



**HARVARD**

John A. Paulson  
School of Engineering  
and Applied Sciences

# **CS153: Compilers**

## **Lecture 4: Recursive Parsing**

Stephen Chong

<https://www.seas.harvard.edu/courses/cs153>

# Announcements

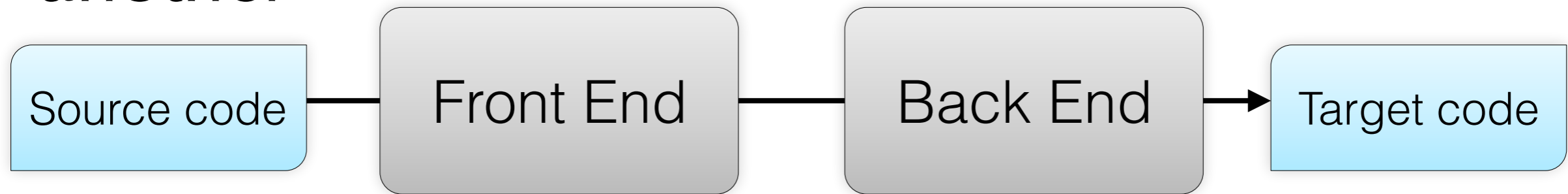
- Proj 1 out
  - Due Thursday Sept 20 11:59
  - Start soon if you haven't!
- Proj 2 released today
  - We will email you link to repo
  - Due three weeks: Thursday Oct 4, 11:59pm.
  - README on the course website

# Today

- Parsing
  - Context-free grammars
  - Derivations
  - Parse trees
  - Ambiguous grammars
  - Recursive descent parsing
  - Parser combinators

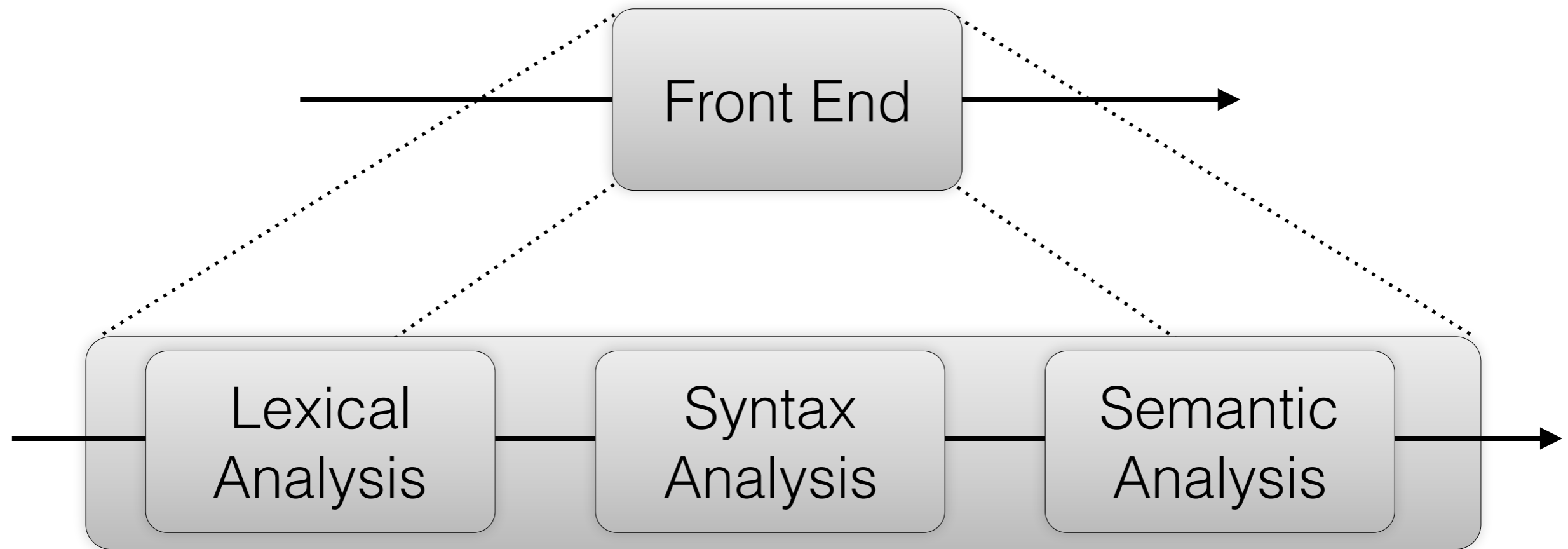
# Lexing and Parsing

- Compiler translates from one language to another



- Front end: Analysis
  - pulls apart program, understand structure and meaning
- Back end: Synthesis
  - puts it back together in a different way

# Lexing and Parsing



- **Lexical analysis:** breaks input into individual words, aka “tokens”
- **Syntax analysis:** parses the phrase structure of program
- **Semantic analysis:** calculates meaning of program

# Parsing

- Two pieces conceptually:
  - Recognizing syntactically valid phrases.
  - Extracting semantic content from the syntax.
    - E.g., What is the subject of the sentence?
    - E.g., What is the verb phrase?
    - E.g., Is the syntax ambiguous? If so, which meaning do we take?
      - ▶ “Time flies like an arrow”, “Fruit flies like a banana”
      - ▶ “2 \* 3 + 4”
      - ▶ “x ^ f y”
- In practice, solve both problems at the same time.

# Specifying the Language

- A language is a set of strings. We need to specify what this set is.
- Can we use regular expressions?
- In *MLLex*, we named regular expressions e.g.,
  - `digits = [0-9]+`
  - `sum = (digits "+")* digits`
  - Defines sums of the form `4893 + 48 + 92`
- But what if we wanted to add parentheses to the language?
  - `digits = [0-9]+`
  - `sum = expr "+" expr`
  - `expr = digits | "(" sum ")"`

# Specifying the Language

- It's impossible for finite automaton to recognize language with balanced parentheses!
- MLLex just treats digits as an abbreviation of the regex `[0-9]+`
  - This doesn't add expressive power to the language
- Doesn't work for example above: try expanding the definition of `sum` in `expr`:
  - `expr = digits | "(" sum ")"`
  - `expr = digits | "(" expr "+" expr ")"`
  - But `expr` is an abbreviation, so we expand it and get
  - `expr = digits |  
 "(" (digits | "(" expr "+" expr ")")  
 "+" (digits | "(" expr "+" expr ")") ")"`
  - Uh oh...



# Context-Free Grammars

- Additional expressive power of recursion is exactly what we need!
- Context Free Grammars (CFGs) are regular expressions with recursion
- CFGs provide declarative specification of syntactic structure
- CFG has set of **productions** of the form  
 $symbol \rightarrow symbol\ symbol\ \dots\ symbol$   
with zero or more symbols on the right
- Each symbol is either **terminal** (i.e., token from the alphabet) or **non-terminal** (i.e., appears on the LHS of some production)
  - No terminal symbols appear on the LHS of productions

# CFG example

$$\begin{array}{lll} S \rightarrow S; S & E \rightarrow \text{id} & L \rightarrow E \\ S \rightarrow \text{id} := E & E \rightarrow \text{num} & L \rightarrow L, E \\ S \rightarrow \text{print} ( L ) & E \rightarrow E + E & \\ & E \rightarrow ( S, E ) & \end{array}$$

- Terminals are: `id print num , + ( ) := ;`
- Non-terminals are: `S, E, L`
  - `S` is the start symbol
- E.g., one sentence in the language is  
`id := num; id := (id := num+num, id+id)`
  - Source text (before lexical analysis) might have been  
`a := 7; b := (c := 30+5, a+c)`

# Derivations

- To show that a sentence is in the language of a grammar, we can perform a **derivation**
  - Start with start symbol, repeatedly replace a non-terminal by its right hand side

• E.g.,

•  $S$

•  $S; S$

•  $id := E; S$

•  $id := E; id := E$

•  $id := num; id := E$

• ...

•  $id := num; id := (id := num+num, id+id)$

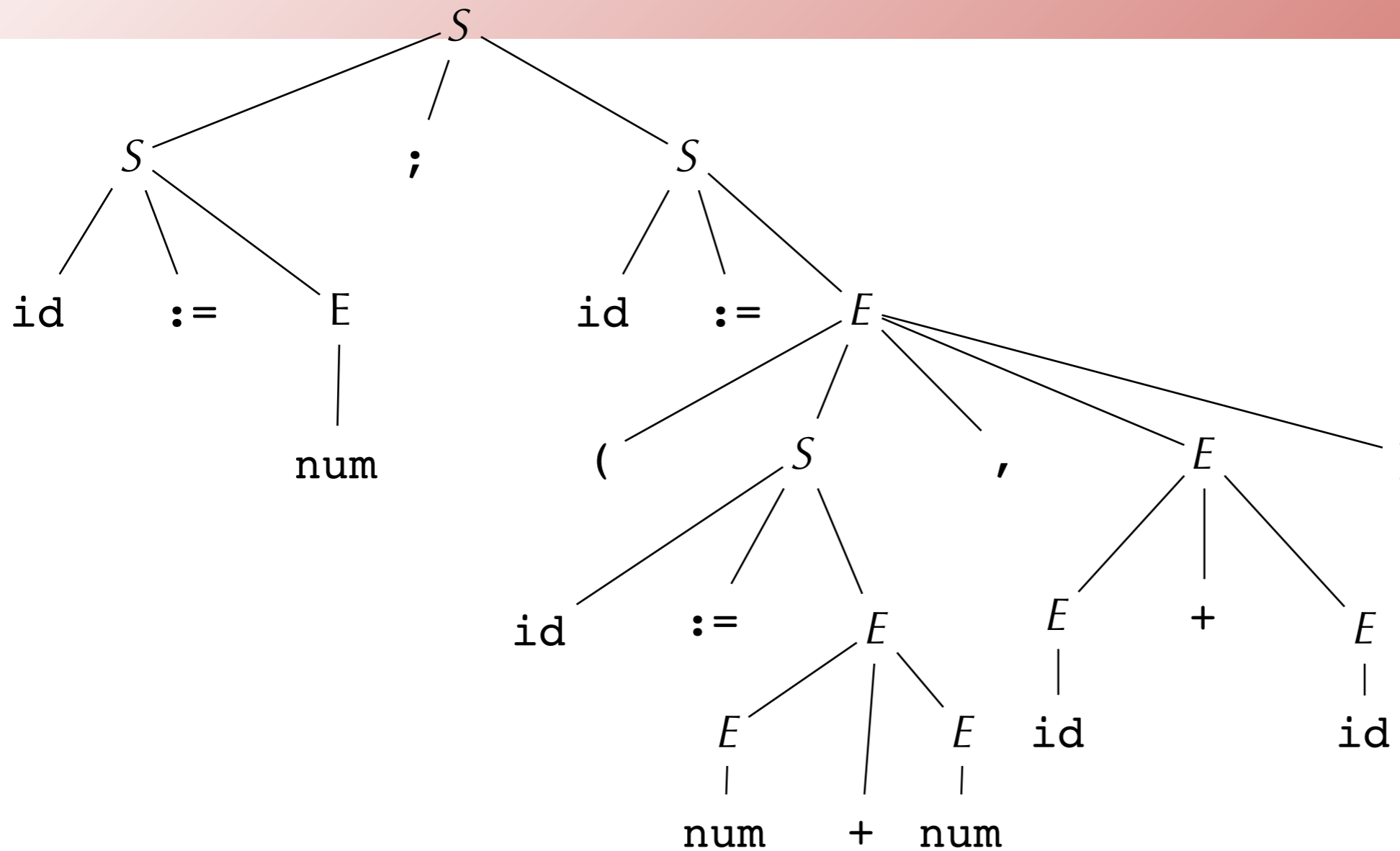
$S \rightarrow S; S$   
 $S \rightarrow id := E$   
 $S \rightarrow print(L)$   
 $E \rightarrow id$   
 $E \rightarrow num$   
 $E \rightarrow E + E$   
 $E \rightarrow (S, E)$   
 $L \rightarrow E$   
 $L \rightarrow L, E$

# CFGs and Regular Expressions

- CFGs are strictly more expressive than regular expressions

How can you translate a regular expression into a CFG?

# Parse Tree

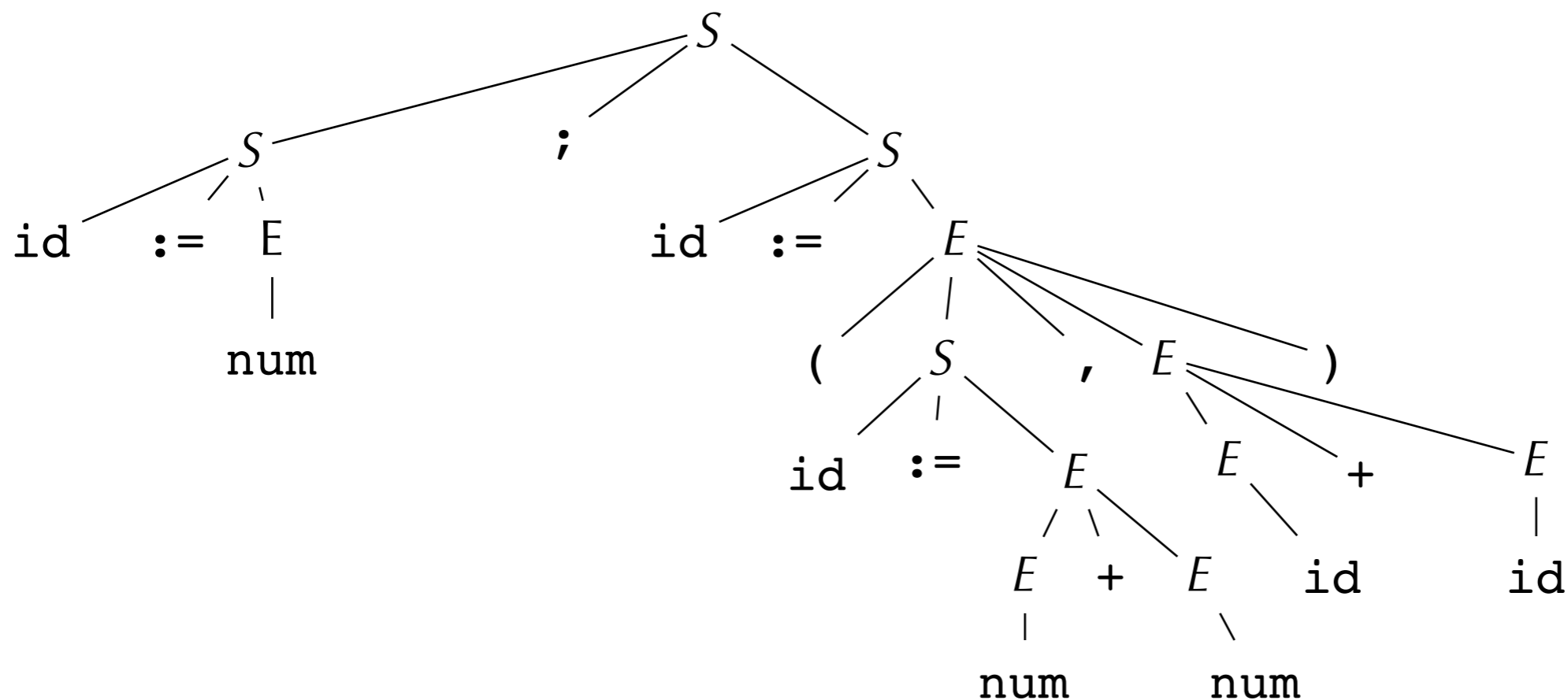


$S \rightarrow S; S$   
 $S \rightarrow id := E$   
 $S \rightarrow print ( L )$   
 $E \rightarrow id$   
 $E \rightarrow num$   
 $E \rightarrow E + E$   
 $E \rightarrow ( S, E )$   
 $L \rightarrow E$   
 $L \rightarrow L, E$

- A **parse tree** connects each symbol to the symbol it was derived from
- A derivation is, in essence, a way of constructing a parse tree.
  - Two different derivations may have the same parse tree

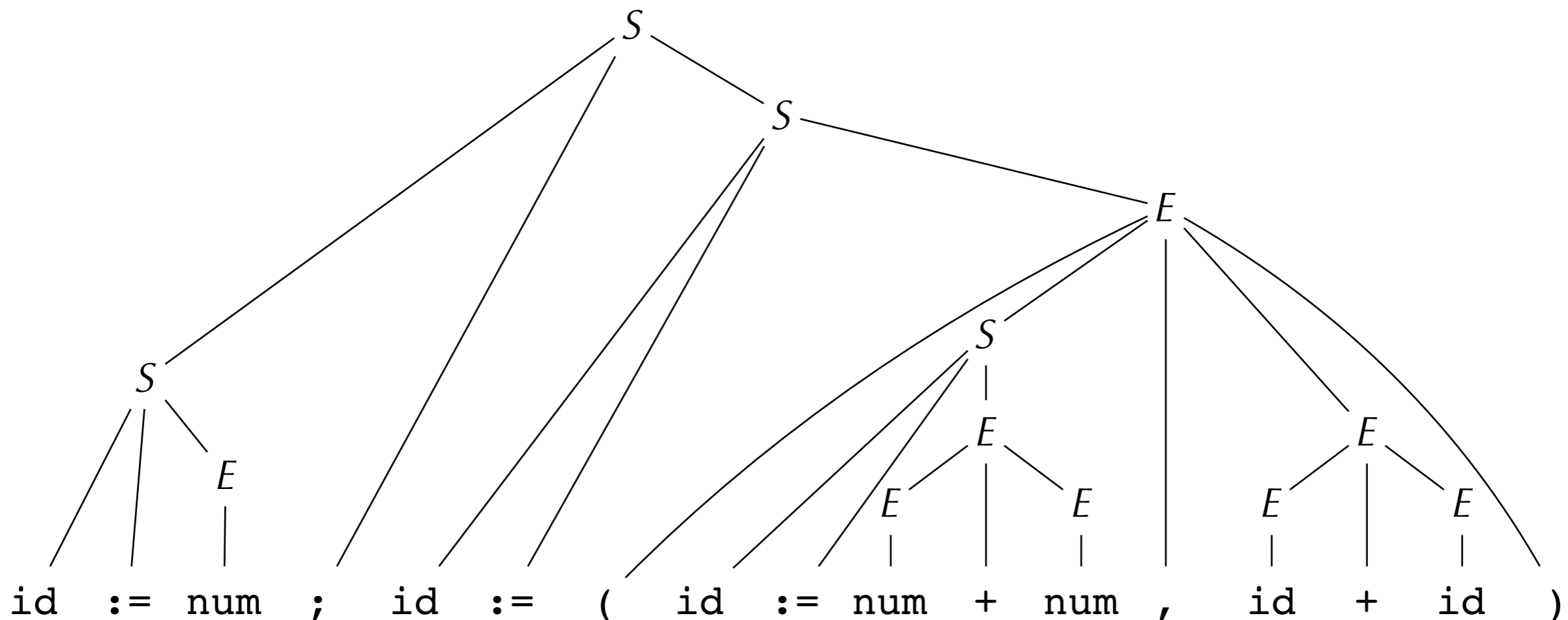
# How to Build a Parse Tree/ Find a Derivation

- Conceptually, two possible ways:
  - Start from start symbol, choose a non-terminal and expand until you reach the sentence
  - Start from the terminals and replace phrases with non-terminals



# How to Build a Parse Tree/ Find a Derivation

- Conceptually, two possible ways:
  - Start from start symbol, choose a non-terminal and expand until you reach the sentence
  - Start from the terminals and replace phrases with non-terminals

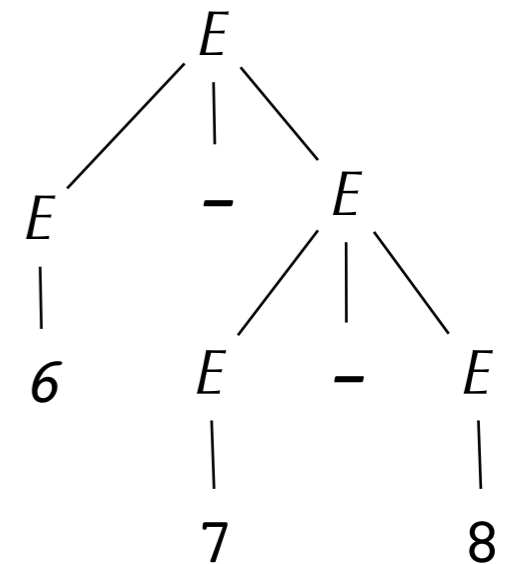
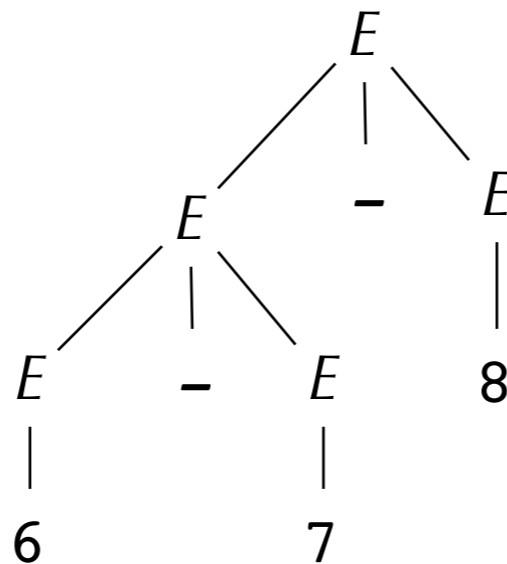


# Ambiguous Grammar

- A grammar is **ambiguous** if it can derive a sentence with two different parse trees

• E.g.,

$E \rightarrow \text{id}$   
 $E \rightarrow \text{num}$   
 $E \rightarrow E * E$   
 $E \rightarrow E / E$   
 $E \rightarrow E + E$   
 $E \rightarrow E - E$   
 $E \rightarrow (E)$



- Ambiguity is usually bad: different parse trees often have different meaning!
- But we can usually eliminate ambiguity by transforming the grammar



# Fixing Ambiguity Example

- We would like  $*$  to **bind higher** than  $+$   
(aka,  $*$  to have **higher precedence** than  $+$ )
  - So  $1+2*3$  means  $1+(2*3)$  instead of  $(1+2)*3$
- We would like each operator to **associate to the left**
  - So  $6-7-8$  means  $(6-7)-8$  instead of  $6-(7-8)$
- Symbol  $E$  for expression,  $T$  for term,  $F$  for factor

$$\begin{array}{lll} E \rightarrow E + T & T \rightarrow T * F & F \rightarrow \text{id} \\ E \rightarrow E - T & T \rightarrow T / F & F \rightarrow \text{num} \\ E \rightarrow T & T \rightarrow F & F \rightarrow (E) \end{array}$$

# How to Parse

- Manual, (recursive descent)

Top down

- Easy to write
- Good error messages
- Tedious, hard to maintain

- Parsing Combinators

- Encode grammars as higher-order functions
- Essentially, functions generate a recursive descent parser

- Antlr <http://www.antlr.org/>

- Yacc

Bottom up

- ...

# Recursive Descent

- See file `Lec04-recdesc-a.ml`
- Try the following:
  - `exp_parse "32"`
  - `exp_parse "let foo = 7 in 42"`
  - `exp_parse "let foo = 7 let bar"`

# Recursive Descent

- See file `Lec04-recdesc-b.ml`
- More direct implementation of grammar

$$E \rightarrow E + T$$

$$T \rightarrow T * F$$

$$F \rightarrow \text{id}$$

$$E \rightarrow E - T$$

$$T \rightarrow T / F$$

$$F \rightarrow \text{num}$$

$$E \rightarrow T$$

$$T \rightarrow F$$

$$F \rightarrow (E)$$

- Each non-terminal is a function

# Left Recursion

- Recursive descent parsing doesn't handle left recursion well!
- We can refactor grammar to avoid left recursion
- E.g., transform left recursive grammar

$$\begin{array}{lll} E \rightarrow E + T & T \rightarrow T * F & F \rightarrow \text{id} \\ E \rightarrow E - T & T \rightarrow T / F & F \rightarrow \text{num} \\ E \rightarrow T & T \rightarrow F & F \rightarrow (E) \end{array}$$

to

$$\begin{array}{lll} E \rightarrow T E' & T \rightarrow F T' & F \rightarrow \text{id} \\ E' \rightarrow + T E' & T' \rightarrow * F T' & F \rightarrow \text{num} \\ E' \rightarrow - T E' & T' \rightarrow / F T' & F \rightarrow (E) \\ E' \rightarrow & T' \rightarrow & \end{array}$$

# Left Recursion

- See file `Lec04-recdesc-c.ml`
- Try the following:
  - `exp_parse "6 - 7 - 8";;` *Observe the left associativity*

# Parser Combinators

- Parser combinators are an elegant functional-programming technique for parsing
  - Higher-order functions that accept parsers as input and returns a new parser as output
- That's what our code already is!