# Grammars & Parsing

Lecture 12

CS 2112 – Fall 2018

# Motivation

The cat ate the rat.

The cat ate the rat slowly.

The small cat ate the big rat slowly.

The small cat ate the big rat on the mat slowly.

The small cat that sat in the hat ate the big rat on the mat slowly.

The small cat that sat in the hat ate the big rat on the mat slowly, then got sick.

…

- Not all sequences of words are legal sentences
  - The ate cat rat the
- How many legal sentences are there?
- How many legal programs are there?
- Are all Java programs that compile legal programs?
- How do we know what programs are legal?

http://java.sun.com/docs/books/jls/third_edition/html/syntax.html

# A Grammar

Sentence ::= Noun Verb Noun
Noun ::= boys | girls | bunnies
Verb ::= like | see

- Our sample grammar has these rules:
  - A Sentence can be a Noun followed by a Verb followed by a Noun
  - A Noun can be 'boys' or 'girls' or 'bunnies'
  - A Verb can be 'like' or 'see'
- Examples of Sentence:
  - boys see bunnies
  - bunnies like girls
  - …

- Grammar: set of rules for generating sentences in a language
- White space between words does not matter
- The words boys, girls, bunnies, like, see are called *tokens* or *terminals*
- The words Sentence, Noun, Verb are called *syntactic classes* or *nonterminals*
- This is a very boring grammar because the set of Sentences is finite (exactly 18)

# A Recursive Grammar

Sentence ::= Sentence and Sentence
| Sentence or Sentence
| Noun Verb Noun

Noun ::= boys | girls | bunnies

Verb ::= like | see

- This grammar is more interesting than the last one because the set of Sentences is infinite

- Examples of Sentences in this language:
  - boys like girls
  - boys like girls and girls like bunnies
  - boys like girls and girls like bunnies and girls like bunnies
  - boys like girls and girls like bunnies and girls like bunnies and girls like bunnies
  - ...

- What makes this set infinite? Answer:
  - Recursive definition of Sentence

# Detour

- What if we want to add a period at the end of every sentence?

Sentence ::= Sentence and Sentence .

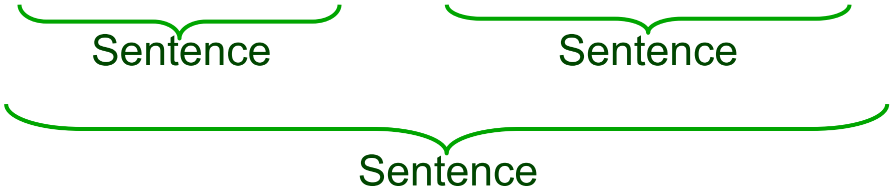
| Sentence or Sentence .

| Noun Verb Noun .

Noun ::= . . .

- Does this work?
- No! This produces sentences like:

girls like boys . and boys like bunnies . .



# Sentences with Periods

TopLevelSentence ::= Sentence .

Sentence ::= Sentence and Sentence
| Sentence or Sentence
| Noun Verb Noun

Noun ::= boys | girls | bunnies

Verb ::= like | see

- Add a new rule that adds a period only at the end of the sentence.
- The tokens here are the 7 words plus the period (.)
- This grammar is ambiguous:

  boys like girls and girls like

  boys or girls like bunnies

# Grammar for Simple Expressions

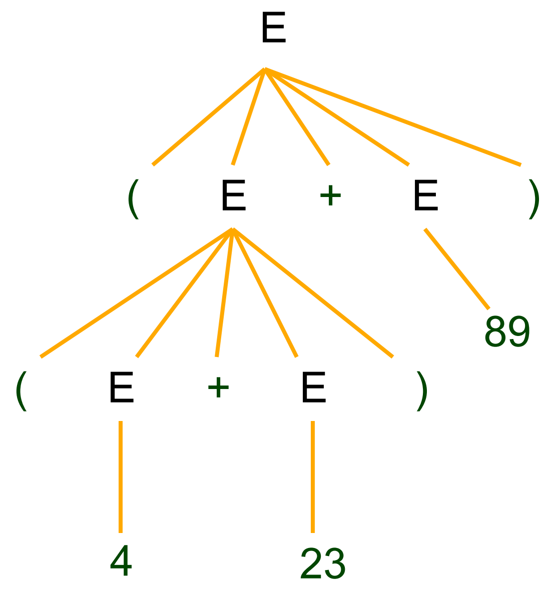
E ::= integer | ( E + E )

- Simple expressions:
  - An E can be an integer.
  - An E can be '(' followed by an E followed by '+' followed by an E followed by ')'
- Set of expressions defined by this grammar is an inductively-defined set
  - Is the language finite or infinite?
  - Do recursive grammars always yield infinite languages?
- Here are some legal expressions:
  - 2
  - (3 + 34)
  - ((4+23) + 89)
  - ((89 + 23) + (23 + (34+12)))
- Here are some illegal expressions:
  - (3
  - 3 + 4
- The *tokens* in this grammar are (, +, ), and any integer

# Parsing

- Grammars can be used in two ways
  - A grammar defines a *language* (i.e., the set of properly structured *sentences*)
  - A grammar can be used to *parse a sentence* (thus, checking if the *sentence* is in the *language*)
- To *parse* a sentence is to build a *parse tree*
  - This is much like *diagramming a sentence*

- Example: Show that ((4+23) + 89) is a valid expression E by building a *parse tree*

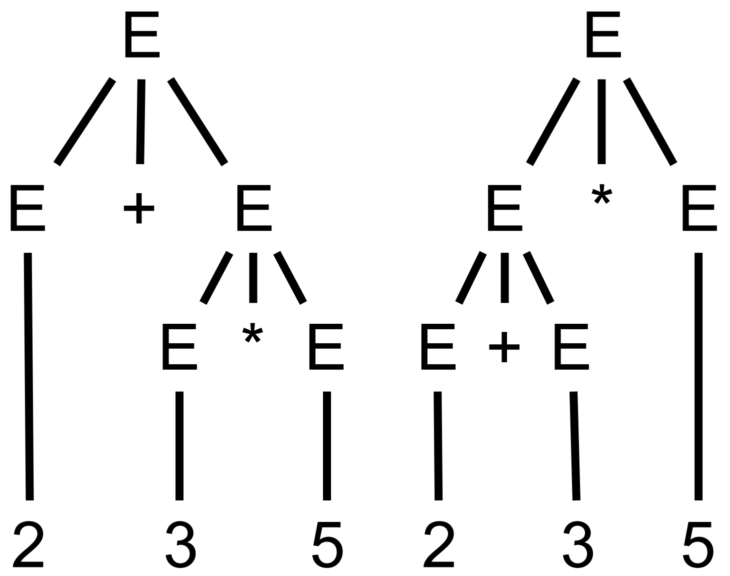


# Ambiguity

- Grammar is **ambiguous** if some strings have more than one parse tree
- Example: arithmetic expressions without precedence:

$$E \rightarrow n \mid E + E \mid E * E \mid ( E )$$

2 + 3 * 5



# Precedence

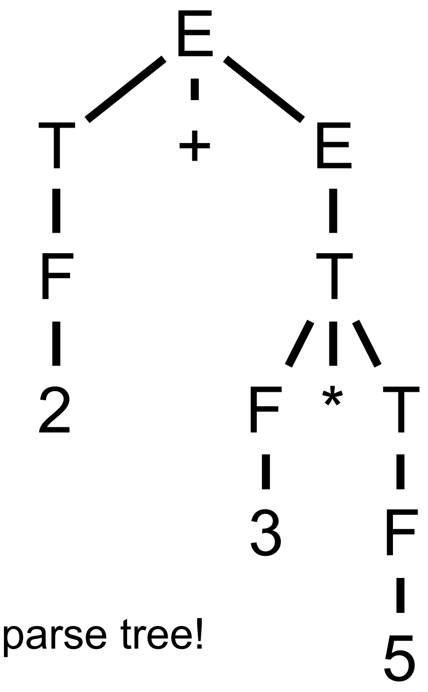
- Ambiguities resulting from not handling precedence can be handled by introducing extra levels of nonterminals.

E (expr) → T | T + E

T (term) → F | F * T

F (factor) → *n* | ( E )

2 + 3 * 5




Only one parse tree!

# Recursive Descent Parsing

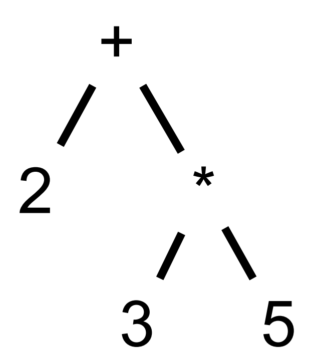
- Idea: Use the grammar to design a *recursive program* to check if a sentence is in the language
- To parse an expression E, for instance
  - We look for each terminal (i.e., each *token*)
  - Each nonterminal (e.g., E) can handle itself by using a *recursive call*
- The grammar tells how to write the program!
- A **recognizer**:

```
boolean parseE( ) {
  if (first token is an integer) return true;
  if (first token is '(') {
    scan past '(' token;
    parseE( );
    scan past '+' token;
    parseE( );
    scan past ')' token;
    return true;
  }
  return false; }
```

# Abstract Syntax Trees vs. Parse Trees

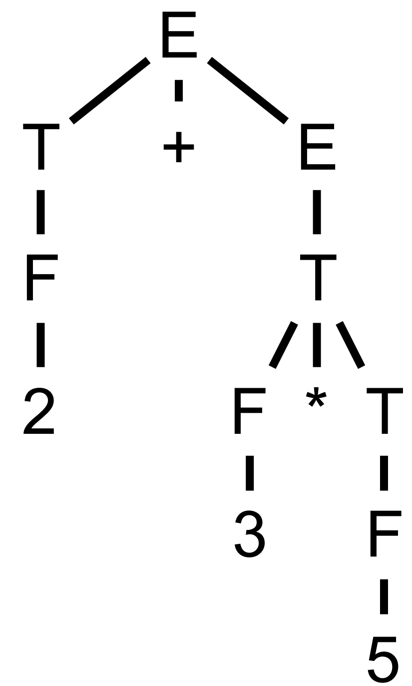
- Result of parsing: often a data structure representing the input.
- Parse tree has information we don't need, e.g. parentheses.

**Abstract syntax tree**




```
new BinaryOp(TIMES,
  new BinaryOp(PLUS,
               new Num(2),
               new Num(3)),
  new Num(5))
```

**Parse tree / concrete syntax tree**



# Java Code for Parsing E

```
public static ExprNode parseE(Scanner scanner) {
    if (scanner.hasNextInt()) {
        int data = scanner.nextInt();
        return new Node(data);
    }
    check(scanner, '(');
    left = parseE(scanner);
    check(scanner, '+');
    right = parseE(scanner);
    check(scanner, ')');
    return new BinaryOpNode(PLUS, left, right);
}
```

# Responding to Invalid Input

- Parsing does two things:
  - checks for validity (is the input a valid sentence?)
  - constructs the parse tree (usually called an AST or abstract syntax tree)
- Q: How should we respond to invalid input?
- A: Throw an exception with as much information for the user as possible
  - the nature of the error
  - approximately where in the input it occurred

# The associativity problem

- Top-down parsing works well with **right-recursive** grammars (e.g.,

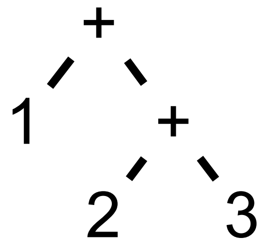
E (expr) → T | T + E

T (term) → F | F * T

F (factor) → *n* | ( E )

- Problem: leads to right-associative operators:

- 1 + 2 + 3 :



# Reassociation

- Trick: rewrite right-recursive rules to use *Kleene star*:

  E (expr) → T | T + E

  *becomes*

  E → T (+ T)* <--- "0 or more repetitions of + T"

- Recursion becomes a loop:

```
public static Expr parseE() {
    Expr e = parseT();
    while (peek() is "+")) {
        consume("+");
        e = new BinaryOpNode(PLUS, e, parseT());
    }
    return e;
}
```

# Using a Parser to Generate Code

- We can modify the parser so that it generates stack code to evaluate arithmetic expressions:

| | |
|---|---|
| 2 | PUSH 2<br>STOP |
| (2 + 3) | PUSH 2<br>PUSH 3<br>ADD<br>STOP |

- Goal: Modify parseE to return a string containing stack code for expression it has parsed

- Method parseE can generate code in a recursive way:
  - For integer i, it returns string "PUSH " + i + "\n"
  - For (E1 + E2),
    - Recursive calls for E1 and E2 return code strings c1 and c2, respectively
    - Return c1 + c2 + "ADD\n"
  - Top-level method appends a STOP command

# Does Recursive Descent Always Work?

- No – some grammars cannot be used with recursive descent
  - A trivial example (causes infinite recursion):

    S ::= b | Sa

- Can rewrite grammar

    S ::= b | bA

    A ::= a | aA

- Sometimes recursive descent is hard to use
  - There are more powerful parsing techniques (not covered in this course)

- Nowadays, there are automated parser and tokenizer generators
  - you write down the grammar, it produces the parser and tokenizer automatically
  - Many based on *LR parsing*, which can handle a larger class of grammars.

# Exercises

Write a grammar and recursive-descent parser for

- palindromes:

  mom        dad        I prefer pi        race car

  A man, a plan, a canal: Panama

  murder for a jar of red rum        sex at noon taxes

- strings of the form $A^nB^n$ for some $n \geq 0$:

  AB        AABB        AAAAAABBBBBB

- Java identifiers:

  a letter, followed by any number of letters or digits

- decimal integers:

  an optional minus sign (–) followed by one or more digits 0-9