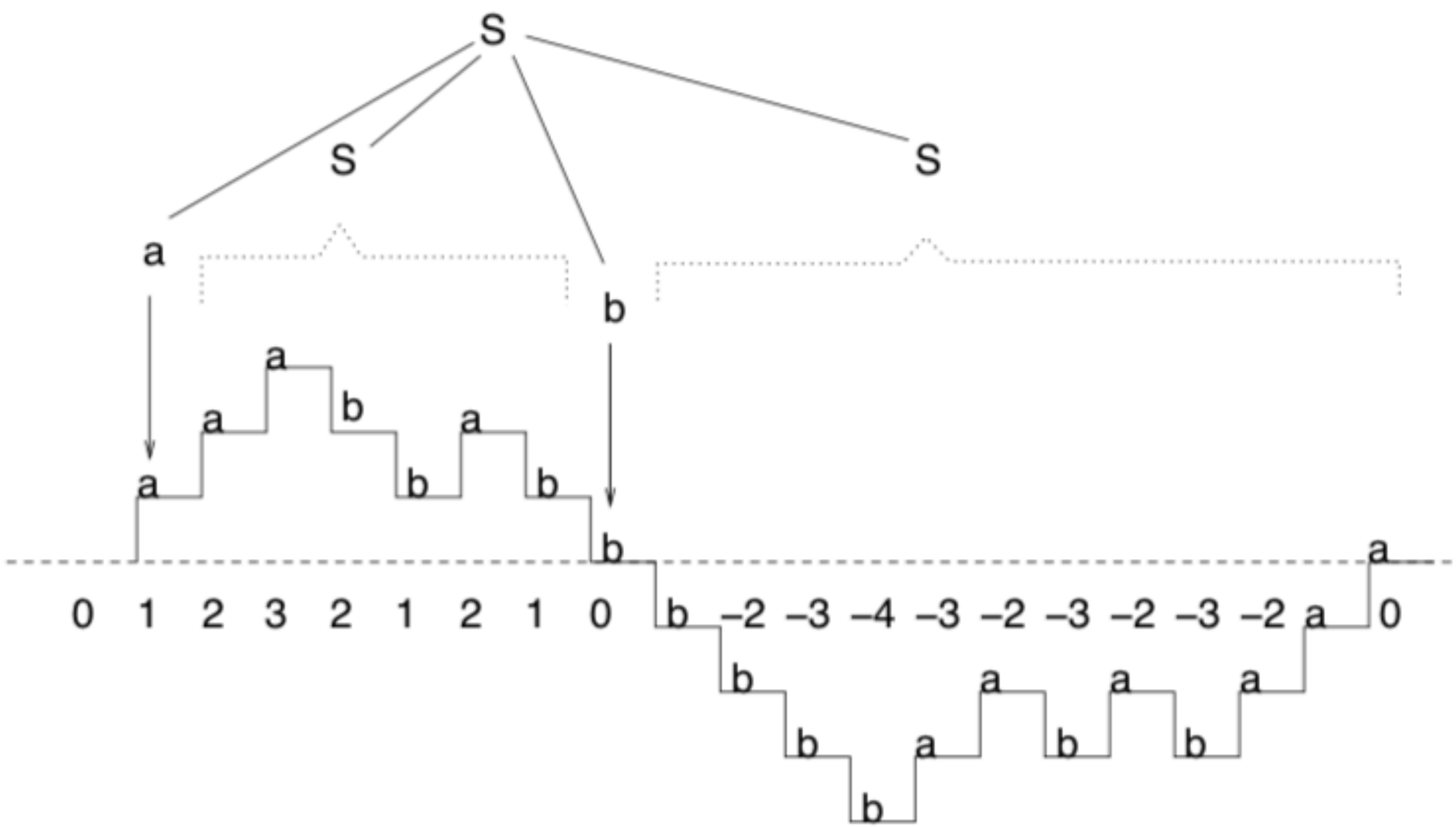
Such a string must have a "hill-valley" plot.

Crosses the X axis.

By induction hypotheses, all "short strings" derivable.

Show longer strings derivable.

Arguing Completeness: Hill/Valley Plots



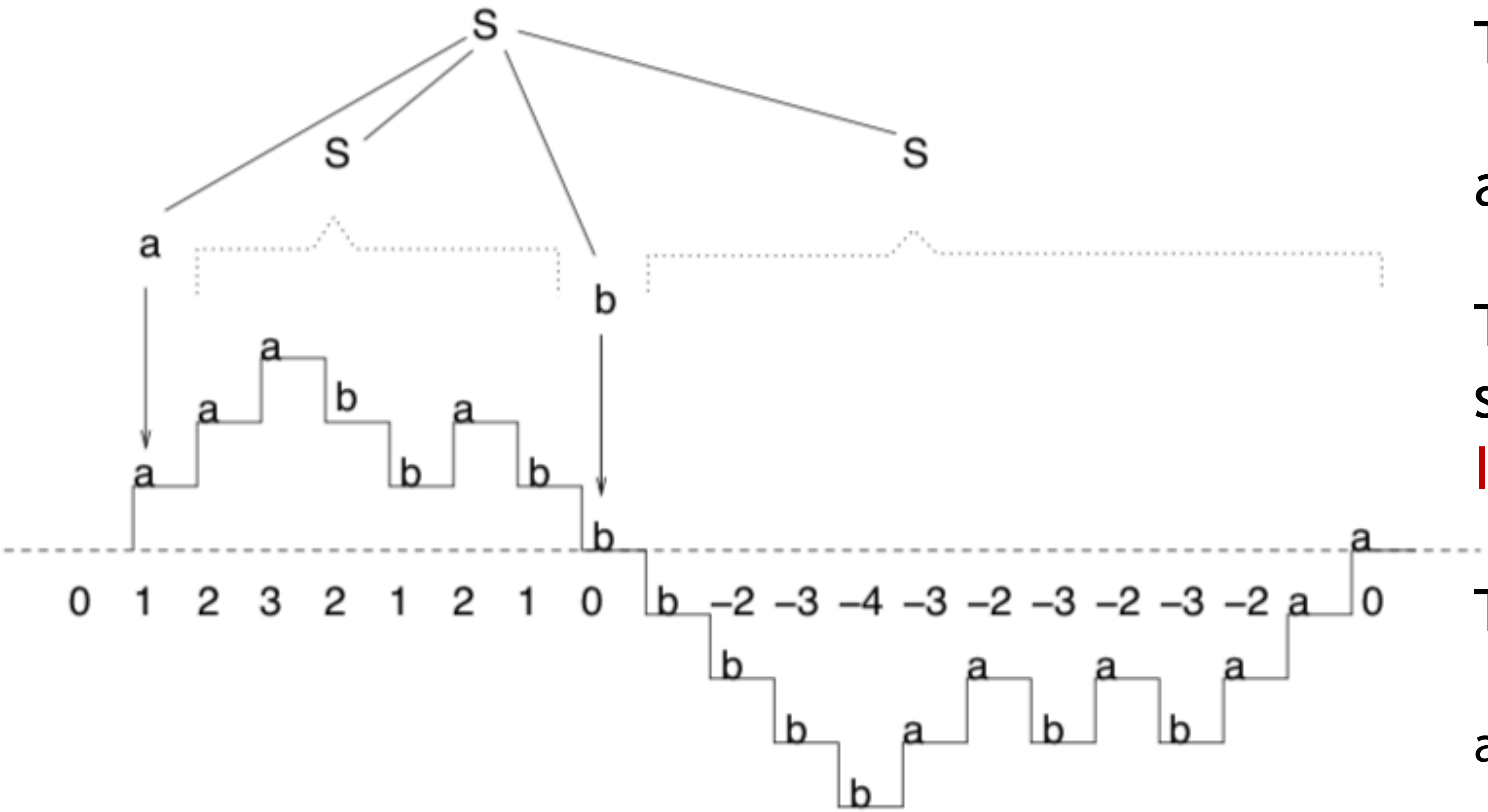
Assumption:

All strings of even length of <= length N are derivable.

The given string that starts and ends with “a” is of the form a …string… a

If we take away the first and last a, then what is left is derivable from S via their own parse trees.
(this is by induction hypothesis)

Arguing Completeness: Hill/Valley Plots



That is, there is a piece

a ...piece in the middle... b

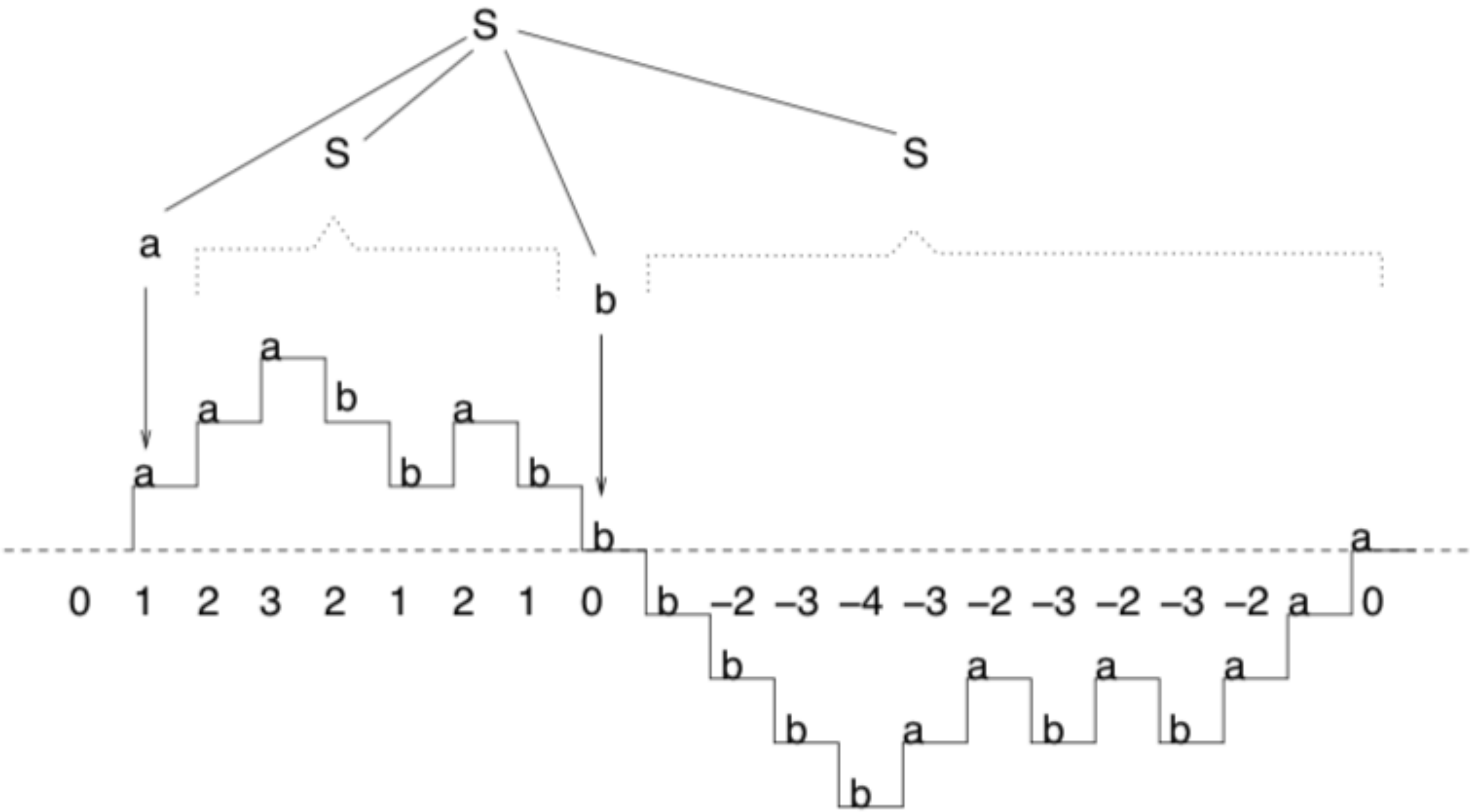
The piece in the middle is a string of length < N.
It is derivable.

Then we have

a ..first piece... b ...second piece...

The second piece is also derivable as it is a shorter string.

Arguing Completeness: Hill/Valley Plots



Thus, the whole string is derivable by a production

a S b S

Thus, all strings that start and end with a are derivable.

Like that, cover all the four cases.

Argue for “starts with a, ends with b”

Another example to test your understanding

- Palindromes
  - We “know” what that set of strings is
  - We must somehow express the strings recursively in such a way that matches the given grammar

- S -> a S a | b S b | M

  M -> ‘’ | a | b

- Strings in the language of S (palindromes) are
  - A string of length 0 or 1 over {a,b}
  - A palindrome with an extra “a” attached at both ends
  - A palindrome with an extra “b” attached at both ends

#1 > #0

- Hint
  - Find a way to plot the “tally” or “hill/valley” plot
  - Dissect the plot recursively into pieces
- You can obtain the grammar by thinking about the completeness proof!

Consistency/Completeness proof for #a = 2 . #b

- Will be in Asg-5

Grammars vs. Ambiguity

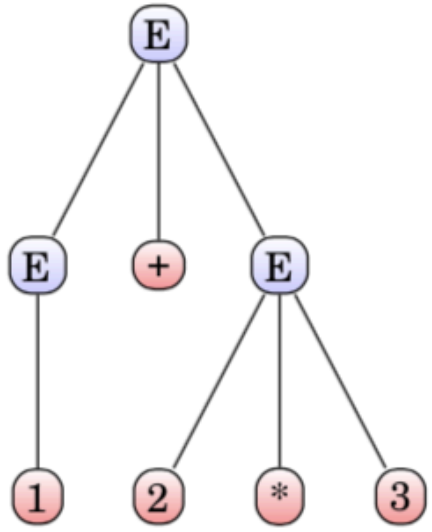
- A grammar G1 may be ambiguous
- Another grammar G2 such that L(G1) = L(G2) may be unambiguous
  - I.e. no string has two distinct parse trees

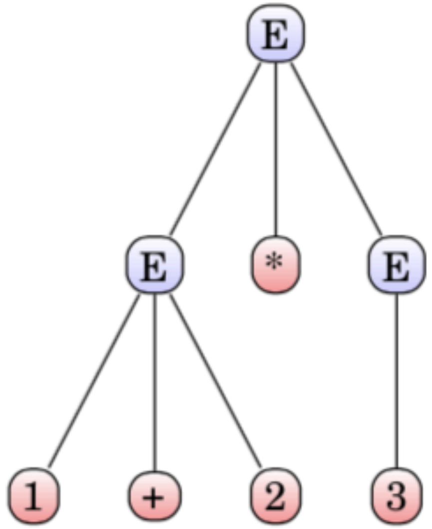
- While L(G1) = L(G2), there is only one parse-tree for L(G2)
- Parse trees determine how
  - A calculator evaluates
  - A compiler generates code
- Let us review the expression grammar (next slide)

Ambiguity and Disambiguation

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

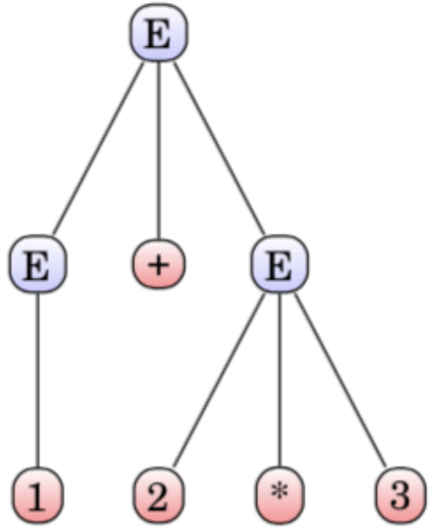
```
E -> E+T | T
T -> T*F | F
F -> 1 | 2 | 3 | ~F | (E)
```





Ambiguity sometimes possible to eliminate

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

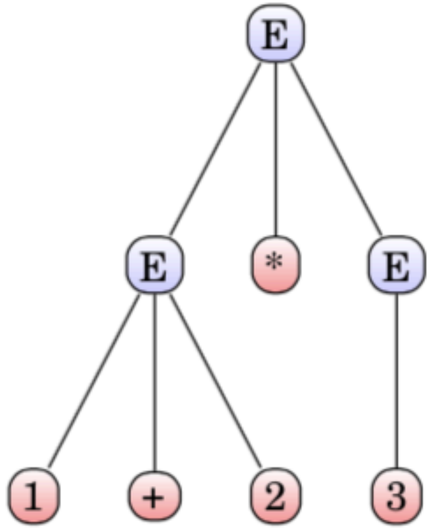



```
E -> E+T | T
T -> T*F | F
F -> 1 | 2 | 3 | ~F | (E)
```

Idea

“Layer the grammar” to capture the precedences correctly.

Often this works!



Gist : by changing the grammar,

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

In general….

- Ambiguity cannot be gotten rid of
  - There are languages for which NO grammar is UNAMBIGUOUS!

- (Later in this course)
  - There is no algorithm to check whether a given CFG is ambiguous!
  - This is harder than “Np-complete etc”.
    - **There isn’t ANY algorithm of ANY complexity whatsoever !!!**

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.

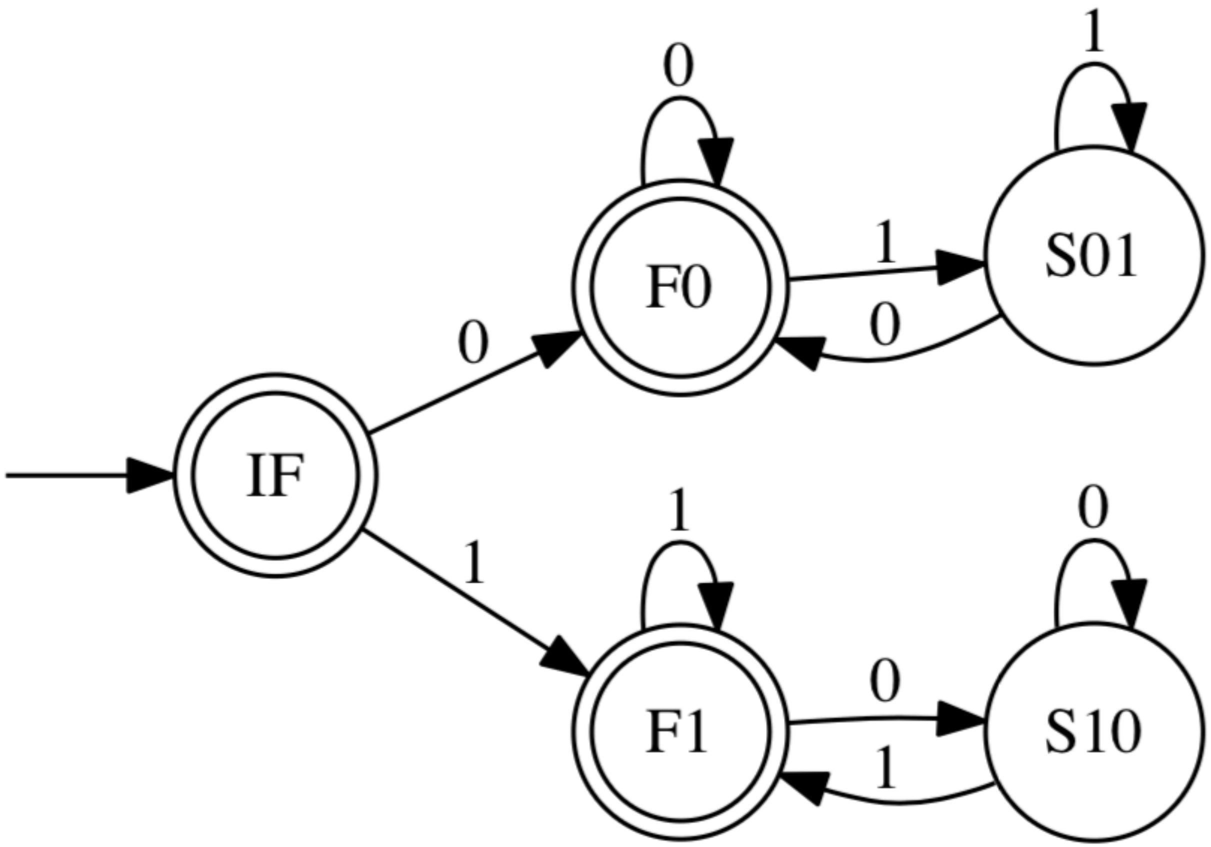
But I can give you papers that cover it (if you wish).

DFA are a special case of CFGs…

- All regular languages are context-free languages also

- Proof is by building a CFG for a given DFA

DFA and CFGs describing them



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

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

DFA via CFG: Purely Right Linear CFGs

```
IF -> ” | 0 F0 | 1 F1
F0 -> ” | 0 F0 | 1 S01
F1 -> ” | 1 F1 | 0 S10
S01 -> 1 S01 | 0 F0
S10 -> 0 S10 | 1 F1
```

The most natural way is to “stare at a DFA” and write down a PURELY RIGHT LINEAR CFG. What is that ?? What is so PURE about it ??

Purely Left Linear CFGs “are reversed DFA”

```
IFr -> ” | F0r 0 | F1r 1
F0r -> ” | F0r 0 | S01r 1
F1r -> ” | F1r 1 | S10r 0
S01r -> S01r 1 | F0r 0
S10r -> S10r 0 | F1r 1
```

Using the “rotating pair of dogs trick”, we can turn a right-linear CFG into a left-linear CFG also!! This proves that even PURELY LEFT LINEAR CFGs denote regular languages. What is PURELY Left Linear ?? What is so pure about it ??

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

Mixed Linearity is NOT Guaranteed Regular!

S -> ” | (S)

If you can express the given language as PURELY left linear, then that language is regular

If you can express the given language as PURELY right linear, then that language is regular

If you expressed a given language using LEFT-LINEAR and RIGHT-LINEAR rules, then… no bets!

Which are CFL and which aren't? (intuitively)

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. $L_{eq012} = \{0^n 1^n 2^n \ : \ n \geq 0\}$

How to prove that a language is NOT a CFL?

- We have a Pumping Lemma for CFLs!
- Used to show that a given language is not context-free
- Usage similar to the regular-language pumping lemma
- The "pump" happens for a different reason
  - **Actually a "parse tree pump"**

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 => ''
```

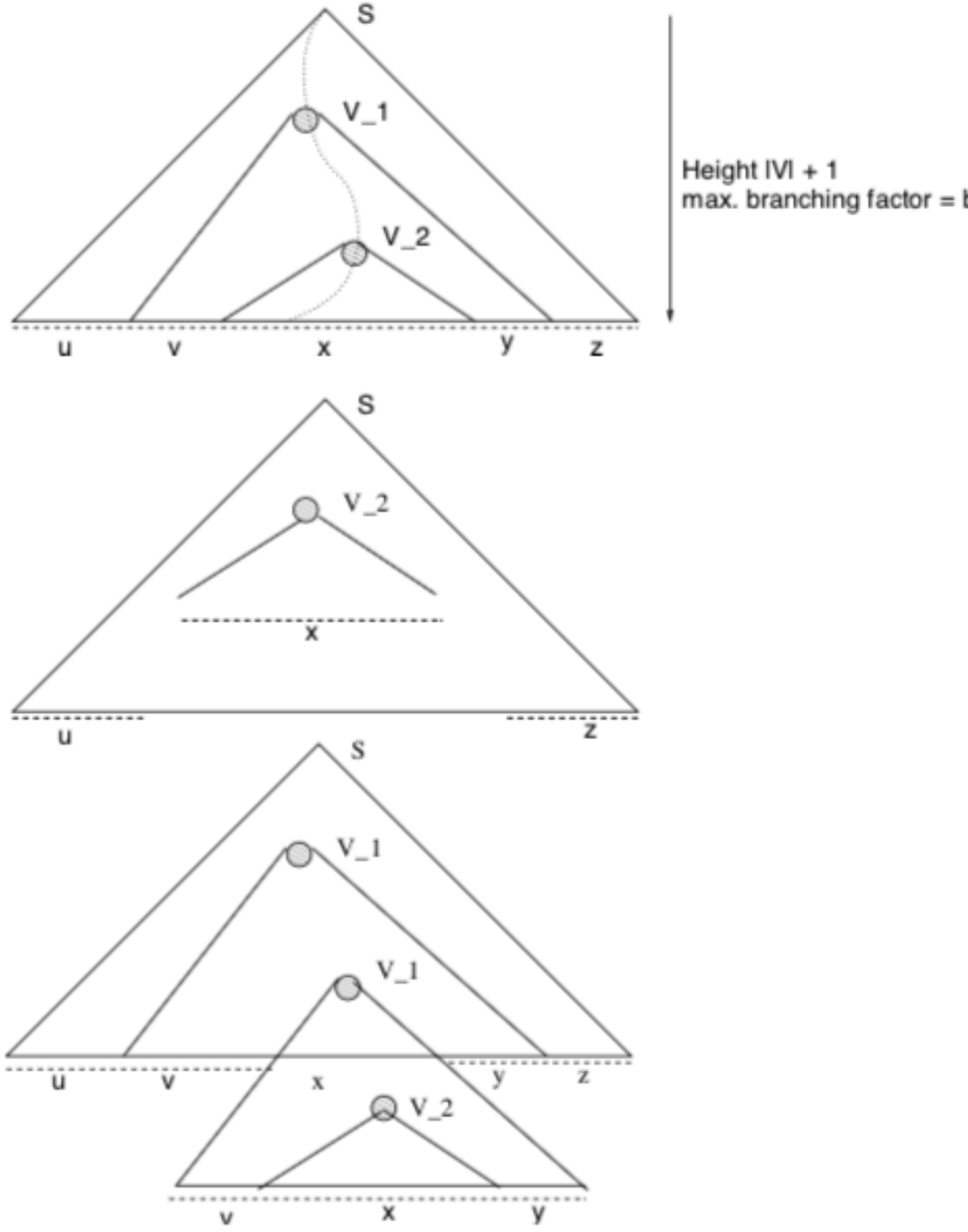
Getting to Pump CFGs: Part 4 of 4

```
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}. $\square$