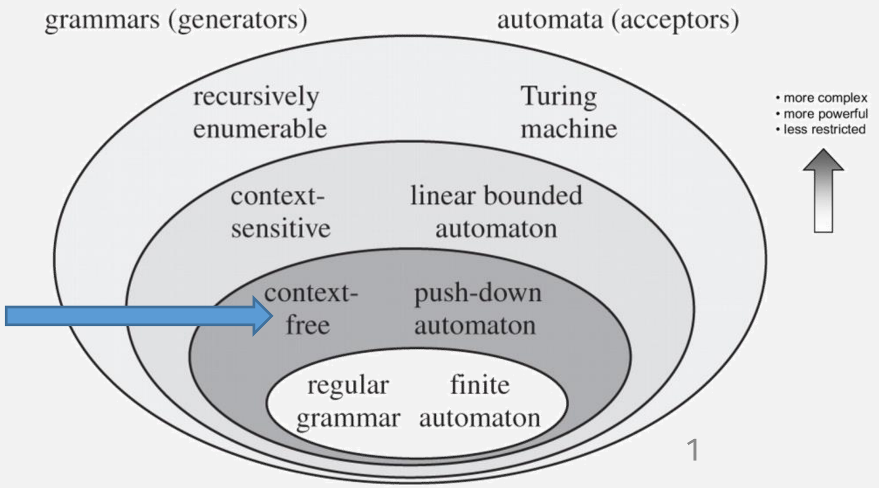
CS420
Context-Free Languages (CFLs)

Monday, March 1, 2021



Announcements

- HW4 in

- HW5 out
  - Due Sunday 3/7/2021 11:59pm EST

- Reminder: HW submissions must include README files
  - Cite your sources and collaborators
    - This is how (computer) scientists work
  - Answers must be written in your own words

<u>Last Time</u>:

Pumping lemma If A is a regular language, then there is a number p (the pumping length) where if s is any string in A of length at least 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. $|xy| \leq p$.

Let B be the language $\{0^n1^n | n \geq 0\}$. We use the pumping lemma to prove that B is not regular. The proof is by contradiction.

- If this language is not regular, then what is it???
- <u>Maybe</u>? … a **context-free language** (CFL)?
- (This language sort of resembles HTML/XML)

A Context-Free Grammar (CFG)

terminals

Top variable is
Start variable

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

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

Variables
(also called a
nonterminal)

terminals (analogous to a DFA's alphabet)

CFGs: Formal Definition

A CFG Describes a Language!

grammar G_1

R is

Top variable is **Start variable**

terminals

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

Variables (also called a nonterminal)

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

terminals (analogous to a DFA's alphabet)

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.

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

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

$S = A$,

Analogies

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

Java Language Described with CFGs

ORACLE

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 *lef 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 th 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

(partially)

Python Language 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 probably not describable with a CFG)

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

~~Python~~ Many Language (partially) 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

Generating Strings with a CFG

$G_1 =$

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

A CFG Represents a Language!

Strings in CFG's language
= all possible generated strings

$L(G_1)$ is $\{0^n\#1^n \mid n \geq 0\}$

Stop when string is all terminals

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

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

Start variable

After applying 1st rule

Used 2nd rule

Used last rule

Formal Definition of a CFL

Any language that can be generated by some context-free grammar is called a ***context-free language***

Flashback: $\{0^n 1^n | \, n \geq 0\}$

- Pumping Lemma says it's not a regular language
- It's a context-free language!
  - Proof?
  - Come up with CFG describing it ...
  - It's similar to:

$G_1 =$

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

$L(G_1)$ is $\{0^n \# 1^n | \, n \geq 0\}$

Formal Definition of a Derivation

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

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

This sequence is called a **derivation**

If u, v, and w are strings of variables and terminals, and $A \rightarrow w$ is a rule of the grammar, we say that uAv ***yields*** uwv, written $uAv \Rightarrow uwv$. Say that u ***derives*** v, written $u \overset{*}{\Rightarrow} v$, if $u = v$ or if a sequence $u_1, u_2, \ldots, u_k$ exists for $k \geq 0$ and

$$u \Rightarrow u_1 \Rightarrow u_2 \Rightarrow \ldots \Rightarrow u_k \Rightarrow v.$$

The ***language of the grammar*** is $\{w \in \Sigma^* | \, S \overset{*}{\Rightarrow} w\}$.

In-class exercise: derivations

$$
\begin{aligned}
\langle \text{EXPR} \rangle &\rightarrow \langle \text{EXPR} \rangle + \langle \text{TERM} \rangle \mid \langle \text{TERM} \rangle \\
\langle \text{TERM} \rangle &\rightarrow \langle \text{TERM} \rangle \times \langle \text{FACTOR} \rangle \mid \langle \text{FACTOR} \rangle \\
\langle \text{FACTOR} \rangle &\rightarrow (\langle \text{EXPR} \rangle) \mid \mathtt{a}
\end{aligned}
$$

- Come up with a derivation (a sequence of substs) for string:
  - a + a x a

A String Can Have Multiple Derivations

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

- <u>EXPR</u> =>
- EXPR + <u>TERM</u> =>
- EXPR + TERM x <u>FACTOR</u> =>
- EXPR + TERM x **a** =>

  ...

RIGHTMOST DERIVATION

- <u>EXPR</u> =>
- <u>EXPR</u> + TERM =>
- <u>TERM</u> + TERM =>
- <u>FACTOR</u> + TERM =>
- **a** + TERM

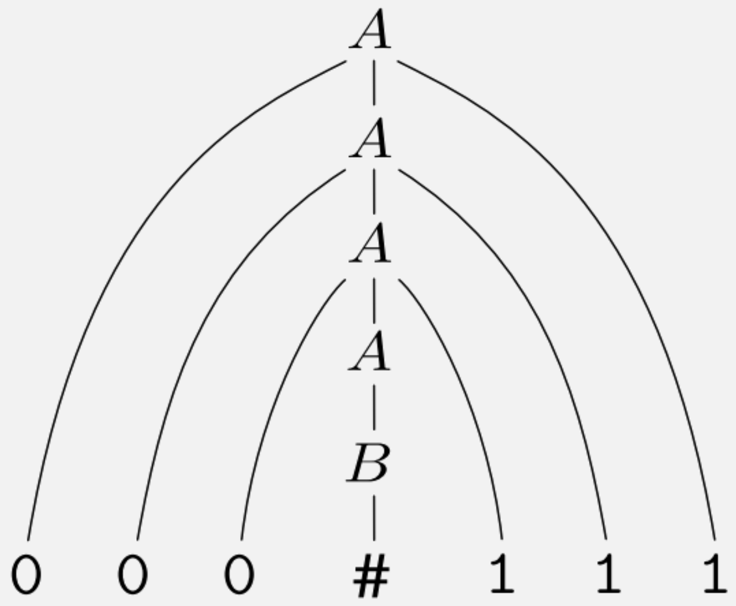
  ...

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**



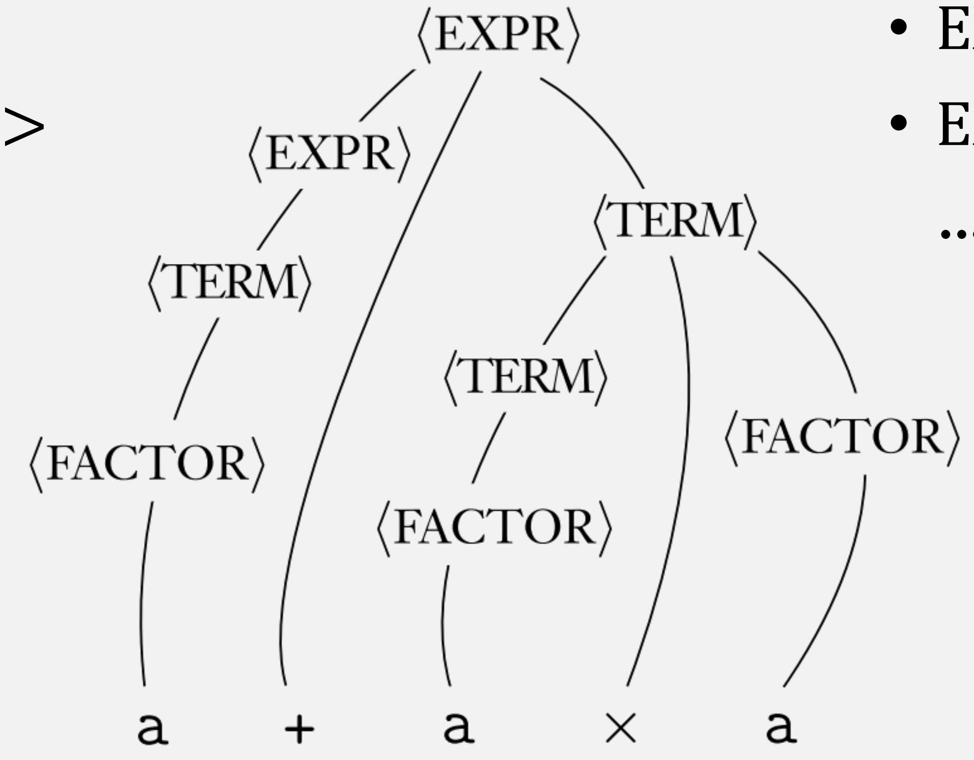
A Parse Tree gives "meaning" to a string

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=>

…

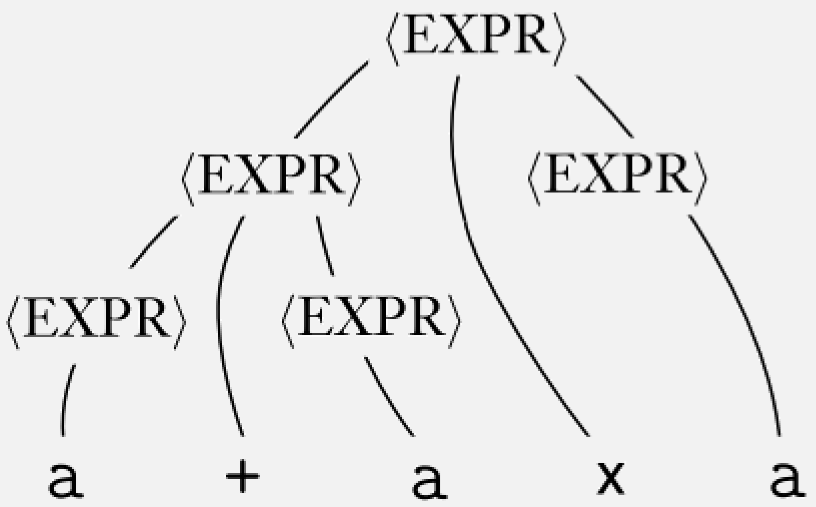
Since the "meaning" (i.e., parse tree) is same, by convention we just use **leftmost** derivation

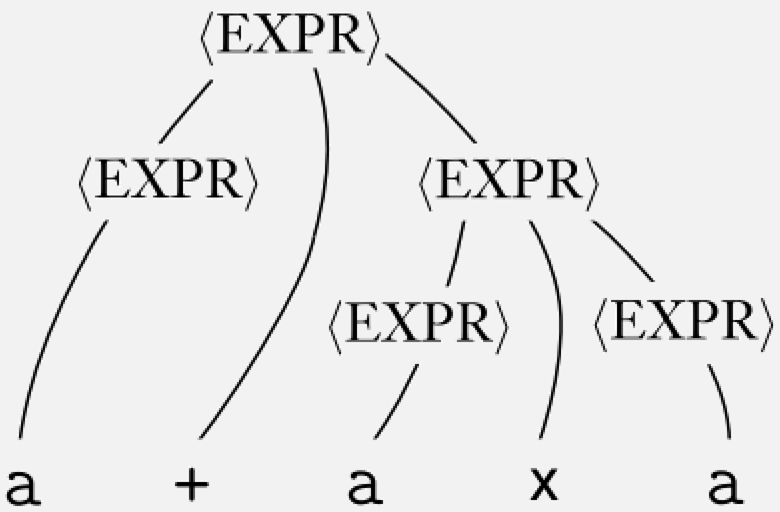
Grammars may be ambiguous

grammar G_5:

$$\langle \text{EXPR} \rangle \to \langle \text{EXPR} \rangle + \langle \text{EXPR} \rangle \mid \langle \text{EXPR} \rangle \times \langle \text{EXPR} \rangle \mid (\langle \text{EXPR} \rangle) \mid \mathtt{a}$$

Same string,
Different derivation,
<u>and</u> different parse tree!





Ambiguity

DEFINITION 2.7

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!
(why is this **bad**?)

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");
```

Ambiguous grammars are confusing. In a language, a string (program) should have only **one meaning**.

There’s no guaranteed way to create an unambiguous grammar (just have to think about it)

Designing Grammars : Basics

- Think about what you want to “link” together
- E.g., XML
  - ELEMENT → <TAG>CONTENT</TAG>
  - Start and end tags are “linked”

- Start with small grammars and then combine (just like FSMs)

Designing Grammars: Building Up

- Start with small grammars and then combine (just like FSMs)
  - To create grammar for lang $\{0^n1^n | n \geq 0\} \cup \{1^n0^n | n \geq 0\}$
  - First create grammar for lang $\{0^n1^n | n \geq 0\}$:

    $$S_1 \rightarrow 0S_11 \mid \varepsilon$$
  - Then create grammar for lang $\{1^n0^n | n \geq 0\}$:

    $$S_2 \rightarrow 1S_20 \mid \varepsilon$$
  - Then combine:

    $$S \rightarrow S_1 \mid S_2$$
    $$S_1 \rightarrow 0S_11 \mid \varepsilon$$
    $$S_2 \rightarrow 1S_20 \mid \varepsilon$$

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

Closed Operations on CFLs

- Start with small grammars and then combine (just like FSMs)

- "Or": $S \rightarrow S_1 \mid S_2$

- "Concatenate": $S \longrightarrow S_1 S_2$

- "Repetition": $S' \longrightarrow S' S_1 \mid \varepsilon$

In-class exercise: Designing grammars

alphabet Σ is $\{0,1\}$

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

- S -> 0C'0 | 1C'1 | ε — "string starts/ends with same symbol, middle can be anything"
- C' -> C'C | ε — "all possible terminals, repeated (ie, all possible strings)"
- C -> 0 | 1 — "all possible terminals"

Check-in Quiz 3/1

On gradescope