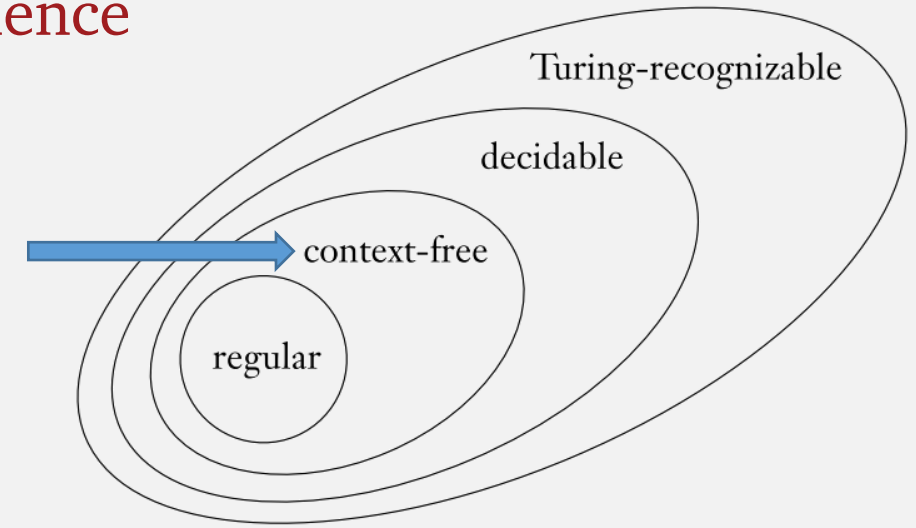
CS 420

Context-Free Grammars (CFGs) and Context-Free Languages (CFLs)

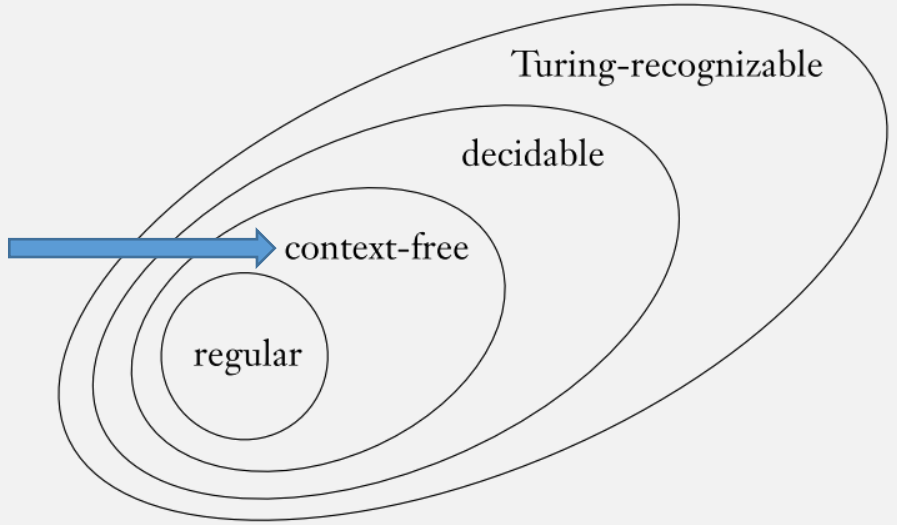
Thursday, October 22, 2026

UMass Boston Computer Science



Announcements

- HW 4 grades out tomorrow
- HW 5 in
  - ~~Due: Thu 10/22 12:30pm~~
- HW 6
  - Out: Thu 10/22
  - Due: Thu 10/29 12:30pm
  - Includes extra credit: must read note



Last Time

The Pumping Lemma …

- … states properties that are True for any Regular Language
- … specifically, properties about "long enough" strings in a Reg Language
- … "long enough" strings must have a repeating pattern (Star Regular Expr)

IMPORTANT:

- ThePumpingLemma cannot prove that a language is **Regular**!
- But … we can use it to prove that a language is NOT **Regular**

… by showing that the repeating pattern is not expressible with a star **Regular** Expression!

IF-THEN statement

… THEN the language is NOT Regular

THEPUMPINGLEMMA IF A IS-A **Regular** Language, THEN p IS-A NatNum (the pumping length) AND IF $s \in A$ AND $|s| \geq p$,THEN s may be divided into three pieces, $s = xyz$, satisfying the following conditions:

1. for each $i \geq 0$, $xy^iz \in A$,
2. $|y| > 0$, and
3. $|xy| \leq p$.

In other words …

s IS-PUMPABLE

Equivalent (**contrapositive**):
IF any of these are NOT True …

Contrapositive:
"IF X THEN Y" is equivalent to "IF **not** Y THEN **not** X"

Kinds of Proofs

- Implication (IF-THEN) `mk-IFTHEN-proof`
  - "assume" IF part, "prove" THEN part
- Using IF-THEN Proof `APPLY <some ifthenproof> TO …`
  - Logically infer (i.e., with modus ponens) conclusion from known definitions and assumptions
- Proof by induction `mk-Inductive-proof`
  - Used to prove properties of recursive definitions or functions
- Proof by contradiction  `mk-Contradiction-proof`
  - Proving the contrapositive

How To Do Proof By Contradiction

3 easy steps:

1. Assume: the opposite of the statement to prove

2. Show: the assumption leads to a Contradiction!

3. Conclude: the original statement must be true

Want to prove: **not** ($\{ 0^n1^n \mid n \geq 0 \}$ IS-A **Regular** Language)

Pumping Lemma: Non-Regularity Example

This **repetition pattern cannot be expressed** with a **Star Regular Expression?**

Let B be the language $\{0^n 1^n \mid n \geq 0\}$. We use THEPUMPINGLEMMA to prove that B is not regular. The proof is by contradiction.

... by showing that the **repeating pattern** is **not expressible with** a **Star regular expression**!

THEPUMPINGLEMMA summary:
"All infinite regular languages must have a **Star** in its **regular expression"**!

Want to prove: **not** ({ $0^n1^n \mid n \geq 0$ } IS-A **Regular** Language)

THEPUMPINGLEMMA IF A IS-A **Regular** Language
THEN p IS-A NatNum "pumping length"
AND
IF $s \in A$ AND $|s| \geq p$ THEN s IS-A Pumpable

Strings longer than "pumping length" are "pumpable"

Assume opposite ...

```
mk-Contradiction-proof
  (with ASSUMPTION = proof of { 0^n1^n | n ≥ 0 } IS-A Regular Language):
```

Statements

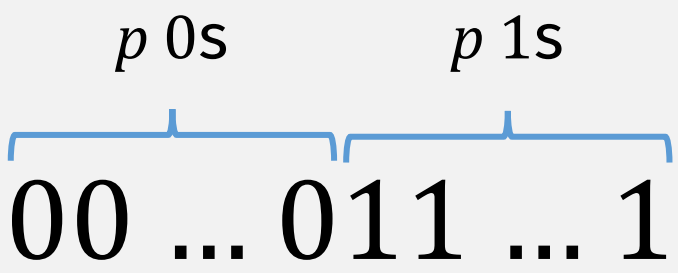
1. p IS-A NatNum
2. IF $s \in \{ 0^n1^n \mid n \geq 0 \}$ AND $|s| \geq p$ THEN s IS-Pumpable

Justifications

i.e., they have a Star repeating pattern

FIRST APPLY THEPUMPINGLEMMA TO ASSUMPTION

SECOND APPLY THEPUMPINGLEMMA TO ASSUMPTION



8. 0^p1^p IS-Pumpable
9. NOT 0^p1^p IS-Pumpable
10. **Contradiction!** → `mk-AND-proof` Proof8, Proof9

Proof needs one Counterexample

... Prove a contradiction

(Justification requires proof of stmt X AND NOT X!)

Want to prove: **not** ({ $0^n1^n \mid n \geq 0$ } IS-A **Regular** Language)

THEPUMPINGLEMMA IF A IS-A **Regular** Language
THEN p IS-A NatNum
AND
IF $s \in A$ AND $|s| \geq p$
THEN s IS-Pumpable

```
mk-Contradiction-proof
  (with ASSUMPTION = proof of { 0^n1^n | n ≥ 0 } IS-A Regular Language):
```

	Statements	Justifications
1.	p IS-A NatNum	**FIRST APPLY** THEPUMPINGLEMMA **TO** ASSUMPTION
2.	IF $s \in \{ 0^n1^n \mid n \geq 0 \}$ AND $\|s\| \geq p$ THEN s IS-Pumpable	**SECOND APPLY** THEPUMPINGLEMMA **TO** ASSUMPTION
3.	$p \geq 0$	Proof1
4.	$0^p1^p \in \{ 0^n1^n \mid n \geq 0 \}$	Proof3?
5.	$\| 0^p1^p \| = 2p$	**BY** DefLength
6.	$2p \geq p$	**BY** Definition of NatNums
7.	$\| 0^p1^p \| \geq p$	**SUBST** Proof5 **INTO** Proof6
8.	0^p1^p IS-Pumpable	**APPLY** Proof2 **TO** `mk-AND-proof` Proof4 Proof7
9.	NOT 0^p1^p IS-Pumpable	???
10.	**Contradiction!**	`mk-AND-proof` Proof8, Proof9

To make proofs more understandable, will begin to allow combining less imporant steps

Want to prove: **not** ({ $0^n1^n \mid n \geq 0$ } IS-A **Regular** Language)

THEPUMPINGLEMMA IF A IS-A **Regular** Language
THEN p IS-A NatNum
AND
IF $s \in A$ AND $|s| \geq p$
THEN s IS-Pumpable

`mk-Contradiction-proof`
`(with` ASSUMPTION = proof of { $0^n1^n \mid n \geq 0$ } IS-A **Regular** Language):

	Statements	Justifications
1.	p IS-A NatNum	**`FIRST APPLY`** THEPUMPINGLEMMA **`TO`** ASSUMPTION
2.	IF $s \in \{ 0^n1^n \mid n \geq 0 \}$ AND $\|s\| \geq p$ THEN s IS-Pumpable	**`SECOND APPLY`** THEPUMPINGLEMMA **`TO`** ASSUMPTION
3.	$0^p1^p \in \{ 0^n1^n \mid n \geq 0 \}$	**BY** Proof1, Definition of Sets
4.	$\| 0^p1^p \| \geq p$	**BY** DefLength, Definition of **NatNums**
8.	0^p1^p IS-Pumpable	**`APPLY`** Proof2 **`TO mk-AND-proof`** Proof4 Proof7
9.	NOT 0^p1^p IS-Pumpable	???
10.	**Contradiction!**	**`mk-AND-proof`** Proof8, Proof9

Want to prove: **not** ({ 0^n1^n | $n \geq 0$ } IS-A **Regular** Language)

THEPUMPINGLEMMA IF A IS-A **Regular** Language
THEN p IS-A NatNum
AND
IF $s \in A$ AND $|s| \geq p$
THEN s IS-Pumpable

mk-Contradiction-proof
(with ASSUMPTION = proof of { 0^n1^n | $n \geq 0$ } IS-A **Regular** Language):

	Statements	Justifications
1.	p IS-A NatNum	**FIRST APPLY** THEPUMPINGLEMMA **TO** ASSUMPTION
2.	IF $s \in$ { 0^n1^n | $n \geq 0$ } AND $\|s\| \geq p$ THEN s IS-Pumpable	**SECOND APPLY** THEPUMPINGLEMMA **TO** ASSUMPTION
3.	$0^p1^p \in$ { 0^n1^n | $n \geq 0$ }	**BY** Proof1, Definition of Sets
4.	$\| 0^p1^p \| \geq p$	**BY** DefLength, Definition of **NatNums**
5.	0^p1^p IS-Pumpable	**APPLY** Proof2 **TO** **mk-AND-proof** Proof3 Proof4
6.	NOT 0^p1^p IS-Pumpable	??? **BY** *NotPumpableAnalysis* **ON** - Pumping Length: p - Language: { 0^n1^n | $n \geq 0$ } - Counterexample String: 0^p1^p
7.	**Contradiction!**	**mk-AND-proof** Proof5, Proof6

Proof needs to show Counterexample is not pumpable

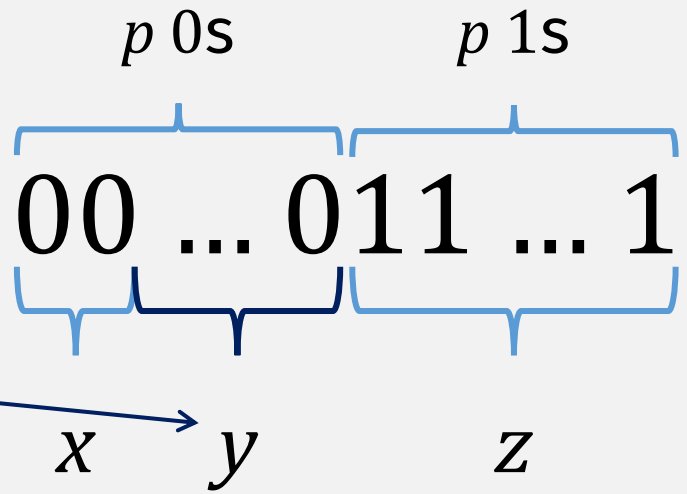
Requires 3 arguments: **counterexample string, Lang, and pumping length p**

Counterexample strings not in lang or shorter than p not allowed!

let *NotPumpableAnalysis* = **`mk-NotPumpable-proof`** (with Pumping Length: p
Language: L
Counterexample String: s
where
$s \in L$ AND $|s| \geq p$):

NEED TO PROVE:
there is **no split** into xyz where
- y **can be repeated**
- and **pumped string still in language!**

Counterexample = 0^p1^p



Split 1
- Choose xyz split so y contains:
  - all 0s

BUT ... THEPUMPINGLEMMA requires **only one** pumpable splitting

So the proof is not done!

Is there another way to split into xyz ?

- Pumping y: produces a string with more 0s than 1s ...
  - ... which is not in the language $\{ 0^n1^n \mid n \geq 0 \}$!
  - So 0^p1^p is not pumpable? (not yet!)

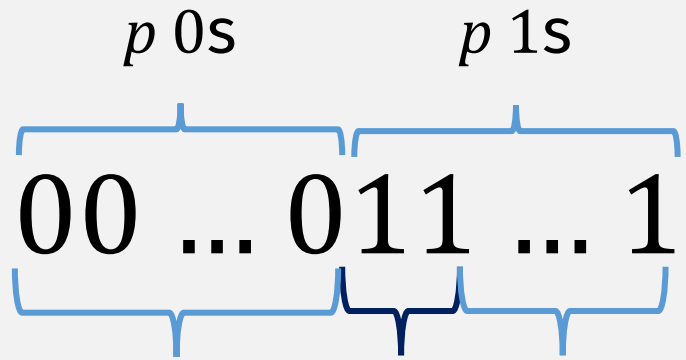
NEED TO PROVE:
s IS-NOT Pumpable

s may be divided into three pieces, $s = xyz$, satisfying the following conditions:

1. for each $i \geq 0$, $xy^iz \in A$,
2. $|y| > 0$, and
3. $|xy| \leq p$.

s IS-PUMPABLE

Counterexample = 0^p1^p



Split 2

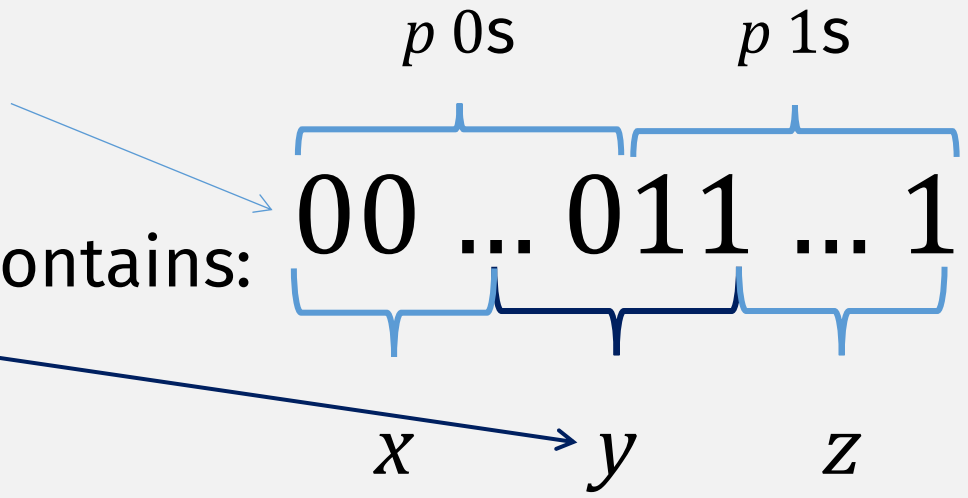
- Choose xyz split so y contains:
  - all 1s

x y z

Is there another way to split into xyz ?

- Is this string pumpable (repeating y produces string still in 0^n1^n)?
  - No!
  - By the same reasoning as in the previous slide

Counterexample = 0^p1^p



Split 3

- Choose xyz split so y contains:
  - both 0s and 1s

- Is this string pumpable (repeating y produces string still in 0^n1^n)?
  - No!
  - Pumped string will have equal 0s and 1s …
  - But they will be in the wrong order

Did we examine every possible splitting?

Yes! So we proved 0^p1^p IS-NOT Pumpable!

But maybe we didn't have to look at so many cases …

The Pumping Lemma: Condition 3

THEPUMPINGLEMMA IF A IS-A **Regular** Language, THEN p IS-A NatNum (the pumping length) AND IF $s \in A$ AND $|s| \geq p$,THEN s may be divided into three pieces, $s = xyz$, satisfying the following conditions:

1. for each $i \geq 0$, $xy^iz \in A$,
2. $|y| > 0$, and
3. $|xy| \leq p$.

In other words ...

s IS-PUMPABLE

The repeating part y ...
must be in the first p characters!

Split 1

p 0s

00 ... 011 ... 1

(this is the only case we needed to consider!)

y must be in here!

The Pumping Lemma: Pumping Down

THEPUMPINGLEMMA IF A IS-A **Regular** Language, THEN p IS-A NatNum (the pumping length) AND IF $s \in A$ AND $|s| \geq p$,THEN s may be divided into three pieces, $s = xyz$, satisfying the following conditions:

1. for each $i \geq 0$, $xy^iz \in A$,
2. $|y| > 0$, and
3. $|xy| \leq p$.

In other words ...

s IS-PUMPABLE

Repeating part y must be non-empty ...
but can be repeated zero times!

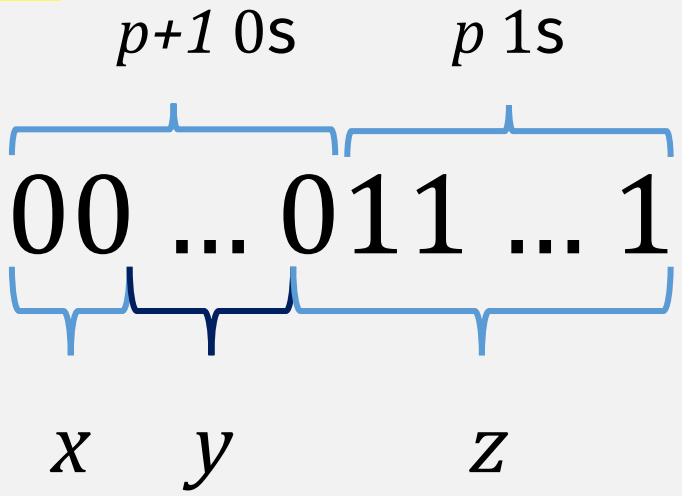
Example: $L = \{0^i1^j \mid i > j\}$

Want to prove: $L = \{0^i 1^j \mid i > j\}$ IS-NOT-A **Regular** Language

Pumping Down

Proof sketch (by contradiction):

- Assume: L IS-A **Regular** Language
  - So **it must satisfy** THEPUMPINGLEMMA
  - I.e., **all strings ≥ length p are pumpable**
- Counterexample = $0^{p+1}1^p$



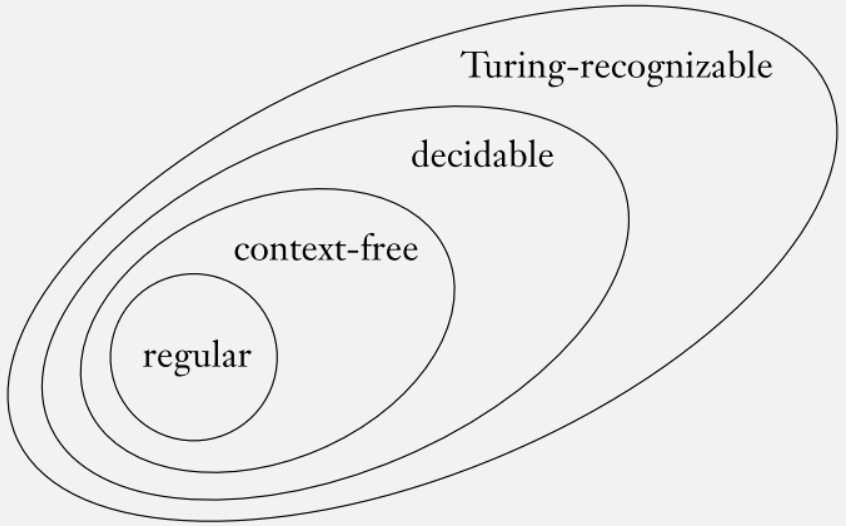
- Choose xyz split so y contains:
  - all 0s
  - (Only possibility, by condition 3)

- Repeat y zero times (**pump down**): produces string with # 0s ≤ # 1s
  - … not in the language $\{0^i 1^j \mid i > j\}$
  - So $\{0^i 1^j \mid i > j\}$ does not satisfy THEPUMPINGLEMMA
  - But **Regular** Languages must be pumpable!
  - This is a **contradiction**!

Road Map for Remainder of Semester

- If a language is not **Regular**, then what is it?

- There are many more classes of languages!



Last Time:

Non-Regular Languages

Example:

An arbitrary count

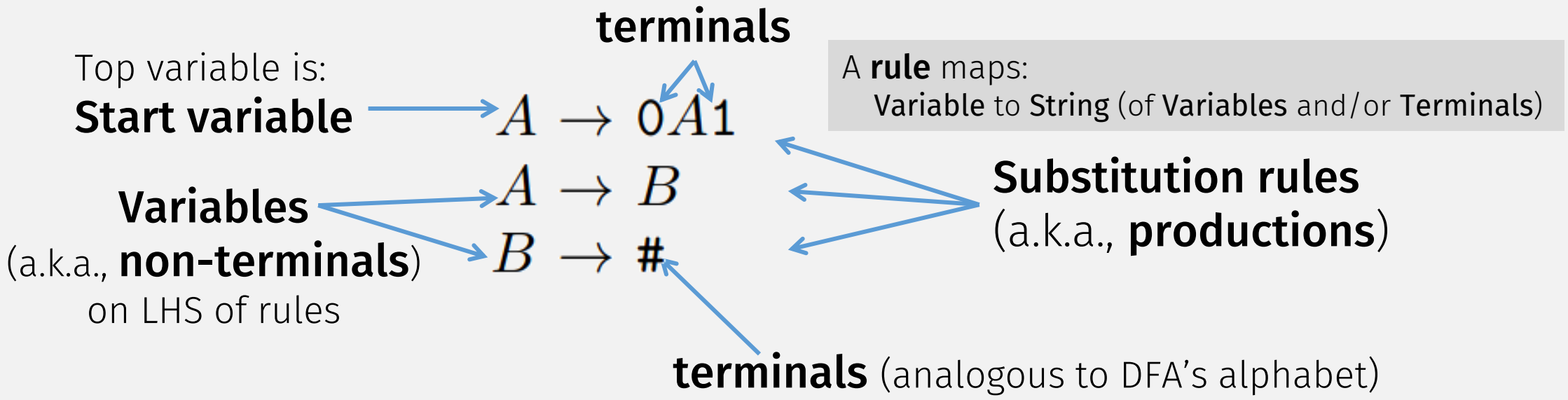
$$L = \{ \mathtt{0}^n\mathtt{1}^n \mid n \geq 0 \}$$

- A DFA recognizing L would **require infinite states!** (impossible)
  - States representing: **zero `0`s seen, one `0` seen, two `0`s, …**
- **This language is the same as many PLs,** e.g., **HTML!**
  - To better see this replace:
    - "`0`" with "`<tag>`" or "`(`"
    - "`1`" with "`</tag>`" or "`)`"
- The Problem: **remembering nestedness**
  - **Need to count arbitrary nesting depths**
    - E.g., `{ { { … } } }`
  - Thus: **most programming language syntax** is **not regular!**

We can *prove* **non-regularness** … with the **Pumping Lemma** (and **proof by contradiction**)

But … **what kind of language is it then?**

A Context-Free Grammar (CFG)



Context-Free Grammar (CFG): Formally

let G_1 = **mk-CFG** (V, Σ, R, S) …

R is this set of rules: Var→String (of vars+terminals) mappings:

Top variable is:
Start variable

$$A \to 0A1$$

Variables
(a.k.a., **non-terminals**)

$$A \to B$$

$$B \to \#$$

Substitution rules
(a.k.a., **productions**)

terminals (analogous to DFA's alphabet)

CFG Practical Application:
Used to **describe programming language syntax**!

A ***context-free grammar*** is a `mk-CFG` (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.

where:

$V = \{A, B\}$,

$\Sigma = \{0, 1, \#\}$,

$S = A$,

Java Syntax: Described with CFGs

Definition:
A **CFG** describes a **context-free language!**

ORACLE

Java SE > Java SE Specifications > Java Language Specification

Chapter 2

Prev

Chapter 2. Grammars

This chapter describes the context-free grammars used in this specification to define the lexical and syntactic structure of a program

2.1. Context-Free Grammars

"productions" = rules

"nonterminal" = variable

A *context-free grammar* consists of a number of *productions*. Each production has an abstract symbol called a *nonterminal* as its *left hand side*, and a sequence of one or more nonterminal and *terminal* symbols as its *right-hand side*. For each grammar, the terminal symbols are drawn from a specified *alphabet*.

"goal symbol" = Start variable

A **CFG** specifies a language!

Starting from a sentence consisting of a single distinguished nonterminal, called the *goal symbol*, a given context-free grammar specifies a language, namely, the set of possible sequences of terminal symbols that can result from repeatedly replacing any nonterminal in the sequence with a right-hand side of a production for which the nonterminal is the left-hand side.

(definition of a **language:** set of sequences of symbols)

2.2. The Lexical Grammar

A *lexical grammar* for the Java programming language is given in §3. This grammar has as its terminal symbols the characters of the Unicode character set. It defines a set of productions, starting from the goal symbol *Input* (§3.5), that describe how sequences of Unicode characters (§3.1) are translated into a sequence of input elements (§3.5)

https://docs.oracle.com/javase/specs/jls/se7/html/jls-2.html

Definition:
A **CFG** describes a
context-free language!

DefCFL

(some defined CFG)

IF G IS-A **Context-free** Grammar
AND
G **describes** L (some defined language)
THEN L IS-A **Context-free** Language

(abbrev)

DefCFL

IF G IS-A **CFG**
AND
G **describes** L
THEN L IS-A **CFL**

Analogies

Regular Language	Context-Free Language (CFL)
Regular Expression	Context-Free Grammar (CFG)
A **Reg Expr** describes a **Regular lang**	A **CFG** describes a **CFL**

thm

def

(partially)

Python Syntax: Described with a CFG

10. Full Grammar specification

This is the full Python grammar, as it is read by the parser generator and used to parse Python source files:

```
# Grammar for Python

# NOTE WELL: You should also follow all the steps listed at
# https://devguide.python.org/grammar/

# Start symbols for the grammar:
#       single_input is a single interactive statement;
#       file_input is a module or sequence of commands read from an input file;
#       eval_input is the input for the eval() functions.
#       func_type_input is a PEP 484 Python 2 function type comment
# NB: compound_stmt in single_input is followed by extra NEWLINE!
# NB: due to the way TYPE_COMMENT is tokenized it will always be followed by a NEWLINE
single_input: NEWLINE | simple_stmt | compound_stmt NEWLINE
file_input: (NEWLINE | stmt)* ENDMARKER
eval_input: testlist NEWLINE* ENDMARKER
```

(indentation checking not expressible with CFG?)

https://docs.python.org/3/reference/grammar.html

Many Other Language (partially)
~~Python~~ Syntax: Described with a CFG

10. Full Grammar specification

This is the full Python grammar, as it is read by the parser generator and used to parse Python source files:

```
# Grammar for Python

# NOTE WELL: You should also follow all the steps listed at
# https://devguide.python.org/grammar/

# Start symbols for the grammar:
#       single_input is a single interactive statement;
#       file_input is a module or sequence of commands read from an input file;
#       eval_input is the input for the eval() functions.
#       func_type_input is a PEP 484 Python 2 function type comment
# NB: compound_stmt in single_input is followed by extra NEWLINE!
# NB: due to the way TYPE_COMMENT is tokenized it will always be followed by a NEWLINE
single_input: NEWLINE | simple_stmt | compound_stmt NEWLINE
file_input: (NEWLINE | stmt)* ENDMARKER
eval_input: testlist NEWLINE* ENDMARKER
```

https://docs.python.org/3/reference/grammar.html

Java Syntax: Described with CFGs

ORACLE

Java SE > Java SE Specifications > Java Language Specification

Prev

Chapter 2. Grammars

This chapter describes the context-free grammars used in this specification to define the lexical and syntactic structure of a program

2.1. Context-Free Grammars

A *context-free grammar* consists of a number of *productions*. Each production has an abstract symbol called a *nonterminal* as its *left hand side*, and a sequence of one or more nonterminal and *terminal* symbols as its *right-hand side*. For each grammar, the terminal symbols are drawn from a specified *alphabet*.

Starting from a sentence consisting of a single distinguished nonterminal, called the *goal symbol*, a given context-free grammar specifies a language, namely, the set of possible sequences of terminal symbols that can result from repeatedly replacing any nonterminal in the sequence with a right-hand side of a production for which the nonterminal is the left-hand side.

2.2. The Lexical Grammar

A *lexical grammar* for the Java programming language is given in §3. This grammar has as its terminal symbols the characters of the Unicode character set. It defines a set of productions, starting from the goal symbol *Input* (§3.5), that describe how sequences of Unicode characters (§3.1) are translated into a sequence of input elements (§3.5)

Definition:
A **CFG** describes a **context-free language!**
but **what strings** are **in the language?**

https://docs.oracle.com/javase/specs/jls/se7/html/jls-2.html

Generating Strings with a CFG

Called "**context free**" because: **we replace vars** with **RHS of rule,** without **looking at surrounding chars**, i.e., **context**

In-class exercise:

Write 3 more strings that can be generated by this grammar

Definition:
A **CFG** describes a **context-free language!**
but **what** strings are **in the language?**

1st rule → $A \to 0A1$

2nd rule → $A \to B$

Last rule → $B \to \#$

"Applying a rule"
= replace LHS variable with RHS sequence

At each step, *arbitrarily* choose any variable to replace, and any rule to apply

Stop when: **string is all terminals**

A CFG **generates** a string, by repeatedly applying substitution rules:

Example: $A \Rightarrow 0A1 \Rightarrow 00A11 \Rightarrow 000A111 \Rightarrow 000B111 \Rightarrow 000\#111$

Start with: Start variable

Apply 1st rule

1st rule again

1st rule again

Apply 2nd rule

Apply last rule

Generating Strings with a CFG

Definition:
A **CFG** describes a **context-free language!**
but **what** strings are **in the language?**

let $G_1 =$

$$A \to 0A1$$
$$A \to B$$
$$B \to \#$$

Strings in CFG's language
= all possible **generated** / **derived** strings

$$\text{LANGOF}(G_1) = \{0^n\#1^n \mid n \geq 0\}$$

A CFG **generates** a string, by repeatedly applying substitution rules:

Example: $A \Rightarrow 0A1 \Rightarrow 00A11 \Rightarrow 000A111 \Rightarrow 000B111 \Rightarrow 000\#111$

This sequence of steps is called a **derivation**

Derivations: Formally

A ***context-free grammar*** is a `mk-CFG` (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.

let G = **`mk-CFG`** (V, Σ, R, S) ...

Single-step

$$\alpha A \beta \underset{G}{\Rightarrow} \alpha \gamma \beta$$

Where: sequence of: variables or terminals

$$\alpha, \beta \in (V \cup \Sigma)^*$$

$A \in V$ ← Variable

$A \to \gamma \in R$ ← Rule

Derivations: Formally

A ***context-free grammar*** is a `mk-CFG` (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.

let G = **`mk-CFG`** (V, Σ, R, S) ...

Single-step

$$\alpha A \beta \underset{G}{\Rightarrow} \alpha \gamma \beta$$

Where:

$$\alpha, \beta \in (V \cup \Sigma)^*$$

$$A \in V \leftarrow \text{Variable}$$

$$A \to \gamma \in R \leftarrow \text{Rule}$$

Multi-step (recursively defined, on # steps)

Base case: $\alpha \overset{*}{\underset{G}{\Rightarrow}} \alpha$ (0 steps)

Recursive case: (1 or more steps)

Single step

- If: $\alpha \underset{G}{\Rightarrow} \beta$ and $\beta \overset{*}{\underset{G}{\Rightarrow}} \gamma$

(smaller) Recursive "call"

- Then: $\alpha \overset{*}{\underset{G}{\Rightarrow}} \gamma$

Formal Definition of a CFL

A ***context-free grammar*** is a `mk-CFG` (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.

let G = **`mk-CFG`** (V, Σ, R, S) ...

"all possible sequences of terminal symbols ..."

... "that can be **generated** with rules of grammar G"

"the language of a grammar G is ..."

$$\textsc{LangOf}(G) = \left\{ w \in \Sigma^* \mid S \overset{*}{\underset{G}{\Rightarrow}} w \right\}$$

Flashback: $\{0^n 1^n \mid n \geq 0\}$

- **Pumping Lemma** says: **not a regular language**
- It's a **context-free language!**
  - **Proof** (sketch)**?**
  - Key step: **Come up with CFG describing it …**
  - Hint: It's similar to:

$$A \to 0A1$$
$$A \to B$$
$$B \to \#$$

ε (empty string)

$\textsc{LangOf}(G_1) = \{0^n \# 1^n \mid n \geq 0\}$

G_2

Statements and Justifications?

Proof: $L = \{0^n 1^n \mid n \geq 0\}$ IS-A **CFL**

let $G_2 =$ $A \to 0A1$, $A \to B$, $B \to \varepsilon$

Statements

1. G_2 IS-A **CFG**
2. G_2 **describes** L
3. $\{0^n 1^n \mid n \geq 0\}$ IS-A **CFL**

Justifications

1. **BY** DefCFGArrowRules (analogous to DefDFAStateDiagram)
2. **BY** ExTable1! **COMPARING** G_2, L
3. **APPLY** DefCFL **TO** `mk-AND-proof` Proof1, Proof2

DefCFL

IF G IS-A **CFG** AND G **describes** L
THEN L IS-A **CFL**

CFG Examples Table

$L = \{0^n 1^n \mid n \geq 0\}$ IS-A **CFL**

let $G_2 =$ $A \to 0A1$, $A \to B$, $B \to \varepsilon$

let ExTable1! =

String s	$s \in L$?	~~DFA accepts s?~~	CFG G_2 generates s?

"|" symbol = Shorthand for multiple rules with same LHS variable

A String Can Have Multiple Derivations

$$\langle \text{EXPR} \rangle \rightarrow \langle \text{EXPR} \rangle + \langle \text{TERM} \rangle \mid \langle \text{TERM} \rangle$$
$$\langle \text{TERM} \rangle \rightarrow \langle \text{TERM} \rangle \times \langle \text{FACTOR} \rangle \mid \langle \text{FACTOR} \rangle$$
$$\langle \text{FACTOR} \rangle \rightarrow (\langle \text{EXPR} \rangle) \mid \texttt{a}$$

Want to generate this string: **a + a × a**

- EXPR ⇒
- EXPR + TERM ⇒
- EXPR + TERM × FACTOR ⇒
- EXPR + TERM × a ⇒

...

RIGHTMOST DERIVATION

- EXPR ⇒
- EXPR + TERM ⇒
- TERM + TERM ⇒
- FACTOR + TERM ⇒
- **a** + TERM

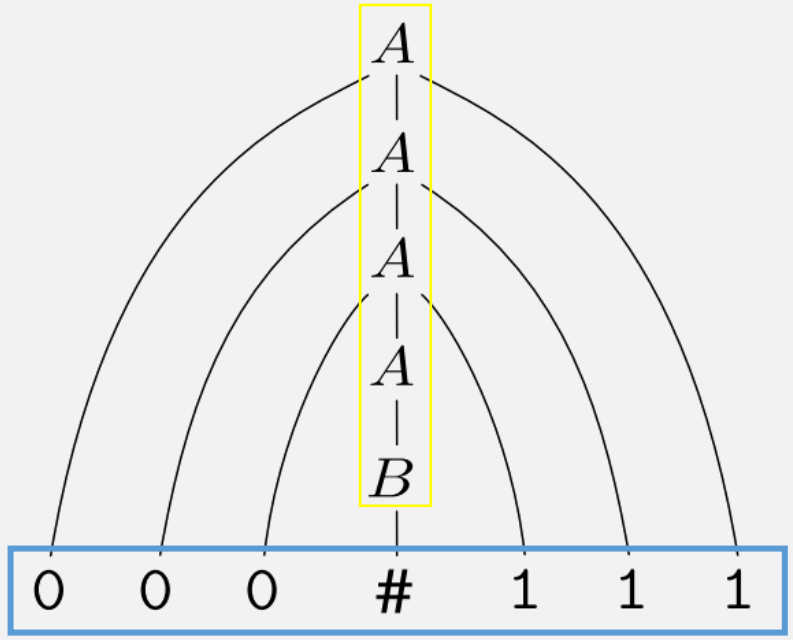
...

LEFTMOST DERIVATION

Derivations and Parse Trees

$$A \Rightarrow 0A1 \Rightarrow 00A11 \Rightarrow 000A111 \Rightarrow 000B111 \Rightarrow 000\#111$$

A derivation may also be represented as a **parse tree**



Multiple Derivations, Single Parse Tree

Leftmost deriviation

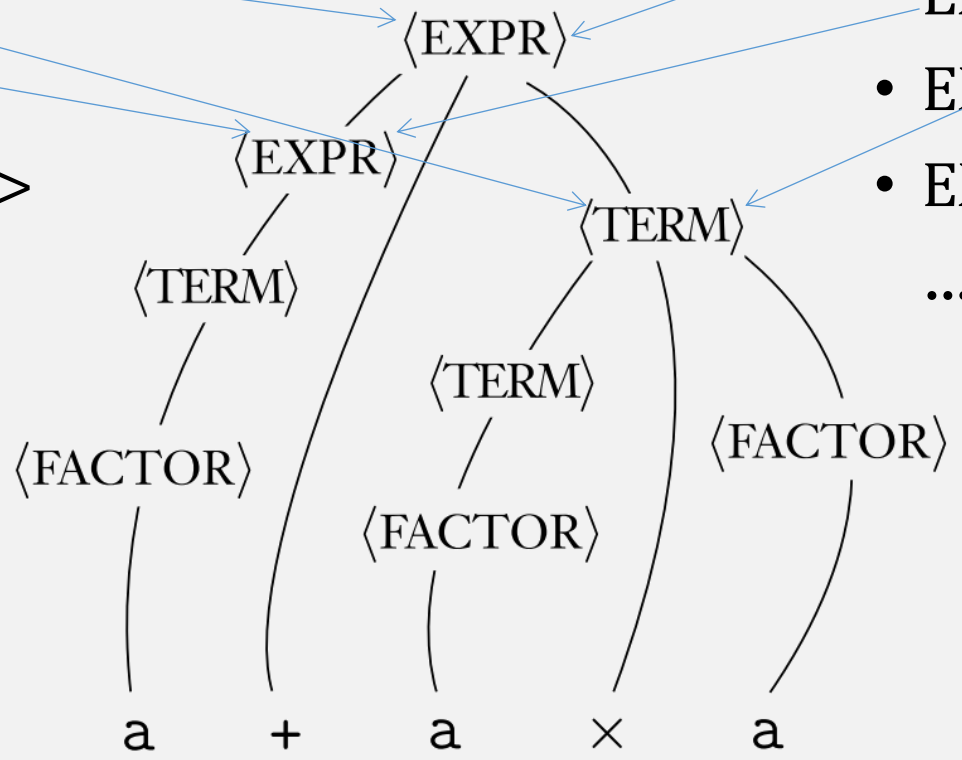
- EXPR =>
- EXPR + TERM =>
- TERM + TERM =>
- FACTOR + TERM =>
- a + TERM

...

Rightmost deriviation

- EXPR =>
- EXPR + TERM =>
- EXPR + TERM x FACTOR =>
- EXPR + TERM x a=>

...



Same parse tree

A **parse tree** represents a CFG computation ... like a **sequence of states** represents a DFA computation

A **Parse Tree** gives "meaning" to a **string**

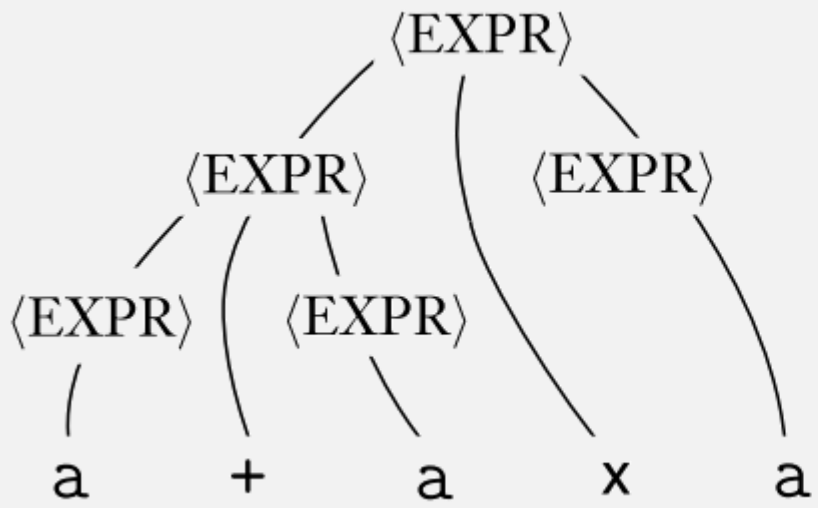
Ambiguity

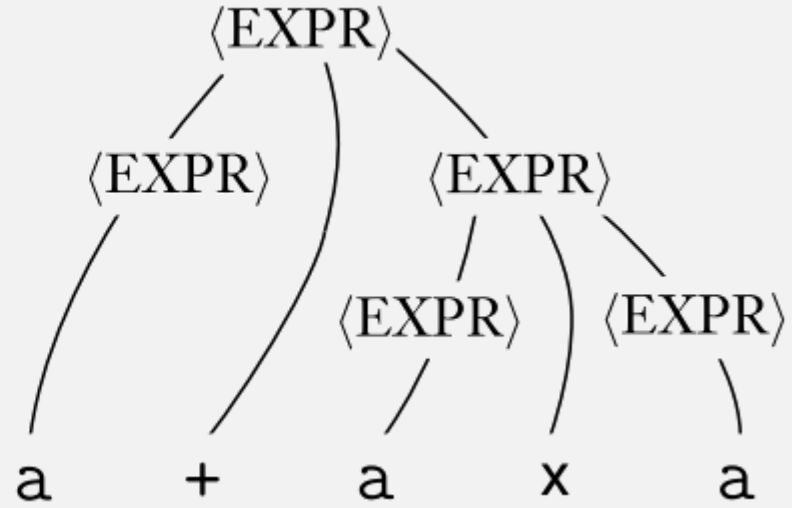
grammar G_5:

$$\langle \text{EXPR} \rangle \rightarrow \langle \text{EXPR} \rangle + \langle \text{EXPR} \rangle \mid \langle \text{EXPR} \rangle \text{x} \langle \text{EXPR} \rangle \mid (\langle \text{EXPR} \rangle) \mid \text{a}$$

Same **string,**
different **derivation,**
and different **parse tree!**

So this string has
two meanings!





Ambiguity

A string w is derived ***ambiguously*** in context-free grammar G if it has two or more different leftmost derivations. Grammar G is ***ambiguous*** if it generates some string ambiguously.

An **ambiguous grammar** can give a **string multiple meanings**, ie **represent two different computations!**
(why is this **bad**?)

6.8.4.1(3): "An **`else`** is associated with the lexically nearest preceding **`if`** that is allowed by the syntax." https://www.open-std.org/jtc1/sc22/wg14/www/standards

Real-life Ambiguity ("Dangling" `else`)

- What is the result of this C program?

```
if (1) if (0) printf("a"); else printf("2");
```



```
if (1)
  if (0)
    printf("a");
  else
    printf("2");
```

vs

```
if (1)
  if (0)
    printf("a");
else
  printf("2");
```

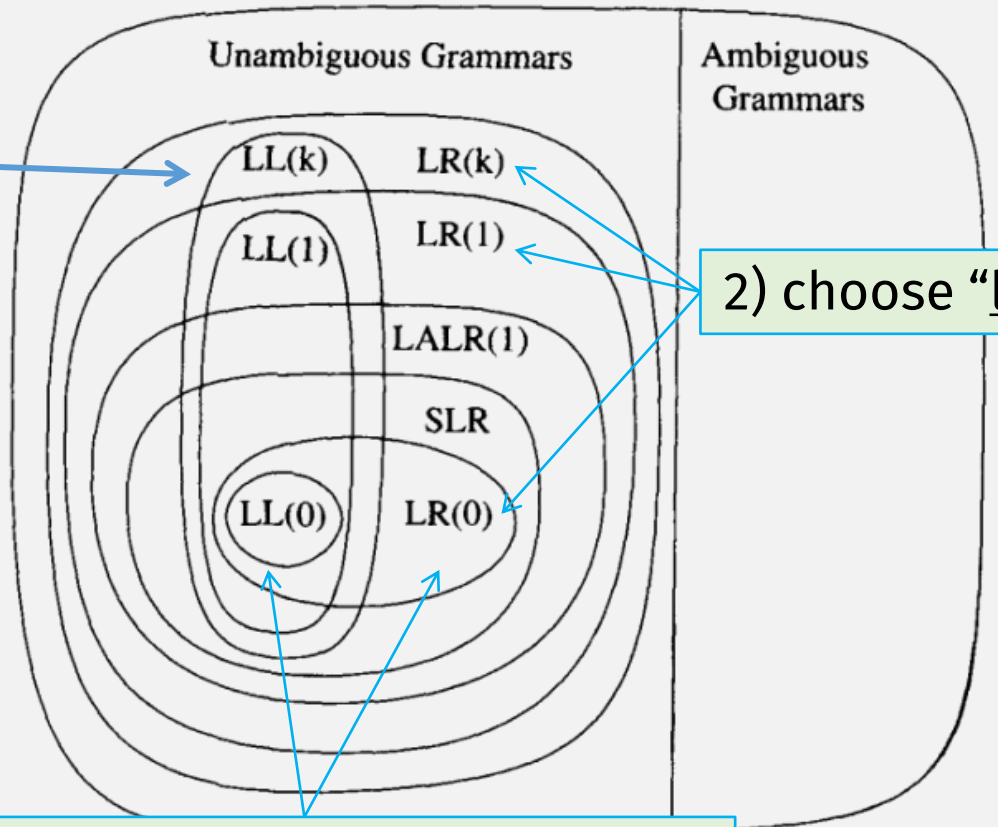
This **string has 2 parsings**, and **thus 2 meanings**!

Ambiguous grammars are confusing. A computation (represented by a string) **should ideally have only one possible result**.

Thus in practice, we typically focus on the **unambiguous** subset of CFGs (CFLs) (more on this later)

Problem is, there's no easy way to create an **unambiguous** grammar (it's up to language designers to "be careful")

Subclasses of CFLs



DCFLs

Programming language parsers / compilers are ideally in here

2) choose "look ahead" amount

2 parser design decisions:
1) Parse from left (L),
or from right (R)

All CFLS

Designing Grammars : Basics

1. **Think about what you want to "link" together**

- E.g., 0^n1^n
  - $A \rightarrow 0A1$
  - # 0s and # 1s are **"linked"**

- E.g., **HTML**
  - $\text{ELEMENT} \rightarrow \text{<TAG>CONTENT</TAG>}$
  - **Start** and **end tags** are **"linked"**

2. **Start** with **small grammars** (computation) and **then combine**
   - just like with **DFA**s, **NFA**s, and **programming**!

In-class exercise: Creating CFG

alphabet Σ is $\{0,1\}$

$$L = \{w \mid w \text{ starts and ends with the same symbol}\}$$

1) come up with examples: In the language: **010, 101, 110001** **1, 0 ?** ☑

Not in the language: **10, 01, 110** **ε ?** ☒

2) Create CFG:

Needed Rules:

$S \rightarrow 0M0 \mid 1M1 \mid 0 \mid 1$ "start/end symbol are "linked" (ie, same); middle can be anything"

$M \rightarrow MT \mid \varepsilon$ "middle: all possible terminals, repeated (ie, all possible strings)"

$T \rightarrow 0 \mid 1$ "all possible terminals"

3) Check CFG: generates examples in the language; doesn't generate examples not in language

Examples Table! (justifies that the "CFG describes L")

Designing Grammars: Building Up

- **Start** with **small grammars** and then **combine** (just like **programming**)
  - To create a **grammar for the language** $\{0^n 1^n | n \geq 0\} \cup \{1^n 0^n | n \geq 0\}$
  - First create **grammar for lang** $\{0^n 1^n | n \geq 0\}$:

$$S_1 \rightarrow 0S_1 1 \mid \varepsilon$$

  - Then create **grammar for lang** $\{1^n 0^n | n \geq 0\}$:

$$S_2 \rightarrow 1S_2 0 \mid \varepsilon$$

  - Then **combine:**

$$S \rightarrow S_1 \mid S_2$$
$$S_1 \rightarrow 0S_1 1 \mid \varepsilon$$
$$S_2 \rightarrow 1S_2 0 \mid \varepsilon$$

New start variable and rule combines two smaller grammars

"|" = "or" = union (combines 2 rules with same left side)

(Closed) Operations for CFLs?

- **Start** with **small grammars** and then **combine** (just like **programming**)
- "Or": $S \to S_1 \mid S_2$
- "Concatenate": $S \to S_1 S_2$
- "Star" (repetition): $S' \to S' S_1 \mid \varepsilon$

Could you write out the full proof?

Regular Language vs CFL Comparison

Regular Languages	Context-Free Languages (CFLs)
Regular Expression	Context-Free Grammar (CFG)
describes a **Regular Lang**	describes a **CFL**

Regular Language vs CFL Comparison

Regular Languages	Context-Free Languages (CFLs)
Regular Expression	Context-Free Grammar (CFG)
describes a **Regular Lang**	describes a **CFL**
Finite State Automaton (FSM)	???
recognizes a **Regular Lang**	recognizes a **CFL**

Regular Language vs CFL Comparison

Regular Languages	Context-Free Languages (CFLs)
Regular Expression	Context-Free Grammar (CFG)
describes a **Regular Lang**	describes a **CFL**
Finite State Automaton (FSM)	**Push-down Automata** (PDA)
recognizes a **Regular Lang**	recognizes a **CFL**

thm

def

def

thm

Regular Language vs CFL Comparison

Regular Languages	Context-Free Languages (CFLs)
Regular Expression	Context-Free Grammar (CFG)
describes a **Regular Lang**	describes a **CFL**
Finite State Automaton (FSM)	**Push-down Automata** (PDA)
recognizes a **Regular Lang**	recognizes a **CFL**
Proved:	To prove:
Regular Lang ⇔ Regular Expr	CFL ⇔ PDA

thm

def

def

thm