Introduction to Parsing

CSCI 3136: Principles of Programming Languages

Nonregular Languages

- Not all languages are regular.

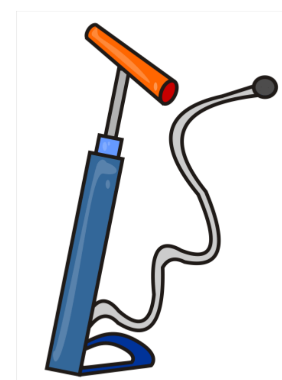
  E.g. L = {0n1n|n >= 0} is not regular.

- Why?

  - Intuition: We need to keep track of how many 0’s we encounter so we can match them with 1's.
  - A DFA has a finite number of states – that is its maximum "memory", so beyond that number we cannot keep track.

- How do we prove this formally?

  The Pumping Lemma!



Finiteness

- The language L = {0i1i | i > 0} is not regular (just believe me for now).
- L is infinite in size as i is not bounded.
- Strings in L can be infinite in length.
- A DFA has a finite number of states.
- Hmmmm ... is this useful?

YES!

Intuition: The Pumping Lemma

- Every Regular Language has a DFA recognizer.
- This recognizer has n states.
- If a string in the language is longer than n, the recognizer has to have a cycle.
- Thus, we can break the string into 3 parts:
  1. characters before the cycle
  2. the cyclic part
  3. characters after the cycle
- "Pumping" means "repeating the cycle"

The Pumping Lemma

For every regular language L, there exists a constant n such that every σ E L, where |σ| ≥ n, can be divided into three substrings σ = αβγ with the following properties:

- |αβ|≤n
- |β| > 0, and
- αβkγ E L, ΄I k≥0

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

The Pumping Lemma

1. n is related to the # of states in the recognizing DFA.

2. $\sigma = \alpha\beta\gamma$ if $|\sigma| \geq n$

   α is the part before the cycle

   β is the cyclic part (possibly with more than one iteration)

   γ is the part after the cycle

3. The properties mean:

   - $|\alpha\beta| \leq n$ – getting to and through the cycle once can't use more than n states and hence can't have more characters

Applying the Pumping Lemma

Procedure: To show that L is not regular

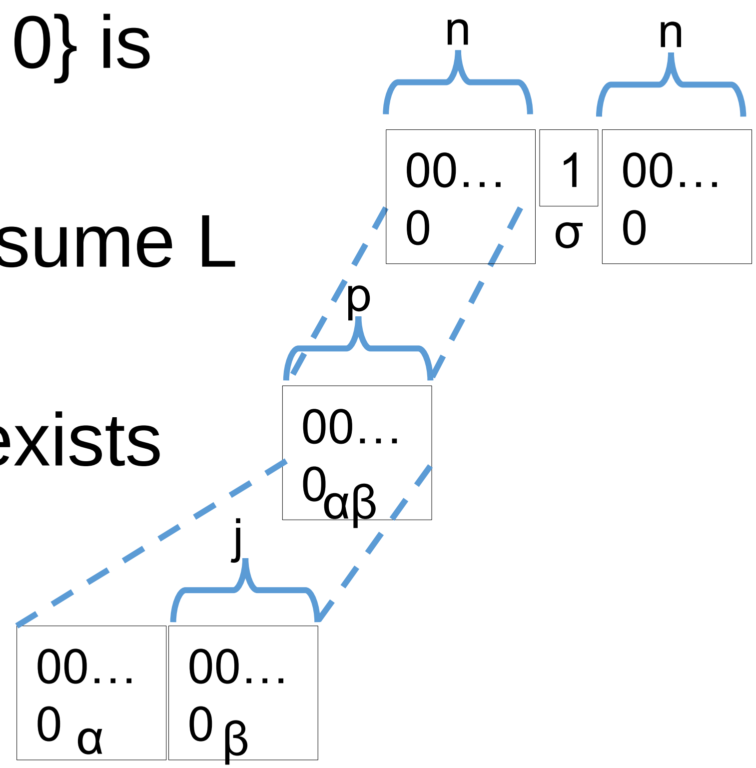
- Convince yourself L is not regular (intuition)
- Assume that L is regular and that there is a constant n as stated by the Pumping Lemma
- Select σ E L such that
  - |σ| > n
  - σ = αβγ, for all α and β
    - |αβ| ≤ n
    - |β| > 0
  - There exists a j ≥ 0 such that αβjγ Z L

HARD PART

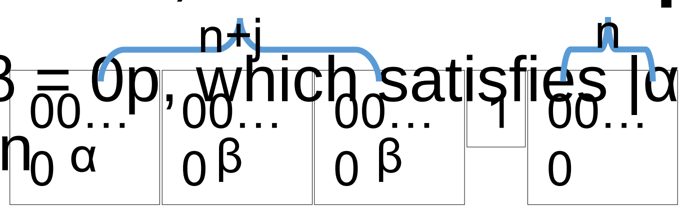
Example: Use the Pumping Lemma

Intuition: DFA can only keep track of n + 1 things (at most).

Show that L = {0m10m|m ≥ 0} is not regular.



- Proof by contradiction: Assume L is regular.
- If L is regular, then there exists an n, by the PL
- Select σ = 0n10n
- Therefore, **for all α and β**
  - αβ = 0p, which satisfies |αβ|≤ n as p = n



  - β = 0j, 0 < j ≤ n

Or in other words ...

- $L = \{0m10m|m \geq 0\}$
- So consider a small string for now ...

0000001000000

IF:

1. The cycle is before the 1, more cycles make a string with more 0's before the 1 ... it can't be there.
2. If the cycle is after the 1, more cycles make a string with more 0's after the 1 ... it can't be there
3. if the cycle contains the one, more cycles make a string with more 1's ... it can't be there.

Using the Pumping Lemma is like an Argument with the Devil

I think L is regular.

I don’t.

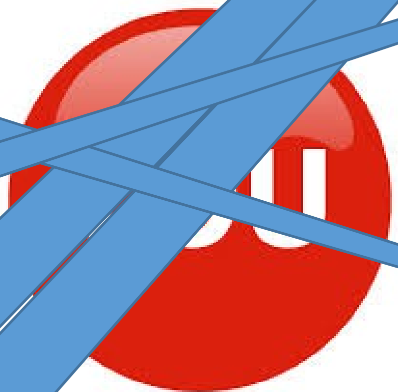
I choose n.

I choose □□ L

I get to divide σ into αβγ

I choose j such that αβjγ Z L

Hence, L is not regular!
I win



Examples

- L = {aibj|i < j}
- L = {ap|p is prime}
- L = {aibj | i = j mod 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

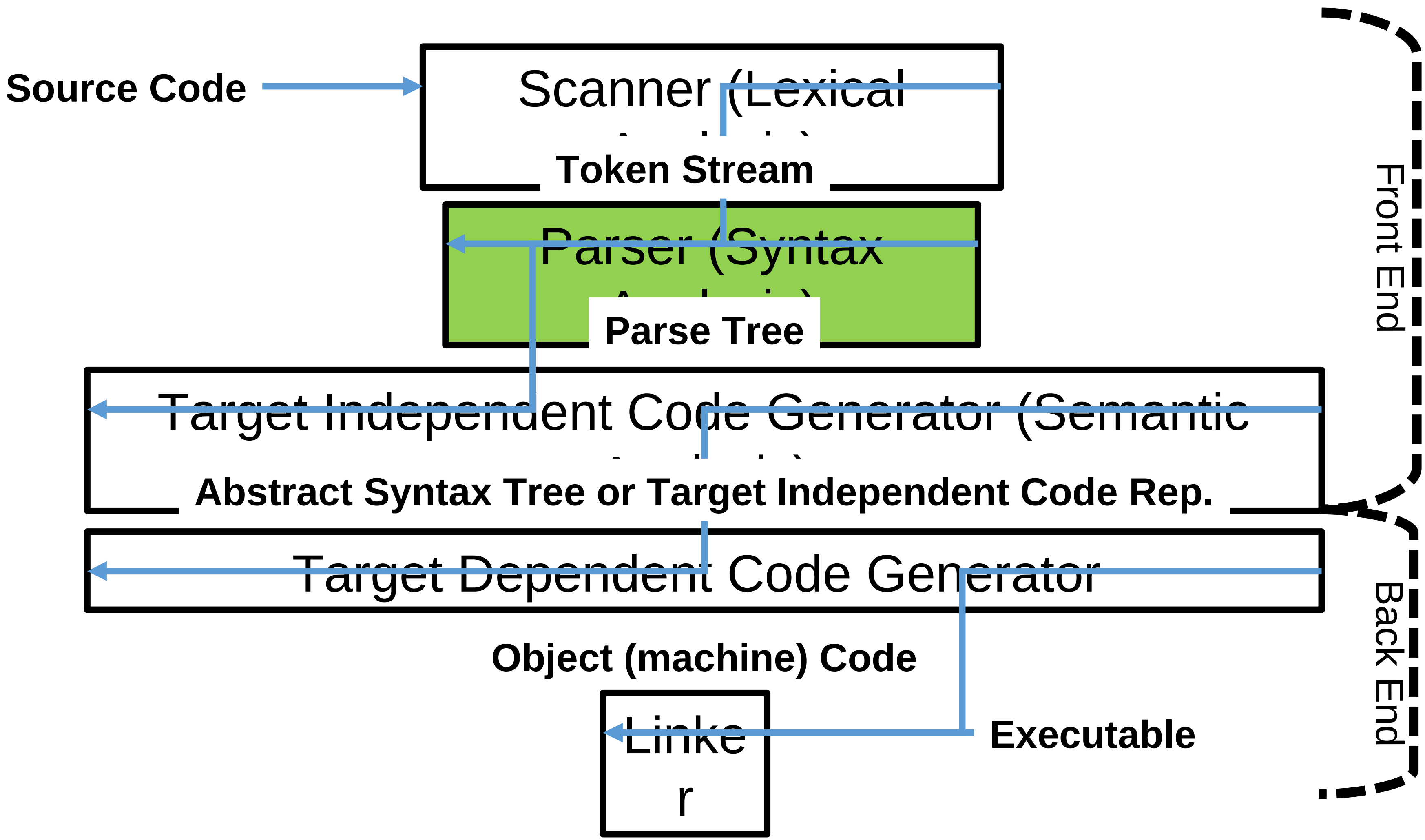
Why Do We Need a Parser?

- 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) X (3 + 2))

  Think of this as (i)i , i > 0, which isn't regular as we know.
- Scanners are useful for
  - Checking if program’s tokens are correct.

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.`

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 tokens returned by the scanner
- P is the set of productions
  - Of the form N → (Σ ν V)*, N E V
  - Also known as “Rewriting Rules”
- S is the start symbol, S E V

A CFG Example: Expressions

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

  E → E Op E

  E → −E

  E → (E)

  E → number

  E → identifier

  Op → +

  Op → −

  Op → X

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** Nopt 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*

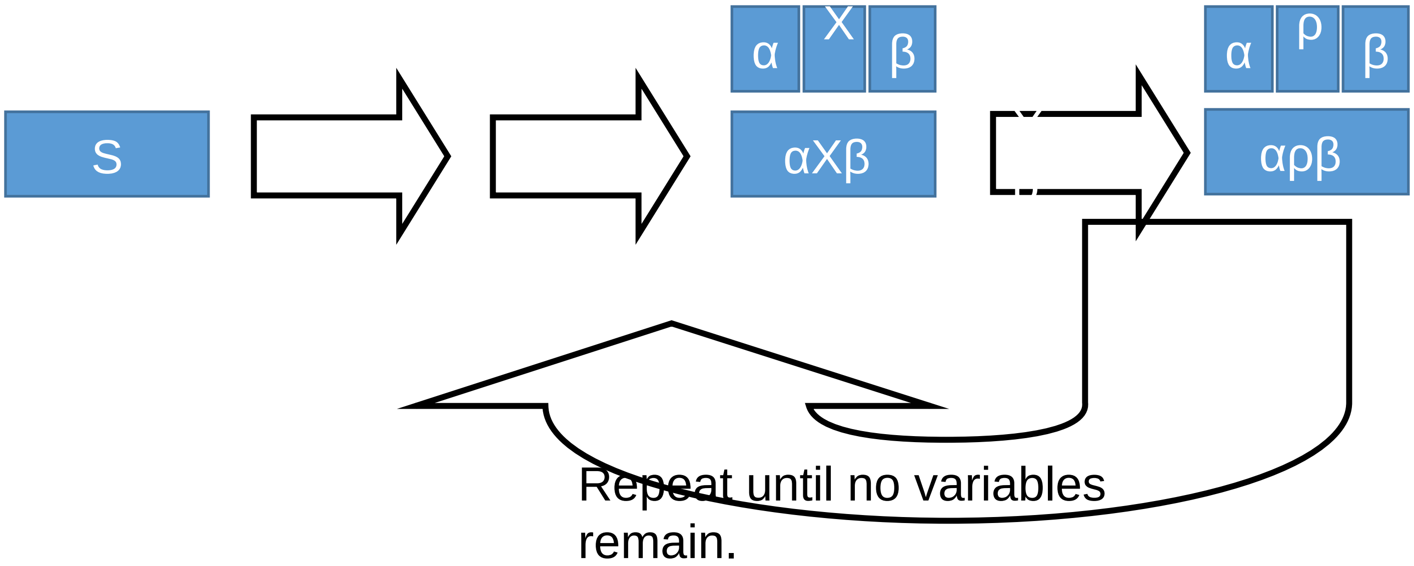
Derivations

- A derivation is a sequence of rewriting operations that starts with the string σ = S and then repeats the following until σ contains only terminals:
  - Select a non-terminal in XEV, such that σ = αXβ

    where α,βE(V νΣ)X
  - Select a production in (X → ρ)EP,
  - Replace X with ρ in the partial derivation σ

    I.e., σ = αρβ
- Eventually, σ will consist of only terminals, meaning the derivation is complete.

Derivations in a Nutshell



Derivation Example of an Expression

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) X **E**

□ (42 + 13) X 11

Grammar

1. *E* → *E Op E*
2. *E* → - *E*
3. *E* → (*E*)
4. *E* → *Number*
5. *E* → *Identifier*
6. *Op* → +

Definitions

- Definition: We write S □ * σ if there exists a derivation

  S □ σ1 □ σ2 □ ...□ σ

- Definition: Every grammar G defines a language:

  L(G)={σ E ΣX | S □ X σ}

- Definition: If G is a context-free grammar then L(G) is a context-free language.

  The language L(G) = {0n1n|n >0}

- Example: What is the language defined by G = (V, Σ, P, S)

  V = {S}

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) = {σσr|σ E ΣX}

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) : XEV, wEα, where (X → α)EP.

Parse Tree Example of an Expression

σ = **E**

□ **E** Op E

□ (**E**) Op E

□ (**E** Op E) Op E

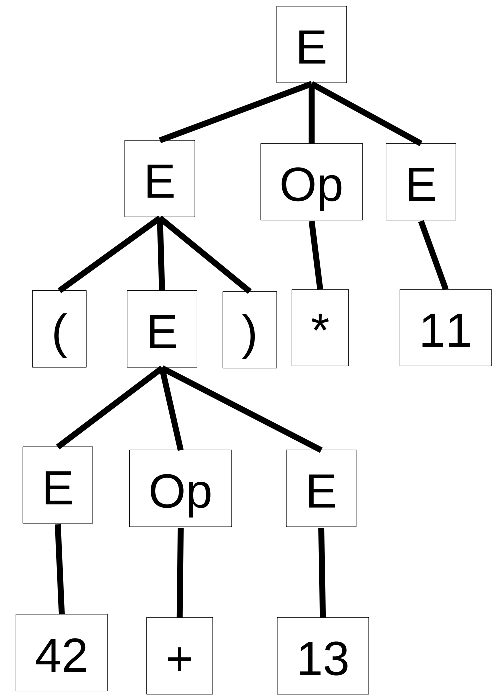
□ (42 **Op** E) Op E

□ (42 + **E**) Op E

□ (42 + 13) **Op** E

□ (42 + 13) X **E**

□ (42 + 13) X 11



Another Example: 1 + 2 * 3

This is ambiguous!

σ = **E**

□ **E** Op E

□ **E** Op E Op E

□ 1 **Op** E Op E

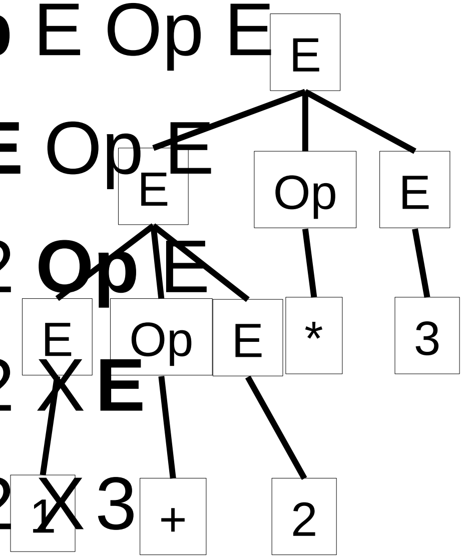
□ 1 + **E** Op E

□ 1 + 2 **Op** E

□ 1 + 2 X **E**

□ 1 + 2 X 3

(1 + 2) * 3



σ = **E**

□ **E** Op E

□ 1 **Op** E

□ 1 + **E**

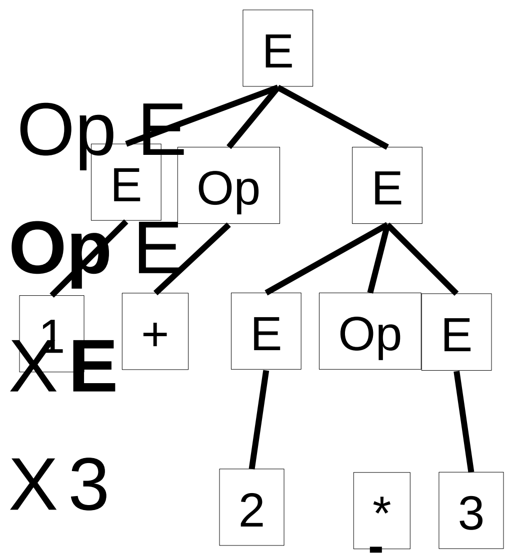
□ 1 + **E** Op E

□ 1 + 2 **Op** E

□ 1 + 2 X **E**

□ 1 + 2 X 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

- $E \rightarrow T$
- $E \rightarrow E + T$
- $E \rightarrow E - T$
- $T \rightarrow F$
- $T \rightarrow T \times F$
- $T \rightarrow T / F$
- $F \rightarrow$ number
- $F \rightarrow$ identifier

- Try deriving 1 + 2 + 3