CMSC330: Context Free Grammars

Chris Kauffman

Last Updated:
Fri Oct 13 03:41:03 PM EDT 2023

Logistics

Assignments

- No online lecture quiz this week due to Exam 1
- Project 4 is up, OCaml basics, due Sun 15-Oct

Reading

Chapter 5 "Context Free Grammars" from Automata Theory, Languages, and Computation by Hopcroft, Motwani, Pullman

Goals

Context Free Grammars, notation, terminology, usage, limits

Limits of Regexs

- Recall that **Regular Languages** (recognized by Regexs/FSMs) had a limit to their power to recognize/accept
- Could not derive a Regex for the following two languages:
  1. Equal-ABs $= \{a^n b^n | n > 0\}$
     Examples of: Equal-ABs $=$ `{ab, aabb, aaabbb, aaaabbbb, ...}`
  2. Balanced-Paren $= \{(^n)^n | n > 0\}$
     Examples of Balanced-Paren $=$ `{(), (()), ((())), ...}`
- Clearly the latter has applications in processing programming languages
- Rather than a Regular language, these are examples of **Context Free Languages**

Context Free Grammars (CFGs)

- **Terminal Symbols** (lowercase letters) comprise the language alphabet just as in FSMs (includes ϵ "empty string")
- **Non-terminal Symbols or Variables** (capital letters) which will be replaced by other symbols
- **Production Rules**: a single Non-terminal on a left-hand side produces right-hand side combination of Terminals/Non-terminals
- Usual convention is the first Non-terminal with a Production Rule listed is the **Start Symbol**

Example: CFG for Equal-ABs

$$X \rightarrow aXb$$
$$X \rightarrow \epsilon$$
$$or$$
$$X \rightarrow aXb|\epsilon$$

- Terminals: Only a, b (not counting empty string ϵ
- Non-Terminals: X
- Productions: 2, both for X
- Start Symbol: X

Deriving/Producing Strings from a CFG

Example: CFG for Equal-ABs

$$X \rightarrow aXb \quad (1)$$
$$X \rightarrow \epsilon \quad (2)$$

Useful to number each of the production rules so they can be easily referenced during derivations

Derive: $aabb$

$X \Rightarrow_1 aXb$ Use production 1
$\Rightarrow_1 aaXbb$ Use production 1
$\Rightarrow_2 aabb$ Use production 2

Derive: Others

- $aaabbb$:
  $X \Rightarrow_1 aXb \Rightarrow_1 aaXbb \Rightarrow_1 aaaXbbb \Rightarrow_2 aaabbb$
- ab: $X \Rightarrow_1 aXb \Rightarrow_2 ab$

Notation: In lecture examples will try to use

- $\rightarrow$ (single arrow) for CFG production rules
- $\Rightarrow$ (double arrow) for derivations

A more Interesting CFG

CFG for PlusTimes

$$
\begin{aligned}
A &\rightarrow A + A && (1)\\
A &\rightarrow A * A && (2)\\
A &\rightarrow N && (3)\\
N &\rightarrow DN && (4)\\
N &\rightarrow D && (5)\\
D &\rightarrow 0|1|2|3|..|9 && (6)
\end{aligned}
$$

- Terminals: $+, *$ and numbers like $5, 124$
- Non-Terminals: A, N, D
- Productions: 3 for A, 2 for N, "1" for D, 5 total

Sample Derivation

$$
\begin{aligned}
\underline{A} &\Rightarrow_1 \underline{A} + A\\
&\Rightarrow_2 \underline{A} * A + A\\
&\Rightarrow_3 N * \underline{A} + A\\
&\Rightarrow_3 N * N + \underline{A}\\
&\Rightarrow_3 \underline{N} * N + N\\
&\Rightarrow_4 D\underline{N} * N + N\\
&\Rightarrow_5 DD * \underline{N} + N\\
&\Rightarrow_4 DD * D\underline{N} + N\\
&\Rightarrow_5 DD * DD + \underline{N}\\
&\Rightarrow_5 \underline{DD} * \underline{DD} + \underline{D}\\
&\Rightarrow_6 12 * 34 + 5
\end{aligned}
$$

The Non-terminal (variable) to which production rules are applied are underlined

(Optional) Exercise: Practice Deriving

- Derive a few strings from the PlusTimes CFG
- Don't need to show every step, just get the big ones

CFG for PlusTimes

$$
\begin{aligned}
A &\rightarrow A + A && (1)\\
A &\rightarrow A * A && (2)\\
A &\rightarrow N && (3)\\
N &\rightarrow DN && (4)\\
N &\rightarrow D && (5)\\
D &\rightarrow 0|1|2|3|..|9 && (6)
\end{aligned}
$$

Derive: 123

Derive: $9 + 2 * 7$

Answers: Practice Deriving

CFG for PlusTimes

$$A \to A + A \quad (1)$$
$$A \to A * A \quad (2)$$
$$A \to N \quad (3)$$
$$N \to DN \quad (4)$$
$$N \to D \quad (5)$$
$$D \to 0|1|2|3|..|9 \quad (6)$$

Derive: 123

$A \Rightarrow_3 N \Rightarrow_4 DN \Rightarrow_4 DDN \Rightarrow_5 DDD \Rightarrow_6 123$

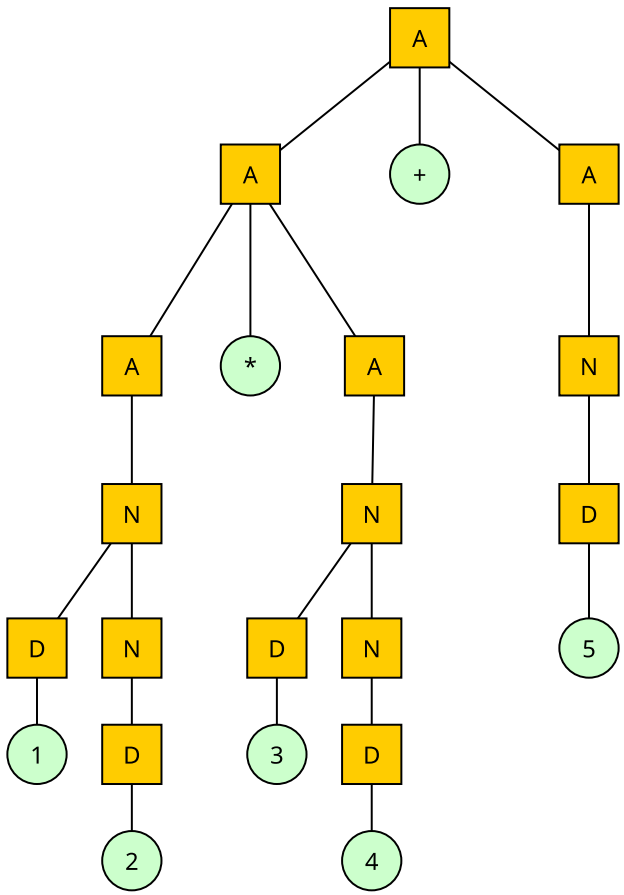
Derive: $9 + 2 * 7$

$A \Rightarrow_1 A+A \Rightarrow_2 A+A*A \Rightarrow_4 N+N*N \Rightarrow_5 D+D*D \Rightarrow_6 9+2*7$

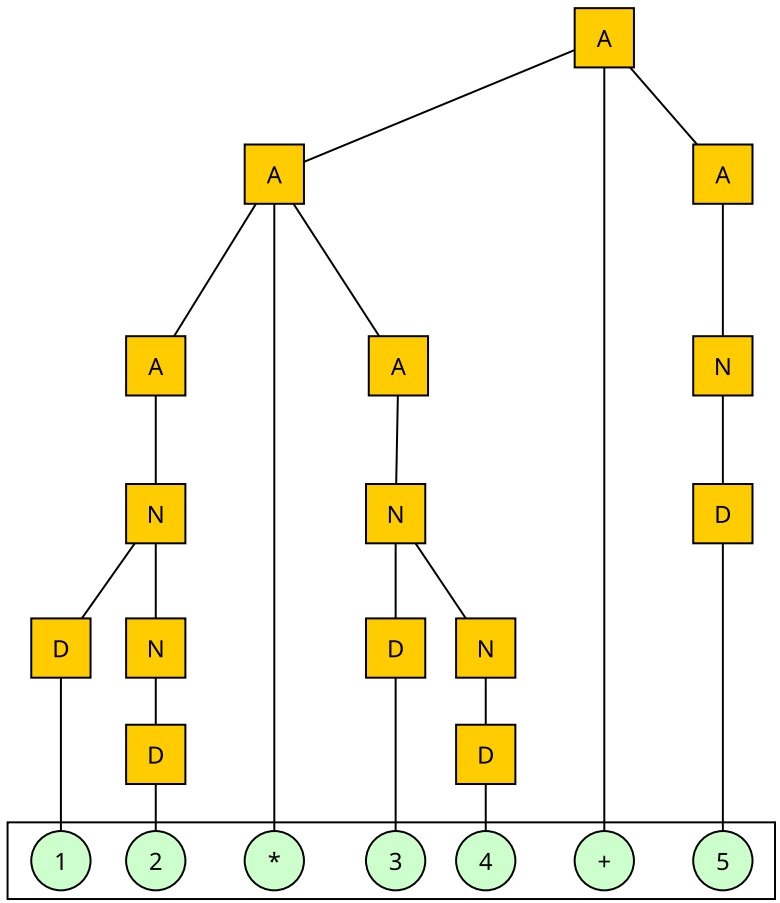
CFG Derivations and Parse Trees

Parse Trees are a graphical representation of a string derived/produced from a CFG, examples below. Can show formally equivalence of Parse Trees and CFGs via inductive proofs.

TYPICAL PARSE TREE with terminals at differing heights



TERMINALS LEVEL to show the Derived Expression



Yield of the tree

Leftmost and Rightmost Derivations

- Notice **choices** in earlier derivation of $12 * 34 + 5$: **pick** a Non-terminal and apply a production rule
- Often want to eliminate choices so enforce an ordering to derivations
- **Leftmost / Left-hand** derivation always applies a production to the leftmost non-terminal
- **Rightmost / Right-hand** derivation does likewise for rightmost non-terminal
- Note: if there are several possible productions for leftmost symbol, leftmost derivation doesn't specify which to use...

Leftmost Derivation

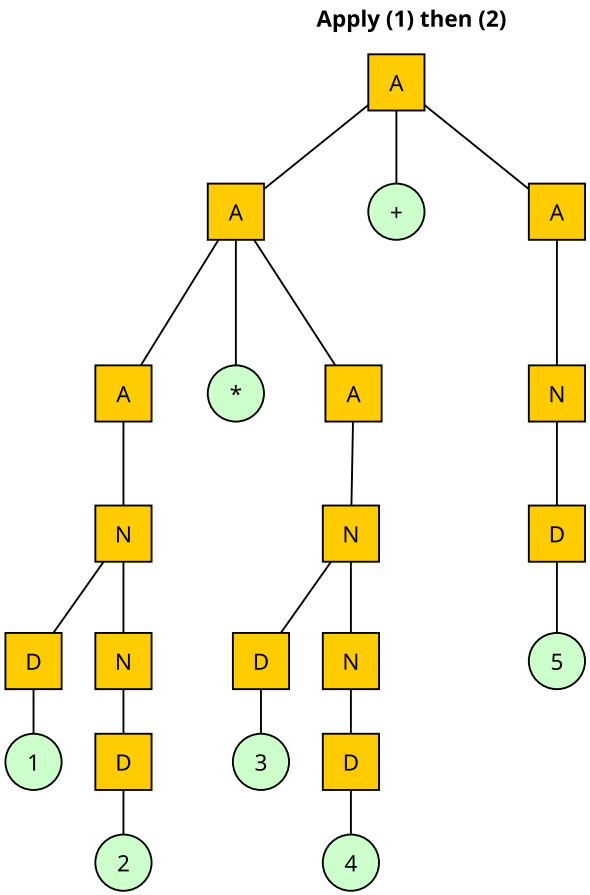
$$
\begin{aligned}
\underline{A} &\Rightarrow_1 \underline{A} + A \\
&\Rightarrow_2 \underline{A} * A + A \\
&\Rightarrow_3 \underline{N} * A + A \\
&\Rightarrow_4 \underline{D}N * A + A \\
&\Rightarrow_6 1\underline{N} * A + A \\
&\Rightarrow_5 1\underline{D} * A + A \\
&\Rightarrow_6 12 * \underline{A} + A \\
&\Rightarrow_3 12 * \underline{N} + A \\
&\Rightarrow_4 12 * \underline{D}N + A \\
&\Rightarrow_6 12 * 3\underline{N} + A \\
&\Rightarrow_5 12 * 3\underline{D} + A \\
&\Rightarrow_6 12 * 34 + \underline{A} \\
&\Rightarrow_4 12 * 34 + \underline{N} \\
&\Rightarrow_5 12 * 34 + \underline{D} \\
&\Rightarrow_6 12 * 34 + 5
\end{aligned}
$$

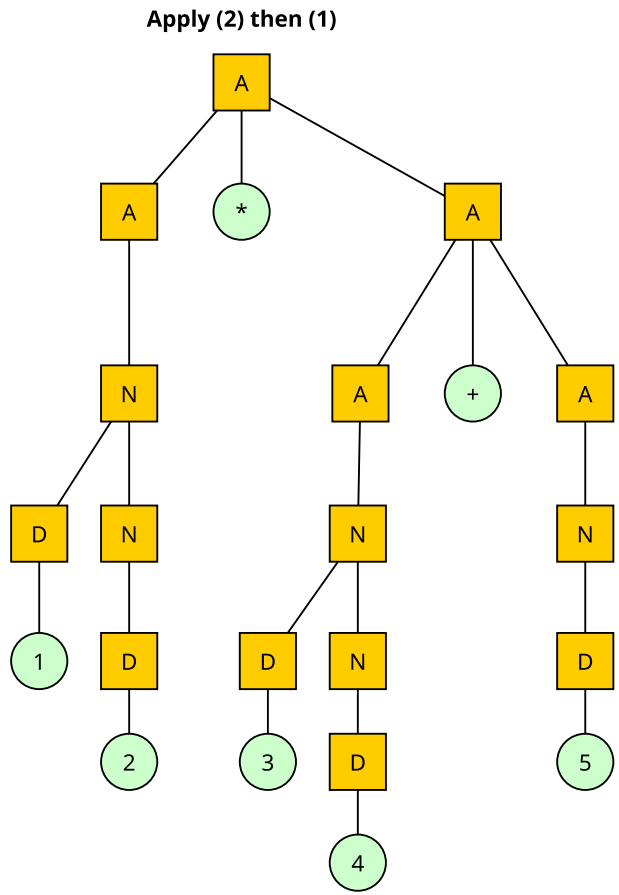
Exercise: Ambiguity in CFGs

This grammar has another choice: when deriving $12 * 34 + 5$, which order to apply

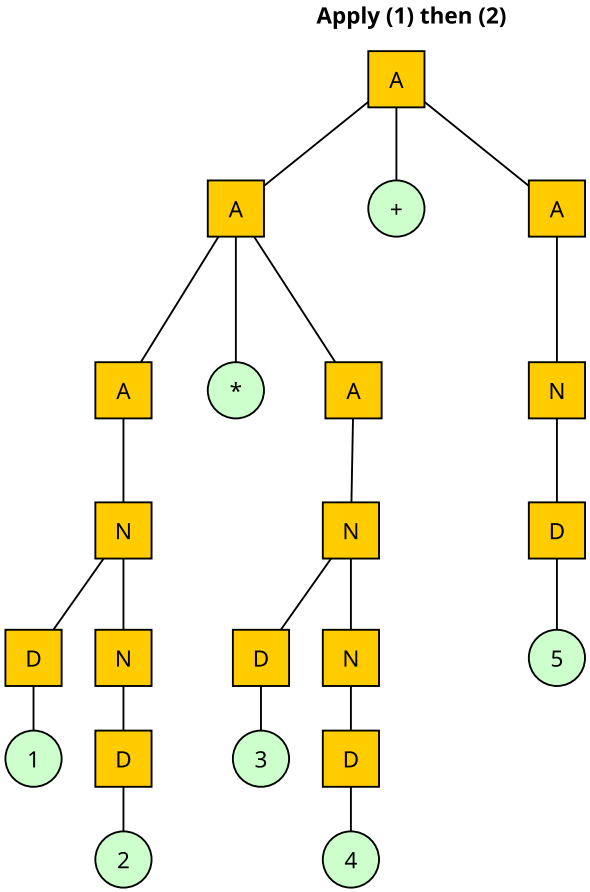
- (1) $A \to A + A$ then (2) $A \to A * A$ OR
- (2) $A \to A * A$ then (1) $A \to A + A$

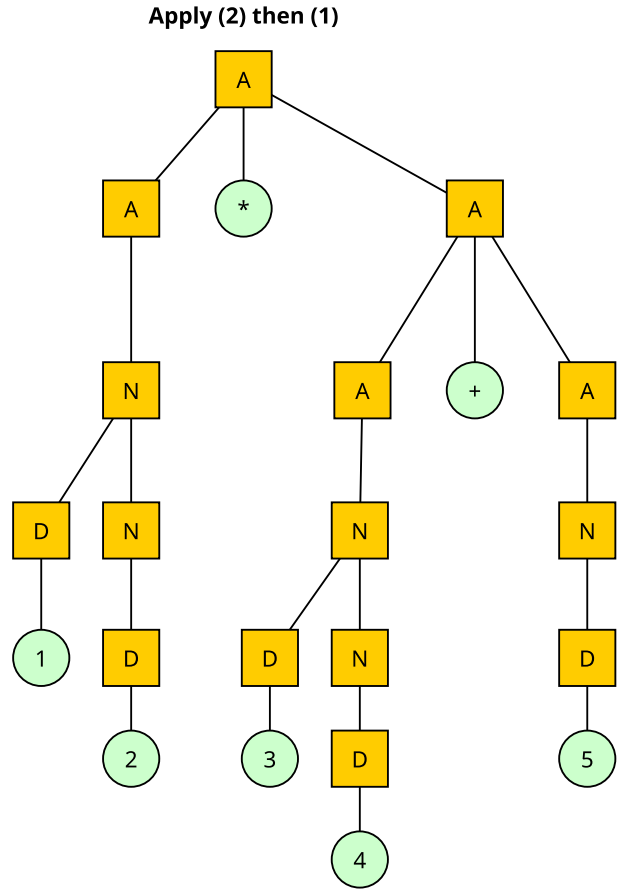
What is the difference shown in the two Parse Trees?





Answers: Ambiguity in CFGs





- $(12 * 34) + 5$ for (1) then (2)
- $12 * (34 + 5)$ for (2) then (1)

Low-precedence operator is higher in the tree, typically want higher-precedence for $*$ over $+$, resolve this momentarily via an adjustment to the grammar

Exercise: Proving a Grammar is Ambiguous

To prove a Grammar is ambiguous

- Using only leftmost derivations. . .
- Derive the same string via two different choices of productions
- Creates two different parse trees for the same string

Show the below grammar is ambiguous by finding two left-most derivations for the same string

$$X \to aX \qquad (1)$$
$$X \to Xb \qquad (2)$$
$$X \to Y \qquad (3)$$
$$Y \to b \qquad (4)$$
$$Y \to \epsilon \qquad (5)$$

Answers: Proving a Grammar is Ambiguous

To prove a Grammar is ambiguous

- Using only Leftmost derivations. . .
- Derive the same string via two different choices of productions
- Creates two different parse trees for the same string

Show the below grammar is ambiguous by finding two left-most derivations for the same string

$$X \to aX \quad (1)$$
$$X \to Xb \quad (2)$$
$$X \to Y \quad (3)$$
$$Y \to b \quad (4)$$
$$Y \to \epsilon \quad (5)$$

ab has several different leftmost derivations; two are

$X \Rightarrow_1 aX \Rightarrow_2 aXb \Rightarrow_3 aYb \Rightarrow_5 ab$

AND

$X \Rightarrow_1 aX \Rightarrow_3 aY \Rightarrow_4 ab$

Drawing Parse Trees for these is good practice

Resolving Ambiguities in CFGs

"In an ideal world, we would be able to give you an algorithm to remove ambiguity from CFG's, much as we were able to show an algorithm in Section 4.4 to remove unnecessary states of a finite automaton. However, the surprising fact is... that there is no algorithm whatsoever that can even tell us whether a CFG is ambiguous in the first place.
Fortunately, the situation in practice is not so grim. For the sorts of constructs that appear in common programming languages, there are well-known techniques for eliminating ambiguity."
–Automata Theory, Languages, and Computation by Hopcroft, Motwani, & Ullman

CFG PlusTimes

$$A \rightarrow A + A \quad (1)$$
$$A \rightarrow A * A \quad (2)$$
$$A \rightarrow N \quad (3)$$
$$N \rightarrow DN \quad (4)$$
$$N \rightarrow D \quad (5)$$
$$D \rightarrow 0|1|2|3|..|9 \quad (6)$$

- Ambiguity comes here from the choice of whether to apply Proudction Rules (1) or (2) when both are options
- Adjust the CFG to remove this choice to eliminate the ambiguity

Example: Resolving Ambiguity in PlusTimes CFG

- Introduce a new non-terminal associated with multiplication
- No longer any choices on how to derive sub-expressions involving *

PlusTimes (Ambiguous)

$$
\begin{aligned}
A &\rightarrow A + A && (1)\\
A &\rightarrow A * A && (2)\\
A &\rightarrow N && (3)\\
N &\rightarrow DN && (4)\\
N &\rightarrow D && (5)\\
D &\rightarrow 0|1|2|3|..|9 && (6)
\end{aligned}
$$

PlusTimesUA (Unambiguous)

$$
\begin{aligned}
A &\rightarrow A + A && (1)\\
A &\rightarrow M && (2)\\
M &\rightarrow M * M && (3)\\
M &\rightarrow N && (4)\\
N &\rightarrow DN && (5)\\
N &\rightarrow D && (6)\\
D &\rightarrow 0|1|2|3|..|9 && (7)
\end{aligned}
$$

Deriving $12 * 34 + 5$ with PlusTimesUA

$$\underline{A} \Rightarrow_1 \underline{A} + A \Rightarrow_2 \underline{M} + A \Rightarrow_3 \underline{M} * M + A \Rightarrow_{4567}$$
$$12 * \underline{M} + A \Rightarrow_{4567} 12 * 34 + \underline{A} \Rightarrow_2 12 * 34 + \underline{M} \Rightarrow_{467} 12 * 34 + 5$$

Operator Precedence and CFGs

- When creating CFGs to parse programming languages, often associate production rules for different levels of **operator precedence**
- Precedence is the human notion of what to do first
- Encoding this in the CFG ensures Parse Trees will have high-precedence ops low down in the tree: eval these first and use results in lower-precedence expressions

CFG UnaryMath

$$A \rightarrow A + A | A - A | M \quad (1)$$
$$M \rightarrow M * M | M / M | U \quad (2)$$
$$T \rightarrow N | U \quad (3)$$
$$U \rightarrow - N \quad (4)$$
$$N \rightarrow \text{number} \quad (5)$$

$$5 + 12 / -3 - 7 \equiv (5 + (12 / (-3)) - 7)$$

- Using CFG on left to parse
- Parentheses indicate "natural order" as well as depth in parse tree
- Unary negation has highest precedence

Exercise: A CFG with Cycles

CFG ParApp

$$A \rightarrow (B) \quad (1)$$
$$A \rightarrow \text{id} \mid \text{number} \quad (2)$$
$$B \rightarrow A \quad (3)$$
$$B \rightarrow BA \quad (4)$$

- id: a valid identifies like `x`, `+`, `foo`, `*`
- number: a valid number like `123`, `0.451`, `etc`

What's interesting about this CFG as compared to those we've seen earlier?

Strings

```
; A
hello
; B
(doit)
; C
(+ 1 2 3)
; D
(define (double x) (* x 2))
; E
((lambda (y z) (if (< y z) y z))) 42 24)
; F
(let ((a 7) (b 9)) (+ a (* 10 b)))
```

Which of these can be derived from ParApp?

Answers: A CFG with Cycles

CFG ParApp

$$A \rightarrow (B) \quad (1)$$
$$A \rightarrow \text{id} | \text{number} \quad (2)$$
$$B \rightarrow BA \quad (3)$$
$$B \rightarrow A \quad (4)$$

What's interesting about this CFG?

- It has a loop or **cycle** of sorts: A→ B → A→ B...
- Such cycles often arise in Programming Language CFGs as nesting of expressions within other expressions is common

Strings

```
; A
hello
; B
(doit)
; C
(+ 1 2 3)
; D
(define (double x) (* x 2))
; E
((lambda (y z) (if (< y z) y z))) 42 24)
; F
(let ((a 7) (b 9)) (+ a (* 10 b)))
```

Which of these can be derived from ParApp?

- All of them: these are all valid Lisp/Scheme/Racket
- Note the simplicity of the CFG and lack of ambiguity *All it costs you are parens...*

Parse Trees vs Abstract Syntax Trees (ASTs)

There are technical distinctions in some circles between these two:

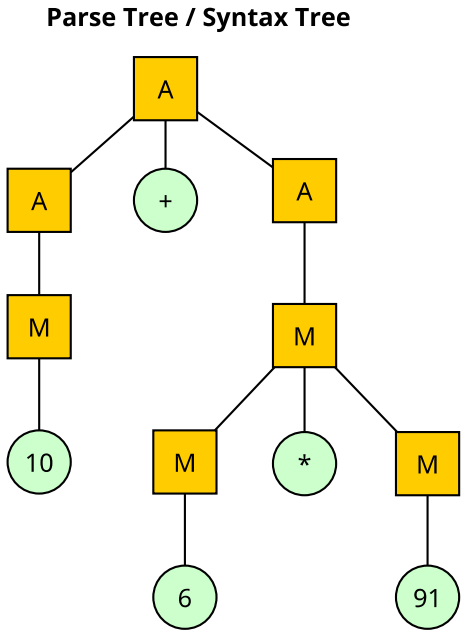
1. **Parse Trees**: show all characters and how they would derive from a CFG
2. **Abstract Syntax Tree (AST)**: eliminate some raw characters, retains data important to evaluate the semantic meaning

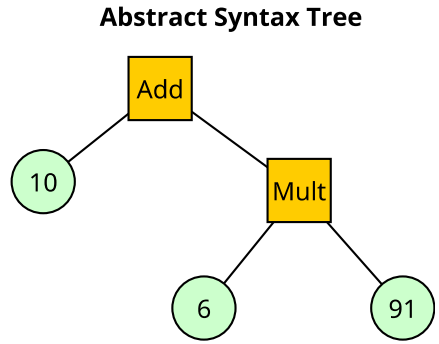
Durng language processing, may never deal directly with raw Parse Tree with "parsing" producing a tree more like an AST

CFG PlusTimesUAS

$$A \rightarrow A + A \quad (1)$$
$$A \rightarrow M \quad (2)$$
$$M \rightarrow M * M \quad (3)$$
$$M \rightarrow \text{number} \quad (4)$$

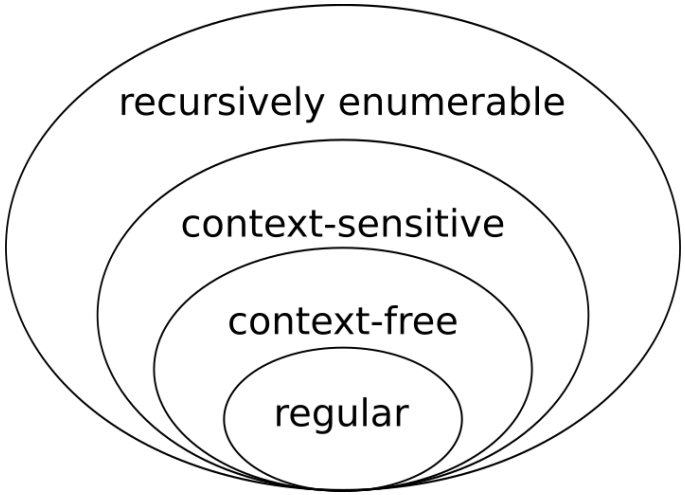
Example: $10 + 6 * 91$





Beyond CFGs: The Chomsky Hierarchy

- Noam Chomsky invented Context Free Grammars during his study of **Natural Languages**
- CFGs can model some parts of Natural Languages but not all
- Chomsky identified a hierarchy of grammar types and related them to computational power in his work



Source: Wikipedia

Grammar	Languages	Recognizing Automaton	Production rules (constraints)*	Examples[5][6]
Type-3	Regular	Finite state automaton	$A \to \mathrm{a}$ and $A \to \mathrm{a}B$	$L = \{a^n \mid n \geq 0\}$
Type-2	Context-free	Non-deterministic pushdown automaton	$A \to \alpha$	$L = \{a^n b^n \mid n > 0\}$
Type-1	Context-sensitive	Linear-bounded non-deterministic Turing machine	$\alpha A \beta \to \alpha \gamma \beta$	$L = \{a^n b^n c^n \mid n > 0\}$
Type-0	Recursively enumerable	Turing machine	$\gamma \to \alpha$ (γ non-empty)	$L = \{w \mid w$ describes a terminating Turing machine$\}$

* Meaning of symbols:

- a = terminal
- A, B = non-terminal
- α, β, γ = string of terminals and/or non-terminals

Next: Lexing / Parsing and Evaluation

- Will answer the question of how to determine if a string of terminals (characters) is recognized/accepted by a CFG or not: **Parsing**
- If the string recognized, the most frequent thing to do with programming language strings is **Evaluate** them