The Pumping Lemma & Introduction to Parsing

CSCI 3136: Principles of Programming Languages

Agenda

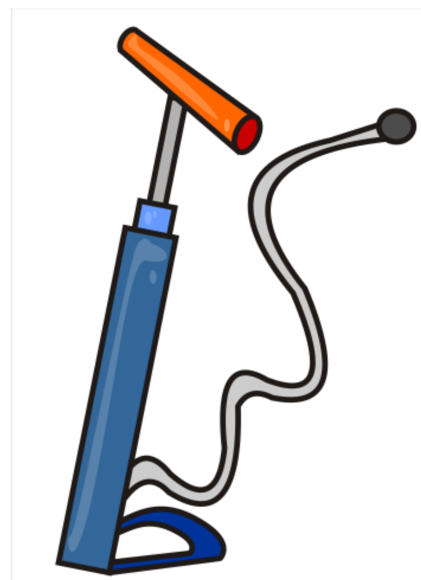
- Announcements
  - Assignment 2 is out and due May 31
- Readings:
  - Today: 2.3.0, 2.3.1
  - Note: I recommend using alternative texts for this part of the course:
  - E..g, Hopcorft et al, “Introduction to Automata Theory”
- Lecture Contents
  - Pumping Lemma
  - Introduction to Parsing

Examples from last lecture:

- Show that $L = \{a^p \mid p \text{ is not prime}\}$ is not regular.
  - Recall that $\{a^p \mid p \text{ is prime}\}$ is not regular
- Show that $L = \{a^p b^q \mid p \text{ or } q \text{ is prime}\}$ is not regular.
- Show that $L = \{a^p a^* a^p \mid p \text{ is prime}\}$ is regular.

Nonregular Languages

- Problem: Not all languages are regular!
  E.g. $L = \{0^n1^n | n >= 0\}$ is not regular.
- Intuition: We need to keep track of how many 0's we encounter.
- A DFA has a finite number of states, so beyond that number we cannot keep track.
- How do we prove this formally?
  The Pumping Lemma!



Intuition

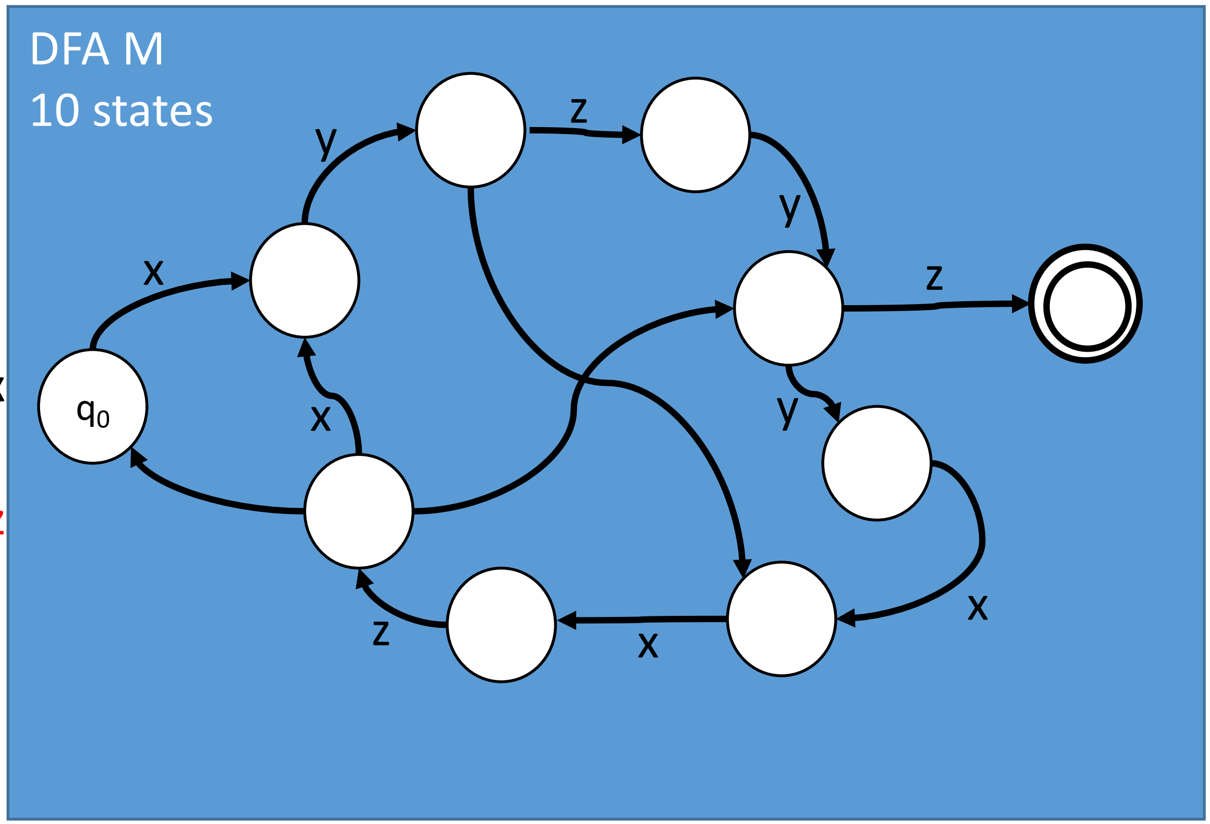
xyzyz

xyzyyxxzx

xyzyyxxzxyzyz

x(yzyyxxzx)^{i}yzyz

Any string of this form is accepted!



Observation: If a DFA with n states reads a string of length n or greater, one or more states will be visited multiple times (there will be a cycle).

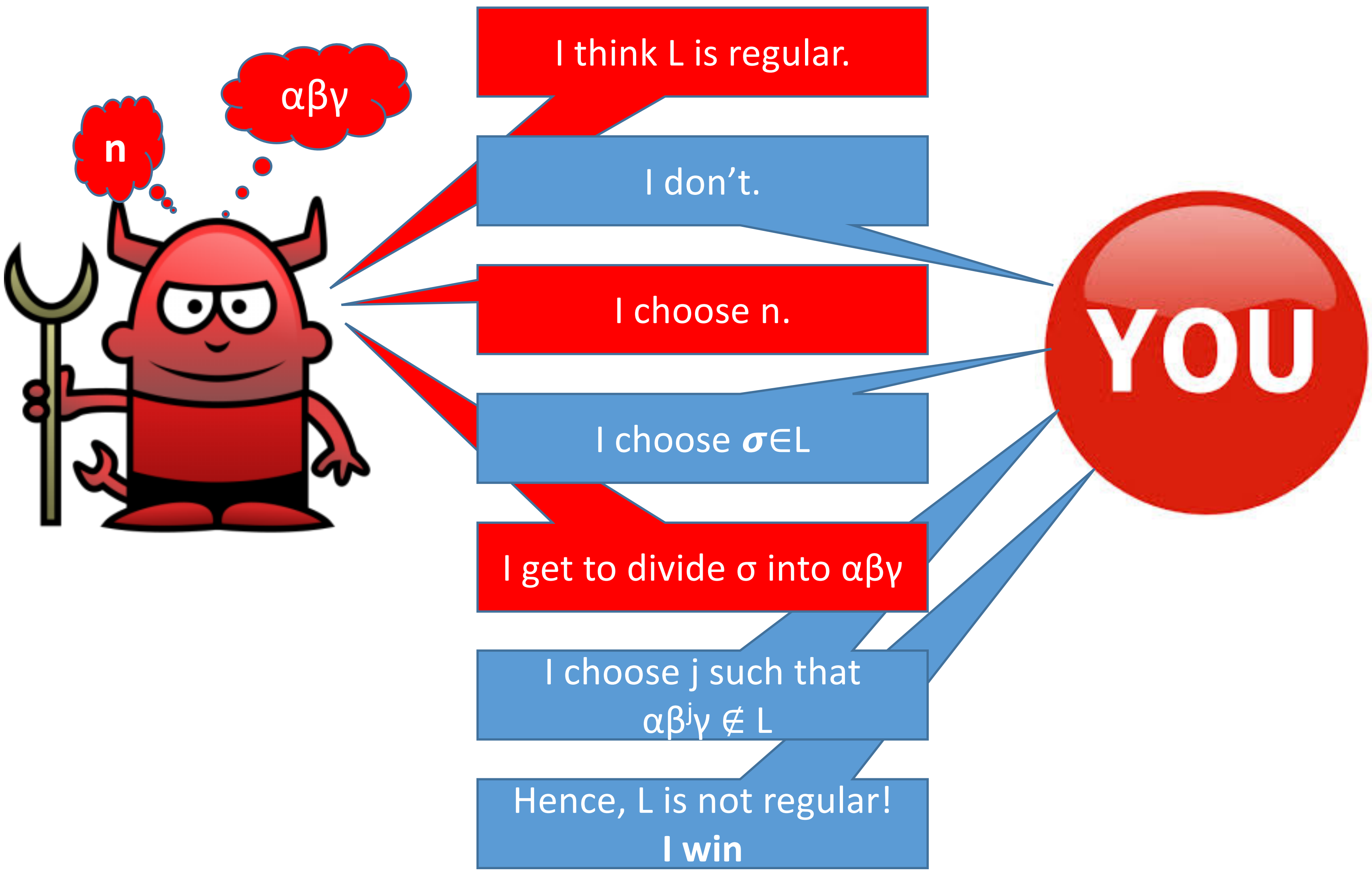
The Pumping Lemma

For every regular language L, there exists a constant n such that every $\sigma \in L$, where $|\sigma| \geq n$, can be divided into three substrings $\sigma = \alpha\beta\gamma$ with the following properties:

- $|\alpha\beta| \leq n$
- $|\beta| > 0$,and
- $\alpha\beta^k\gamma \in L, \forall k \geq 0$

- We can use this Lemma to show that a given language is non-regular.

Using the Pumping Lemma is like an Argument with the Devil



Applying the Pumping Lemma

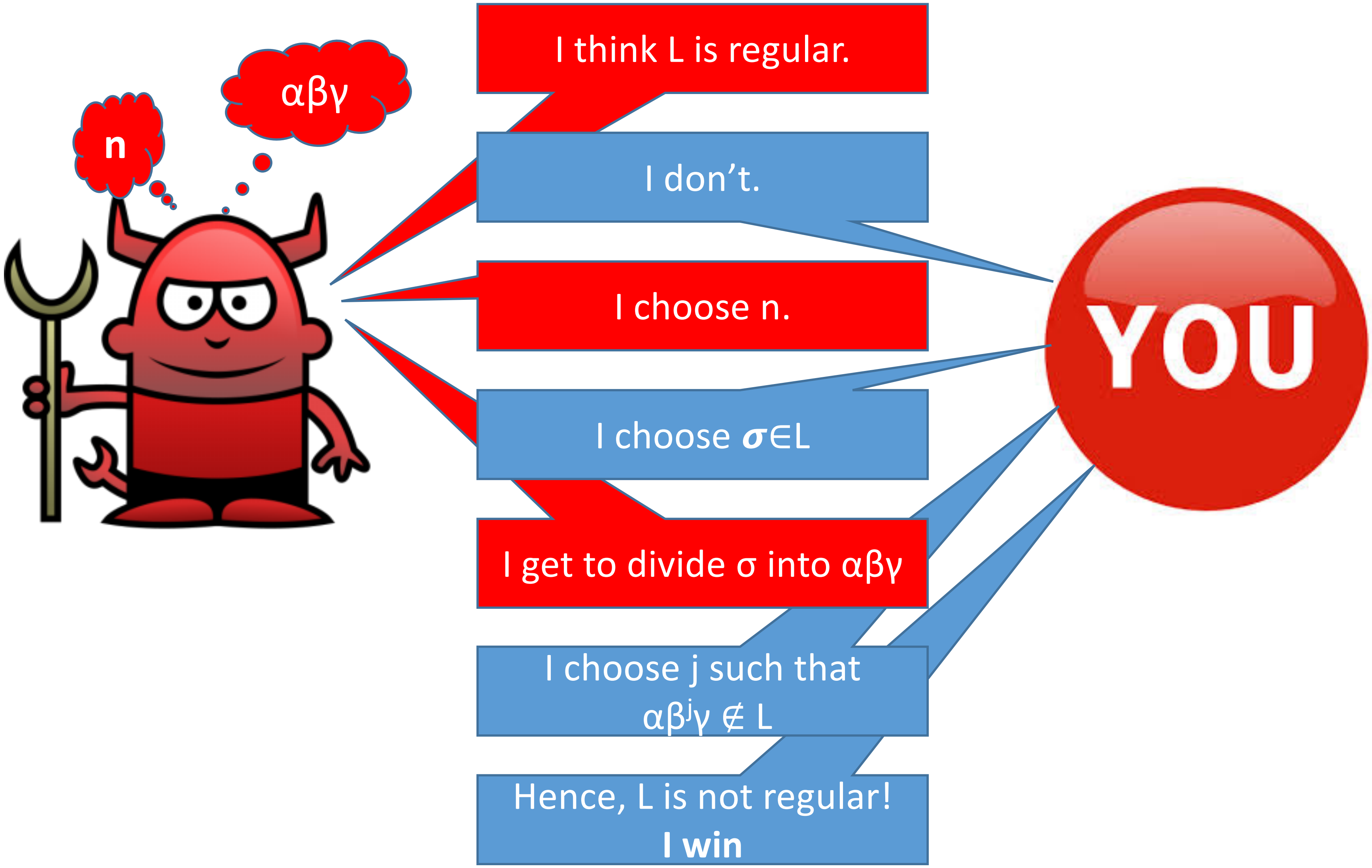
Procedure: To show that L is not regular

- Convince yourself L is not regular (intuition)
- Assume that L is regular and that there is a constant n as stated by the Pumping Lemma
- Select $\sigma \in L$ such that
  - $|\sigma| > n$
  - $\sigma = \alpha\beta\gamma$, **for all α and β**
    - $|\alpha\beta| \leq n$
    - $|\beta| > 0$
  - There ***exists a $j \geq 0$*** such that $\alpha\beta^j\gamma \notin L$
- But according to Pumping Lemma, $\sigma \in L$.
- Contradiction!
- Therefore L is not regular.

σ will (typically) be a function of n

HARD PART

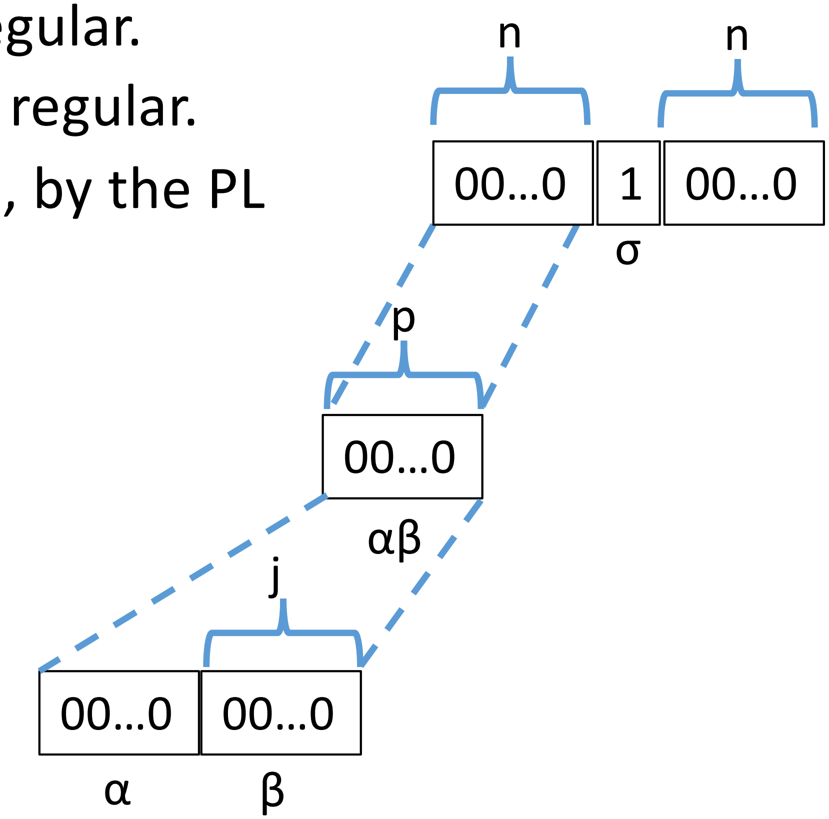
Using the Pumping Lemma is like an Argument with the Devil

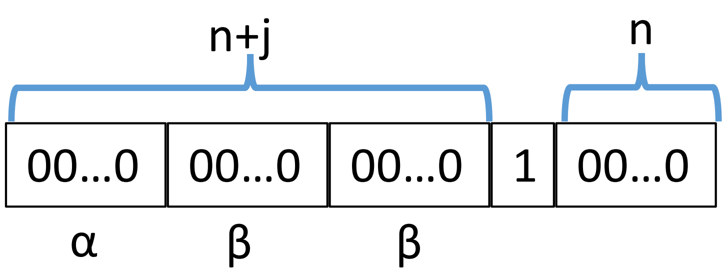


Example: Use the Pumping Lemma

Show that $L = \{0^m 1 0^m \mid m \geq 0\}$ is not regular.

- Proof by contradiction: Assume L is regular.
- If L is regular, then there exists an n, by the PL
- Select $\sigma = 0^n 1 0^n$
- Therefore, **for all α and β**
    - $\alpha\beta = 0^p$, because $p \leq n$
    - $\beta = 0^j$, $0 < j \leq p$
- By the PL, $\alpha\beta^2\gamma \in L$
- But $\alpha\beta^2\gamma = 0^{n+j} 1 0^n \notin L$
- **Contradiction!**





Examples

- $L = \{a^i b^j | i < j\}$
- $L = \{a^p | p \text{ is prime}\}$
- $L = \{a^i b^j \mid i = j \bmod 3\}$

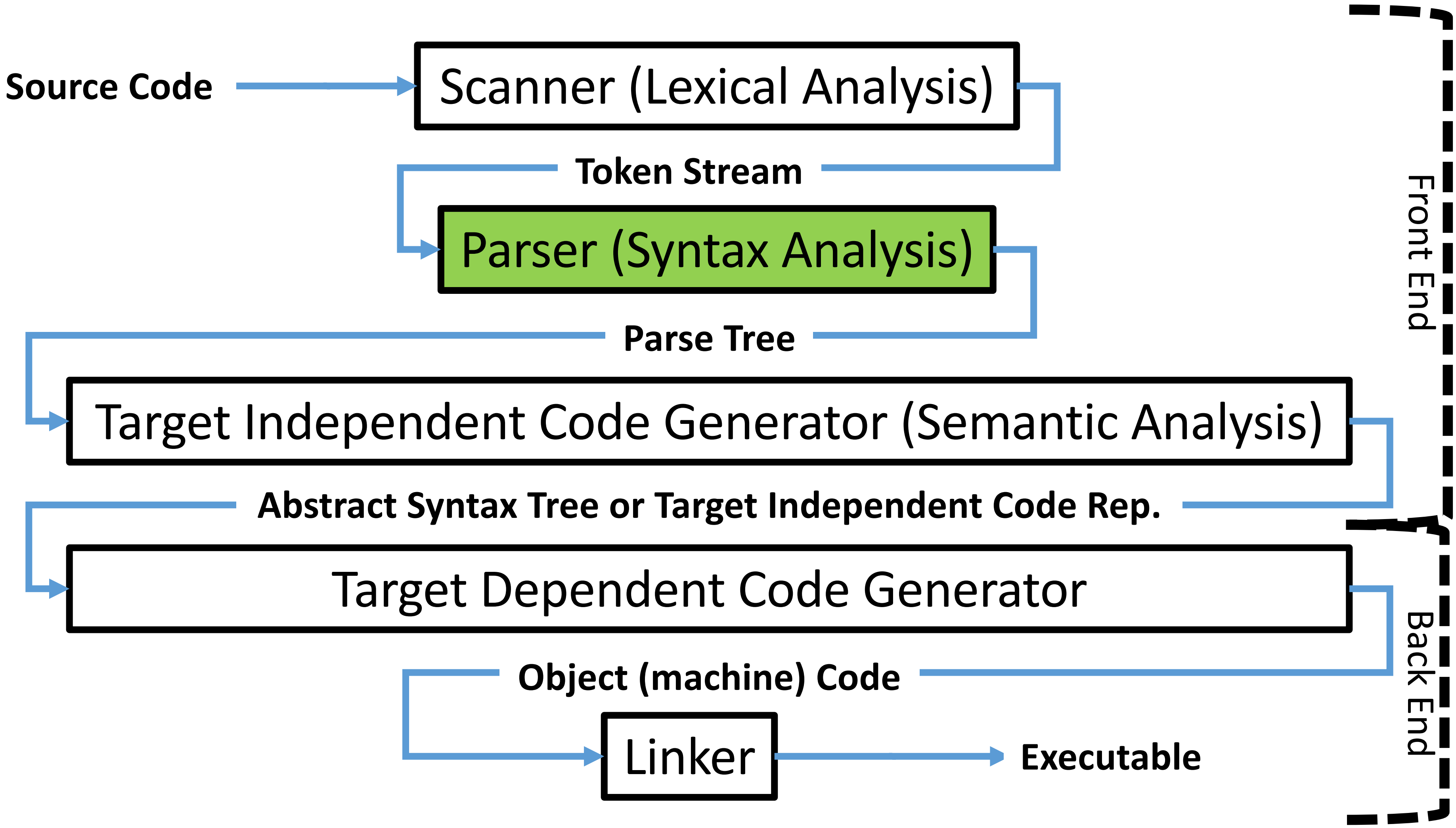
  This one is actually regular
- Note: We cannot use the Pumping Lemma to prove a language is regular.
- Question: How do you show a language is regular?
  - Construct a regular expression for the language
  - Construct an NFA that recognizes the language
  - Construct the language from known Regular Languages using closure properties of regular languages.
- So, we're done… right?

Why Do We Need a Parser?

- A scanner yields a stream of tokens
- Q: Is this sufficient to determine if the input is a valid program?
- A: No! Most programming languages are not regular!

  E.g. braces and brackets must match: $((1 + 3) * (3 + 2))$
- Scanners are useful for
  - Checking if program's tokens are correct
  - Providing higher level representation of programs
- Scanners cannot check if the syntax is correct
  - Analogy: Correctly spelled words do not make a correct sentence
- We need a different mechanism for checking syntax
- We need a parser

Recall: Phases of Compilation



Meet the Parser

- Parsing takes a stream of tokens
  - Checks whether the tokens represent a syntactically correct program
  - Creates a parse tree (a high level representation of the program)
- Question: How do we know what the correct syntax is?
- Answer: Based on the language specification
- Question: How do we specify the syntax
- Answer: By a grammar

Grammars

- Idea: Grammars specify the syntax of a language
- Example: English Sentences
  - *Sentence* → *Phrase Verb Phrase .*
  - *Phrase* → *Noun* | *Adjective Phrase*
  - *Adjective* → `big` | `small` | `green`
  - *Noun* → `boss` | `cheese`
  - *Verb* → `is` | `jumps` | `eats`

  Valid Sentences:
  - `Boss is big cheese.`
  - `Boss eats green cheese.`
  - `Green cheese jumps boss.`

  Not all valid sentences make sense!

Example: Arithmetic Expressions

Grammar

$E \rightarrow E\ Op\ E$

$E \rightarrow - E$

$E \rightarrow (E)$

$E \rightarrow Number$

$E \rightarrow Identifier$

$Op \rightarrow +$

$Op \rightarrow -$

$Op \rightarrow /$

$Op \rightarrow *$

Valid Sentences

(1 + 2 - 3) * 4

- - 3

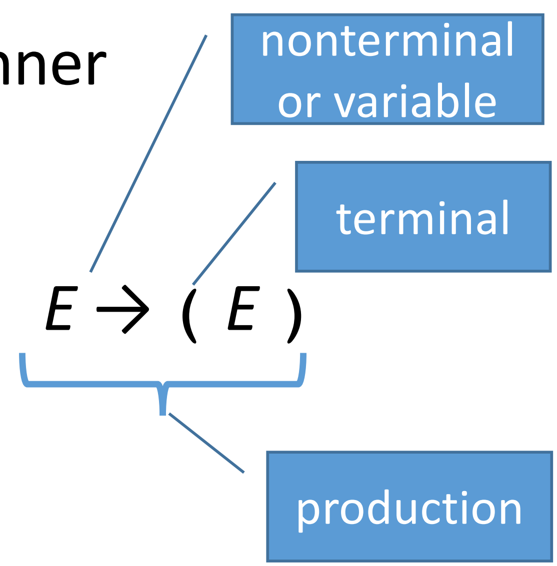
a + b

Typically programming languages are specified by Context Free Grammars (CFG)

Context Free Grammars (CFG)

A CFG G is a 4-tuple G = (V,Σ,P,S) where

- V is the set of non-terminals
  - Also known as “Variables”
  - Denoted by Capitalized letters/words
- Σ is the set of terminals
  - The text tokens returned by the scanner
- P is the set of productions
  - Of the form $N \rightarrow (\Sigma \cup V)^*$, $N \in V$
  - Also known as “Rewriting Rules”
- S is the start symbol, $S \in V$



A CFG Example: Expressions

- V = {E, Op}
- Σ = {identifier, number, (,), +, −, ∗, /}
- P={

    E → E Op E
    E → −E
    E → (E)
    E → number
    E → identifier
    Op → +
    Op → −
    Op → ∗
    Op → /

  }
- S = E

Notes on CFG Notation

- Note: Alternative productions can be merged using |
  - E.g., Op → + | - | * | /
- Several different notations are in use:
  - **Backus-Naur Form (BNF)** uses ::= instead of →
  - **Optional Components notation** N_{opt} means that N is optional in the production
  - **Regular Expressions in RHS notation** allows regular expressions of terminals and nonterminals
- Question: How do we use a grammar?
- We determine whether a program is *derivable* from the grammar

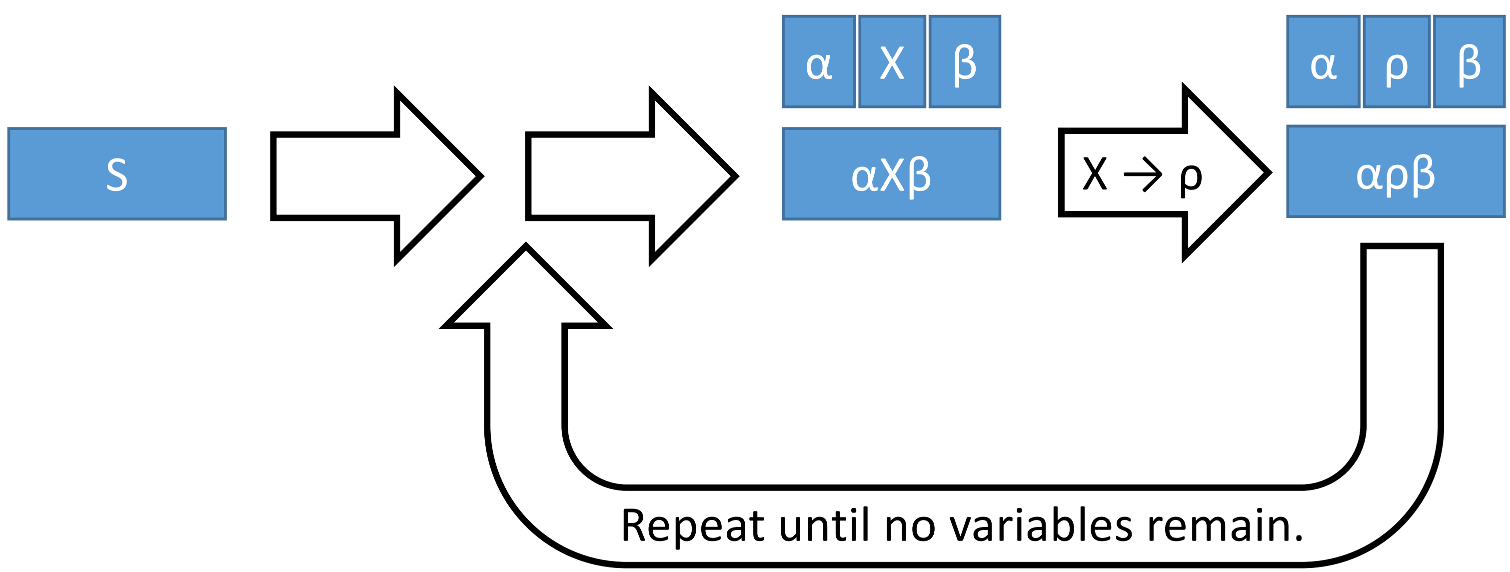
Derivations

- A derivation is a sequence of rewriting operations that starts with the string $\sigma = S$ and then repeats the following until σ contains only terminals:
  - Select a non-terminal in $X \in V$, such that $\sigma = \alpha X \beta$

    where $\alpha, \beta \in (V \cup \Sigma)*$
  - Select a production in $(X \rightarrow \rho) \in P$,
  - Replace X with ρ in the partial derivation σ

    I.e., $\sigma = \alpha \rho \beta$
- Eventually, σ will consist of only terminals, meaning the derivation is complete.

Derivations in a Nutshell



Derivation Example of an Expression

Derive (42 + 13) * 11

σ = **E**

⇒ **E** Op E

⇒ (**E**) Op E

⇒ (**E** Op E) Op E

⇒ (42 **Op** E) Op E

⇒ (42 + **E**) Op E

⇒ (42 + 13) **Op** E

⇒ (42 + 13) ∗ **E**

⇒ (42 + 13) ∗ 11

Grammar

1. *E → E Op E*
2. *E →* **–** *E*
3. *E →* **(** *E* **)**
4. *E → Number*
5. *E → Identifier*
6. *Op → +*
7. *Op →* **–**
8. *Op → /*
9. *Op →* *

Definitions

- Definition: We write $S \Rightarrow^* \sigma$ if there exists a derivation

  $$S \Rightarrow \sigma_1 \Rightarrow \sigma_2 \Rightarrow \ldots \Rightarrow \sigma$$

- Definition: Every grammar G defines a language:

  $$L(G)=\{\sigma \in \Sigma^* \mid S \Rightarrow^* \sigma\}$$

- Definition: If G is a context-free grammar then L(G) is a context-free language.
- Example: What is the language defined by $G = (V, \Sigma, P, S)$
  - $V = \{S\}$
  - $\Sigma = \{0,1,\varepsilon\}$
  - P= {

    $S \rightarrow \varepsilon$

    $S \rightarrow 0\ S\ 1$

    }
  - $S = S$

The language $L(G) = \{0^n1^n | n \geq 0\}$

Example 2

- What is the language defined by G = (V, Σ, P, S)
  - V = {S}
  - Σ = {0,1,ε}
  - P = {

    $S \rightarrow \varepsilon$

    $S \rightarrow 0S0$

    $S \rightarrow 1S1$

    }
  - S = S

The language $L(G) = \{\sigma\sigma^r | \sigma \in \Sigma*\}$
Note: σ^r means reverse of σ

- Observations:
  - These languages are nonregular
  - All regular languages are also context-free languages
  - There are more context-free than regular languages
- Q: How does we represent a derivation?

Parse Trees

- A program is syntactically correct if it can be derived from the grammar of the language it is written in.
- To analyze the program we need a better representation of it.
  - I.e., tokens are the input to the parser
- So, each derivation can be represented by a parse tree.

Structure of Parse Trees

- Root: S, the start nonterminal
- Internal nodes: nonterminals
- Leaf nodes: terminals (called the *yield* of the tree)
- Edge(X,w) : $X \in V$, $w \in \alpha$, where $(X \rightarrow \alpha) \in P$.

Parse Tree Example of an Expression

$\sigma = \mathbf{E}$

$\Rightarrow \mathbf{E}\ \mathrm{Op}\ \mathrm{E}$

$\Rightarrow (\ \mathbf{E}\ )\ \mathrm{Op}\ \mathrm{E}$

$\Rightarrow (\ \mathbf{E}\ \mathrm{Op}\ \mathrm{E}\ )\ \mathrm{Op}\ \mathrm{E}$

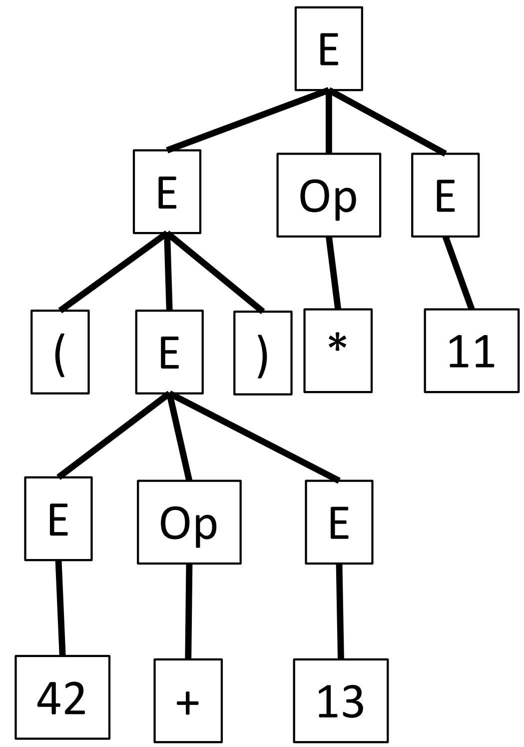
$\Rightarrow (\ 42\ \mathbf{Op}\ \mathrm{E}\ )\ \mathrm{Op}\ \mathrm{E}$

$\Rightarrow (\ 42 + \mathbf{E}\ )\ \mathrm{Op}\ \mathrm{E}$

$\Rightarrow (\ 42 + 13\ )\ \mathbf{Op}\ \mathrm{E}$

$\Rightarrow (\ 42 + 13\ ) \ast \mathbf{E}$

$\Rightarrow (\ 42 + 13\ ) \ast 11$



Another Example: 1 + 2 * 3

This is ambiguous!

σ = **E**

⇒ **E** Op E

⇒ **E** Op E Op E

⇒ 1 **Op** E Op E

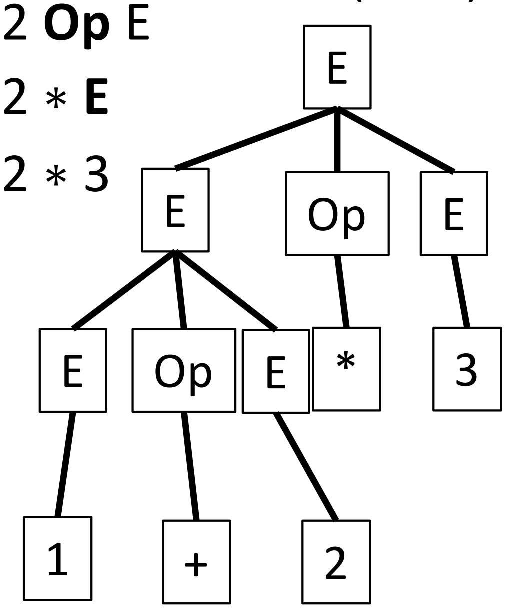
⇒ 1 + **E** Op E

⇒ 1 + 2 **Op** E

⇒ 1 + 2 ∗ **E**

⇒ 1 + 2 ∗ 3

(1 + 2) * 3



σ = **E**

⇒ **E** Op E

⇒ 1 **Op** E

⇒ 1 + **E**

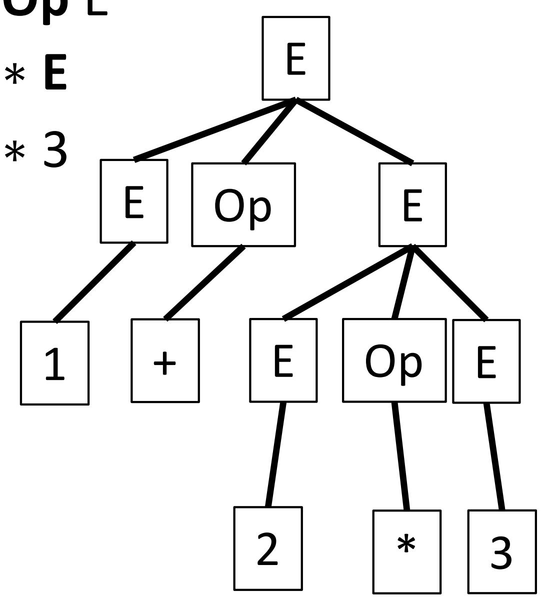
⇒ 1 + **E** Op E

⇒ 1 + 2 **Op** E

⇒ 1 + 2 ∗ **E**

⇒ 1 + 2 ∗ 3

1 + (2 * 3)



Ambiguity

- Observations:
  - There are infinitely many grammars to specify the same language
  - There may be multiple parse trees for the same sentence!
- Definition: If multiple parse trees can be generated by G for the same sentence, then G is *ambiguous*.
- Definition: If L does not have an unambiguous grammar, then L is *inherently ambiguous*
  - Usually not the case for programming languages!

An Unambiguous Expression Grammar

Grammar

- $E \rightarrow T$
- $E \rightarrow E + T$
- $E \rightarrow E - T$
- $T \rightarrow F$
- $T \rightarrow T * F$
- $T \rightarrow T / F$
- $F \rightarrow$ number
- $F \rightarrow$ identifier
- $F \rightarrow (E)$

- Try deriving 1 + 2 * 3