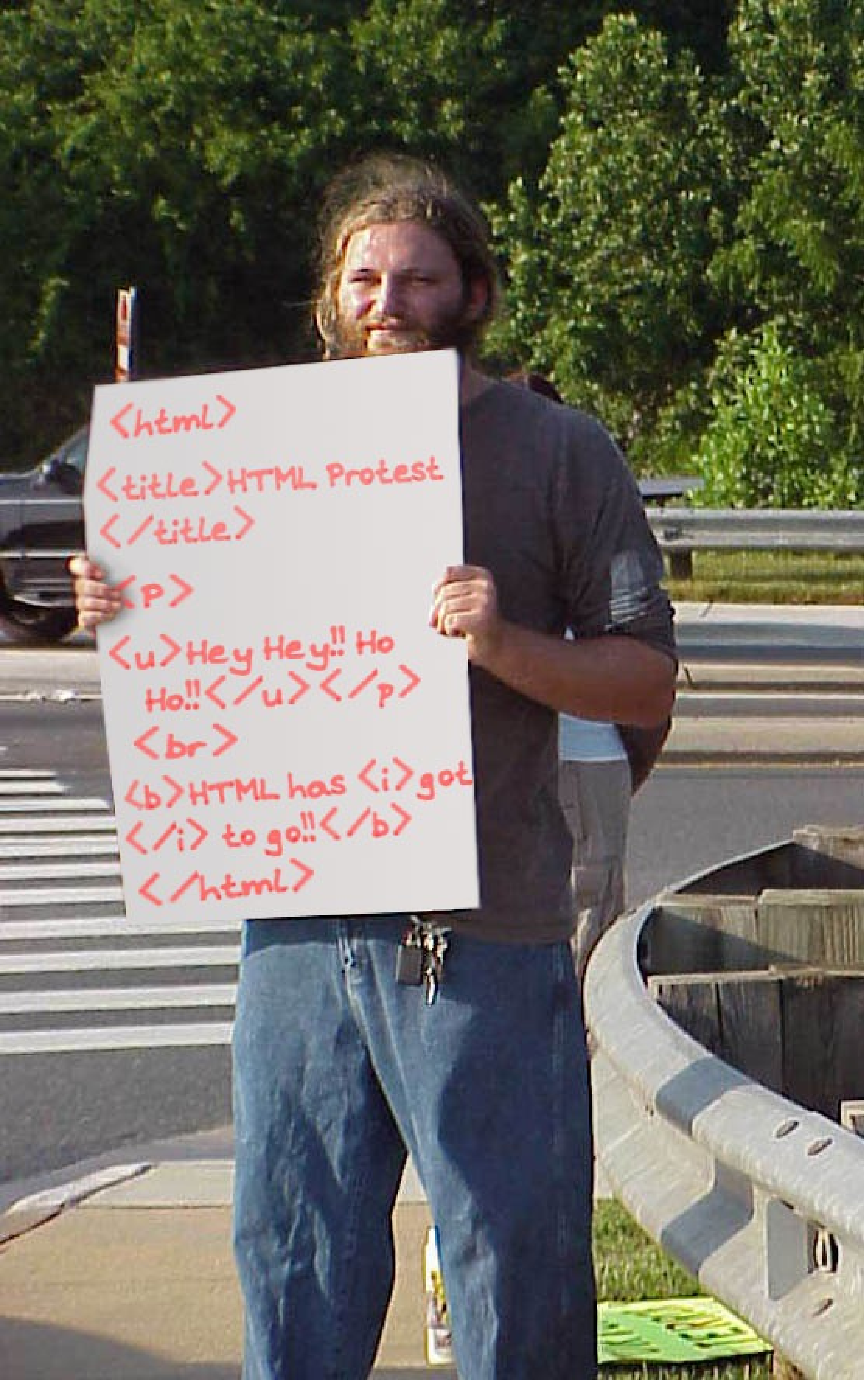


Intro To Parsing

Step By Step

Self-Test Answers

- Are practical parsers and scanners based on deterministic or non-deterministic automata?
  - Deterministic
  - Half credit: “they are equivalent” w/o “convert to DFA”
- How can regular expressions be used to specify nested constructs?
  - They cannot. (Scott Book 2.1.2, second sentence)
  - Nested parentheses $(^n\ )^n$ are the canonical example.
- How is a two-dimensional *transition table* used in table-driven scanning?
  - new_state = **table**[old_state][new_character]

Administrivia

- Number of students submitting PA-*X* to a PA-*Y* area: 5 / 39 = 13%
  - This is a 4000-level class.
  - (We could turn off submissions *after* the due date: this yields a zero-tolerance late policy.)
  - (We could turn off submissions *before* the due date: this means students cannot work ahead.)
- PA1 average: 43.8 / 50 = 88%
  - Excellent work, class!
  - Real time grade-checking demo.

Cunning Plan

- Formal Languages
  - Regular Languages Revisited
- Parser Overview
- Context-Free Grammars (CFGs)
- Derivations
- Ambiguity

One-Slide Summary

- A **parser** takes a sequence of tokens as input. If the input is valid, it produces a *parse tree* (or *derivation*).
- **Context-free grammars** are a notation for specifying formal languages. They contain *terminals*, *non-terminals* and *productions* (aka *rewrite rules*).

Languages and Automata

- *Formal languages* are very important in CS
  - Especially in programming languages
- Regular languages
  - The weakest formal languages widely used
  - Many applications
- We will also study *context-free languages*
  - A “stronger” type of formal language

Limitations of Regular Languages

- Intuition: A finite automaton that runs long enough **must repeat states**
  - Pigeonhole Principle: imagine 20 states and 300 input characters
- A finite automaton can't remember how often it has visited a particular state
  - Only enough to store in which state it is in
  - Cannot count, except up to a finite limit
- Language of balanced parentheses is **not regular**: $\{ (^i)^i \mid i \geq 0\}$
  - http://en.wikipedia.org/wiki/Pumping_lemma_for_regular_languages

The Functionality of the Parser

- **Input:** sequence of tokens from lexer
  - e.g., the .cl-lex files you make in PA2
- **Output:** ***parse tree*** of the program
  - Also called an ***abstract syntax tree***
- **Output:** **error** if the input is not valid
  - e.g., “parse error on line 3”

Example

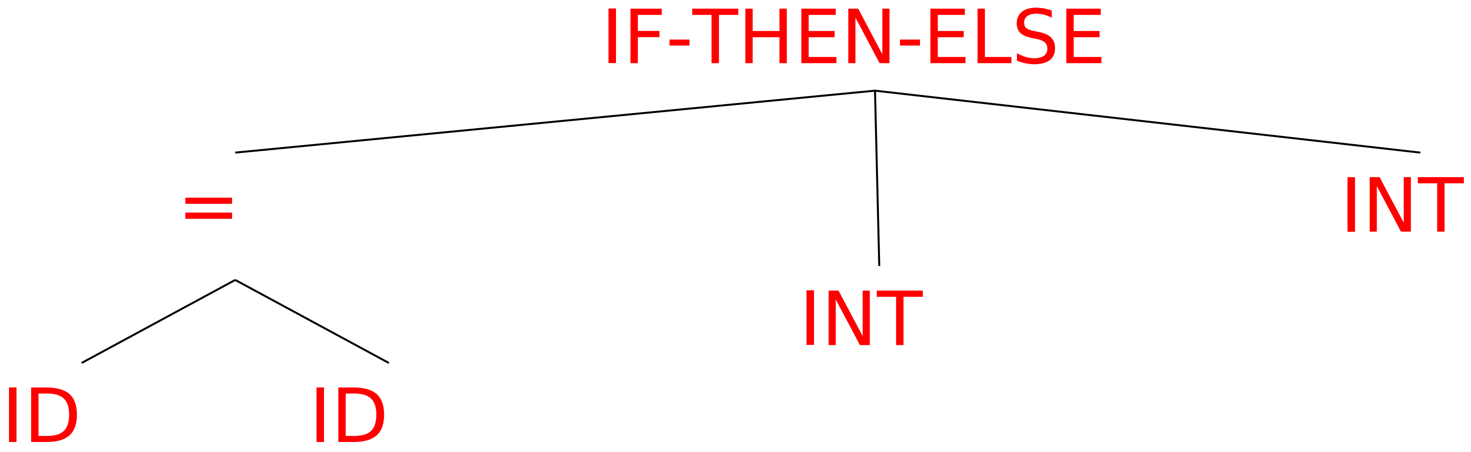
- Cool program text

if x = y then 1 else 2 fi

- Parser input (tokens)

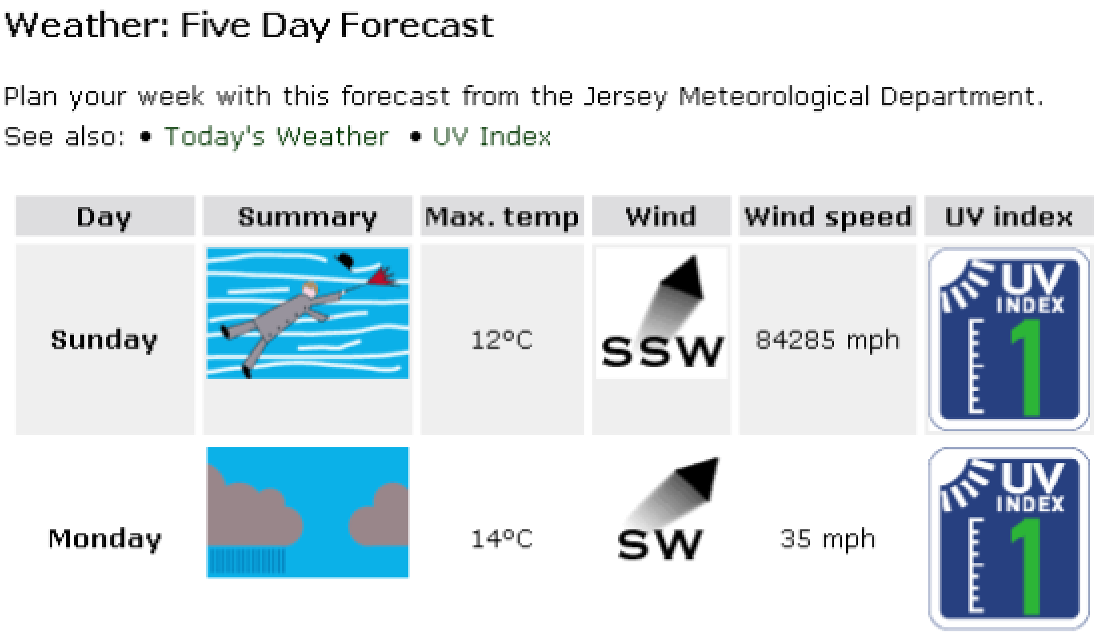
IF ID = ID THEN INT ELSE INT FI

- Parser output (tree)



Comparison: Lexical Analysis

Phase	*Input*	*Output*
Lexer	Sequence of characters	**Sequence of tokens**
Parser	**Sequence of tokens**	Parse tree



Weather: Five Day Forecast

Plan your week with this forecast from the Jersey Meteorological Department.

See also: • Today's Weather • UV Index

Day	Summary	Max. temp	Wind	Wind speed	UV index
Sunday		12°C	SSW	84285 mph	UV INDEX 1
Monday		14°C	SW	35 mph	UV INDEX 1

The Role of the Parser

- Not all sequences of tokens are programs
  - then x * / + 3 while x ; y z then
- The parser **must distinguish between valid and invalid sequences of tokens**
- We need
  - A language or **formalism** to describe valid sequences of tokens
  - A method (an algorithm) for distinguishing valid from invalid sequences of tokens

Programming Language Structure

- Programming languages have **recursive** structure
- Consider the language of arithmetic expressions with integers, +, *, and ( )
- An **expression** is either:
  - an integer
  - an **expression** followed by "+" followed by **expression**
  - an **expression** followed by "*" followed by **expression**
  - a '(' followed by an **expression** followed by ')'
- int , int + int , (int + int) * int are expressions

Notation for Programming Languages

- An alternative notation:

$$E \rightarrow \text{int}$$
$$E \rightarrow E + E$$
$$E \rightarrow E * E$$
$$E \rightarrow (E)$$

- We can view these rules as **rewrite rules**
  - We start with E and replace occurrences of E with some right-hand side
- $E \rightarrow E * E \rightarrow (E) * E \rightarrow (E + E) * E$

  $\rightarrow \ldots \rightarrow (\text{int} + \text{int}) * \text{int}$

Observation

- All arithmetic expressions can be obtained by a sequence of replacements
- Any sequence of replacements forms a valid arithmetic expression
- This means that we cannot obtain

(int))

by any sequence of replacements. Why?

- This notation is a **context-free grammar**

Context Free Grammars

- A context-free grammar consists of
  - A set of ***non-terminals*** *N*
    - Written in uppercase in these notes
  - A set of ***terminals*** *T*
    - Lowercase or punctuation in these notes
  - A ***start symbol*** *S* (a non-terminal)
  - A set of ***productions*** (rewrite rules)
- Assuming $E \in N$

$E \to \varepsilon$, or

$E \to Y_1\ Y_2 \ldots Y_n$ where $Y_i \subseteq N \cup T$

Examples of CFGs

Simple arithmetic expressions:

E → int
E → E + E
E → E * E
E → (E)

- One *non-terminal*: **E**
- Several *terminals*: **int + * ( )**
  - Called terminals because they are never replaced
- By convention the non-terminal for the first production is the start symbol

The Language of a CFG

Read productions as replacement rules:

$X \rightarrow Y_1 \ldots Y_n$

Means X can be replaced by $Y_1 \ldots Y_n$

$X \rightarrow \varepsilon$

Means X can be erased or eaten (replaced with empty string)

Key Idea

To construct a **valid sequence** of terminals:

- Begin with a string consisting of the start symbol "S"
- Replace any non-terminal X in the string by a right-hand side of some production

$$X \rightarrow Y_1 \ldots Y_n$$

- Continue replacing until there are only terminals in the string

The Language of a CFG: $\rightarrow$

More formally, write

$$X_1 \ldots X_{i-1} X_i X_{i+1} \ldots X_n$$
$$\rightarrow$$
$$X_1 \ldots X_{i-1} Y_1 \ldots Y_m X_{i+1} \ldots X_n$$

if there is a production

$$X_i \rightarrow Y_1 \ldots Y_m$$

The Language of a CFG: $\rightarrow^*$

Write

$$X_1 \ldots X_n \rightarrow^* Y_1 \ldots Y_m$$

if

$$X_1 \ldots X_n \rightarrow \ldots \rightarrow \ldots \rightarrow Y_1 \ldots Y_m$$

in 0 or more steps.

NO! - Bad User!!!

You've been warned 3 times that this file does not exist.
Now you've made us catch this worthless exception and we're upset.
Do not do this again.

The Language of a CFG

Let G be a context-free grammar with start symbol S. Then the **language of** G is:

$$L(G) = \{\, a_1 \ldots a_n \mid S \to^{*} a_1 \ldots a_n \text{ and every } a_i \text{ is a terminal} \,\}$$

L(G) is a set of strings over the alphabet of terminals.

CFG Language Examples

- $S \rightarrow 0$ also written as $S \rightarrow 0 \mid 1$

  $S \rightarrow 1$

  Generates the language { "0", "1" }
- What about $S \rightarrow 1\,A$

  $A \rightarrow 0 \mid 1$
- What about $S \rightarrow 1\,A$

  $A \rightarrow 0 \mid 1\,A$
- What about $S \rightarrow \varepsilon \mid (\,S\,)$

Pencil and paper corner!

Arithmetic Example

Simple arithmetic expressions:

$$E \rightarrow E{+}E \mid E * E \mid (E) \mid \mathrm{id}$$

Some elements of the language:

id	$\mathrm{id} + \mathrm{id}$
(id)	$\mathrm{id} * \mathrm{id}$
$(\mathrm{id}) * \mathrm{id}$	$\mathrm{id} * (\mathrm{id})$

SAFE HANDLING INSTRUCTIONS
THIS PRODUCT WAS PREPARED FROM [illegible] AND/OR POULTRY. SOME FOOD PRODUCTS MAY CONTAIN [illegible] THAT COULD CAUSE ILLNESS IF THE PRODUCT IS MISHANDLED OR COOKED IMPROPERLY. FOR YOUR PROTECTION, FOLLOW THESE SAFE HANDLING INSTRUCTIONS.
KEEP REFRIGERATED OR FROZEN. THAW IN REFRIGERATOR OR MICROWAVE.
KEEP RAW MEAT AND POULTRY SEPARATE FROM OTHER FOODS. WASH WORKING SURFACES (INCLUDING CUTTING BOARDS), UTENSILS, AND HANDS AFTER TOUCHING RAW MEAT OR POULTRY.
COOK THOROUGHLY
KEEP HOT FOODS HOT. REFRIGERATE LEFTOVERS IMMEDIATELY OR DISCARD
MEIJER. Fresh
2 8 0 2 9 5 3 0 3 5 0 9
146
Sell By
Feb 11, 07
Net Wt/Ct
1.17 lb
Regular Price Per lb.
$3.99/lb
Regular Total Price
$-429492.0
SINCE 1978
CERTIFIED ANGUS BEEF
New Price Per lb.
$2.99/lb
You Save
$4294
You Pay
$3.50
Distributed by Meijer Grand Rapids, MI 49544

Cool Example

A fragment of COOL:

$$\begin{array}{lll}
\text{EXPR} & \rightarrow & \text{if EXPR then EXPR else EXPR fi} \\
 & | & \text{while EXPR loop EXPR pool} \\
 & | & \text{id}
\end{array}$$

Cool Example (Cont.)

Some elements of the language

id

if id then id else id fi

while id loop id pool

if while id loop id pool then id else id

if if id then id else id fi then id else id fi

Notes

The idea of a CFG is a big step. But:

- Membership in a language is "yes" or "no"
  - we also need *parse tree* of the input
- We must *handle errors* gracefully
- Need an *implementation* of CFGs
  - bison, yacc, ocamlyacc, ply, etc.

More Notes

- Form of the grammar is important
  - Many grammars generate the same language
    - Give me an example.
  - Automatic tools are sensitive to the grammar
  - Note: Tools for regular languages (e.g., flex) are also sensitive to the form of the regular expression, but this is rarely a problem in practice

Derivations and Parse Trees

A ***derivation*** is a sequence of productions

$$S \rightarrow \ldots \rightarrow \ldots$$

A derivation can be **drawn as a tree**

- Start symbol is the tree's root
- For a production $X \rightarrow Y_1 \ldots Y_n$ add children $Y_1, \ldots, Y_n$ to node X

Derivation Example

- Grammar $E \rightarrow E+E \mid E * E \mid (E) \mid id$

- String $id * id + id$

- We're going to build a derivation!

Derivation Example (Cont.)

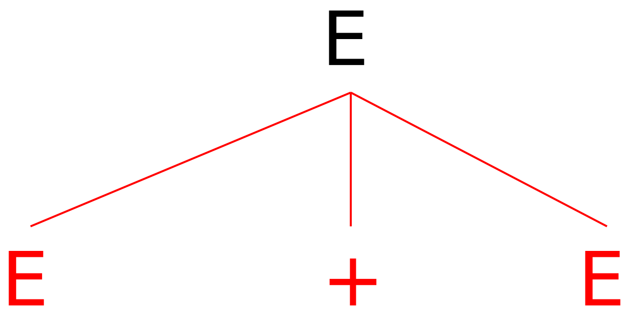
$$\left.\begin{aligned} & E \\ \rightarrow\quad & E+E \\ \rightarrow\quad & E * E+E \\ \rightarrow\quad & \mathrm{id} * E + E \\ \rightarrow\quad & \mathrm{id} * \mathrm{id} + E \\ \rightarrow\quad & \mathrm{id} * \mathrm{id} + \mathrm{id} \end{aligned}\right\} \textbf{Thus } \mathbf{E} \rightarrow^{*} \mathbf{id * id + id}$$

Derivation in Detail (1)

E

E

Derivation in Detail (2)



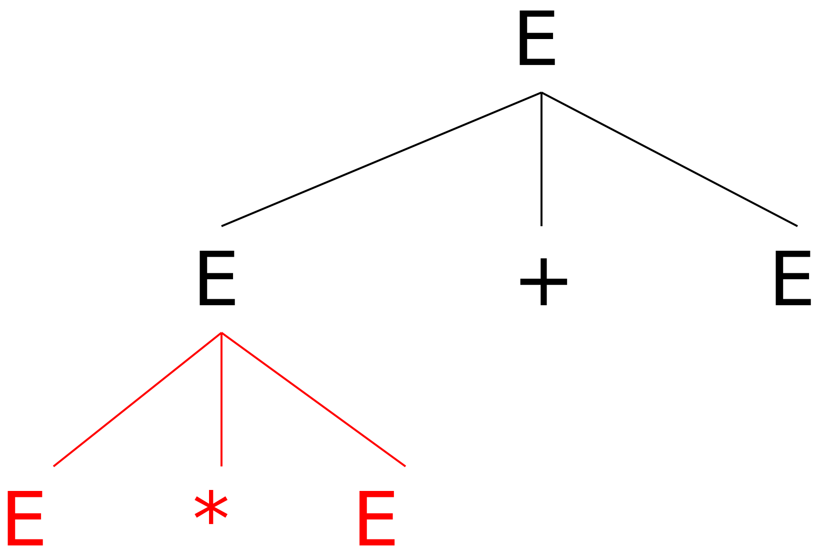
$$
\begin{aligned}
& E \\
\rightarrow\ & E+E
\end{aligned}
$$

Derivation in Detail (3)

$$E$$

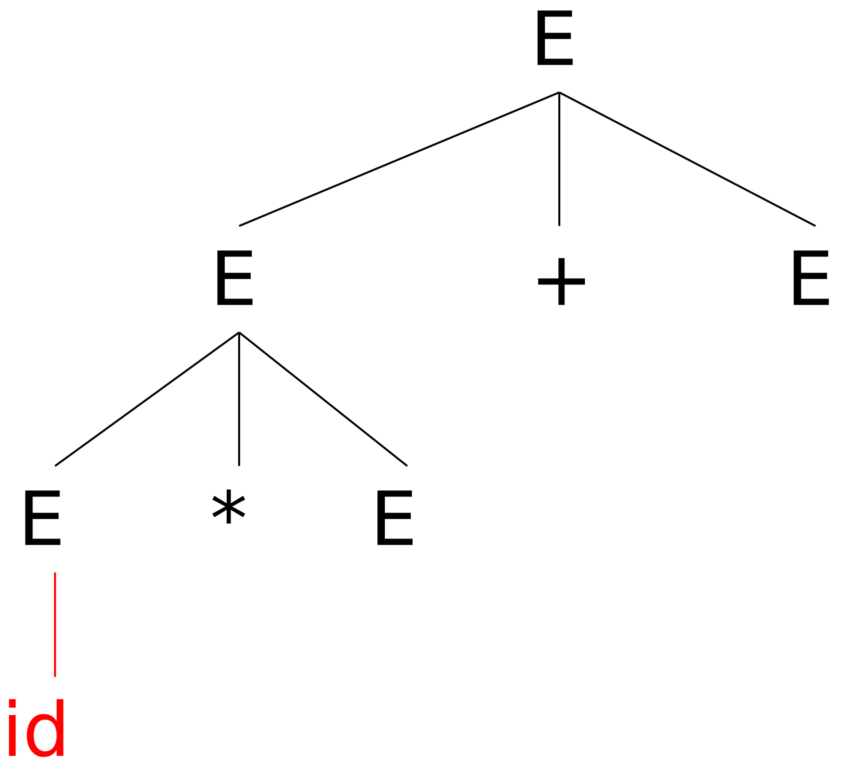
$$\rightarrow \quad E+E$$

$$\rightarrow \quad E * E+E$$



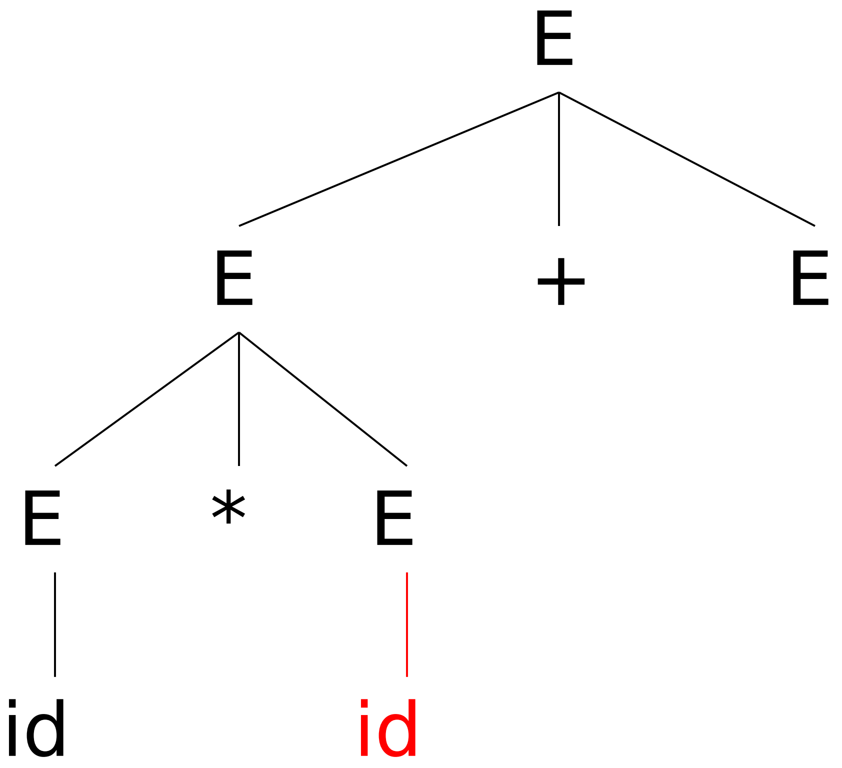
Derivation in Detail (4)

$$
\begin{aligned}
& E \\
\rightarrow\ & E+E \\
\rightarrow\ & E * E+E \\
\rightarrow\ & \mathrm{id} * E + E
\end{aligned}
$$



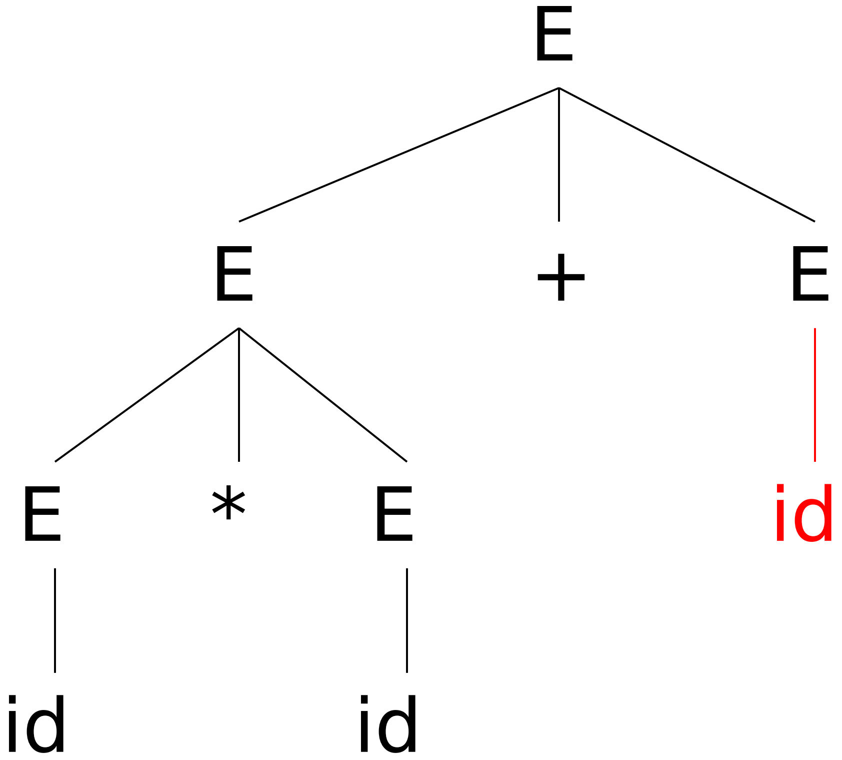
Derivation in Detail (5)

$$
\begin{aligned}
& E \\
\rightarrow\ & E+E \\
\rightarrow\ & E * E+E \\
\rightarrow\ & id * E + E \\
\rightarrow\ & id * id + E
\end{aligned}
$$



Derivation in Detail (6)

$$
\begin{aligned}
& E \\
\rightarrow\ & E+E \\
\rightarrow\ & E * E+E \\
\rightarrow\ & id * E + E \\
\rightarrow\ & id * id + E \\
\rightarrow\ & id * id + id
\end{aligned}
$$



Notes on Derivations

- A ***parse tree*** has
  - Terminals at the leaves
  - Non-terminals at the interior nodes
- A left-right traversal of the leaves is the original input
- The parse tree shows the association of operations, the input string does not!

Q: Radio (117 / 842)

- This NPR radio show features Tom and Ray Magliozzi as Click and Clack the Tappet Brothers. It includes Boston accents, a weekly "Puzzler", and is brought to you in part by "Paul Murky of Murky Research" and the law firm of "Dewey, Cheetham and Howe".

Q: Games (548 / 842)

- In the card game **Hearts**, how many points would you accumulate if you ended up with all of the Queens and all of the Jacks? (Standard scoring.)

Q: Music (158 / 842)

- Name the *"shocking"* 1976 line dance created by Ric Silver and sung and choreographed by Marcia Griffiths.

Q: Games (582 / 842)

- Minty, Snuzzle, Butterscotch, Bluebelle, Cotton Candy, and Blossom were the first six characters in the 1982 edition of this series of Hasbro-produced brushable hoofed dolls with designs on their hips.

Left-most and Right-most Derivations

- The example here is a ***left-most derivation***
  - At each step, replace the left-most non-terminal
- There is an equivalent notion of a ***right-most derivation***
  - Not shown on this slide.
  - But coming up next!

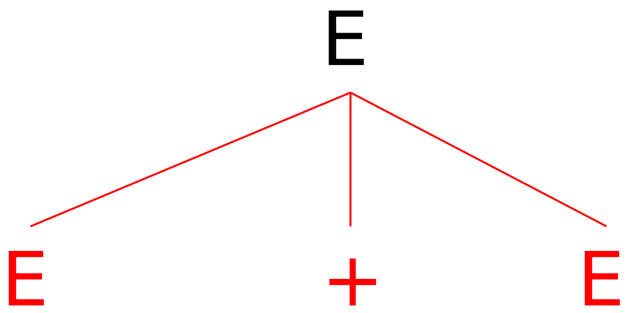
$$
\begin{aligned}
& E \\
\rightarrow\quad & E+E \\
\rightarrow\quad & E+\mathrm{id} \\
\rightarrow\quad & E * E + \mathrm{id} \\
\rightarrow\quad & E * \mathrm{id} + \mathrm{id} \\
\rightarrow\quad & \mathrm{id} * \mathrm{id} + \mathrm{id}
\end{aligned}
$$

Right-most Derivation in Detail (1)

E

E

Right-most Derivation in Detail (2)

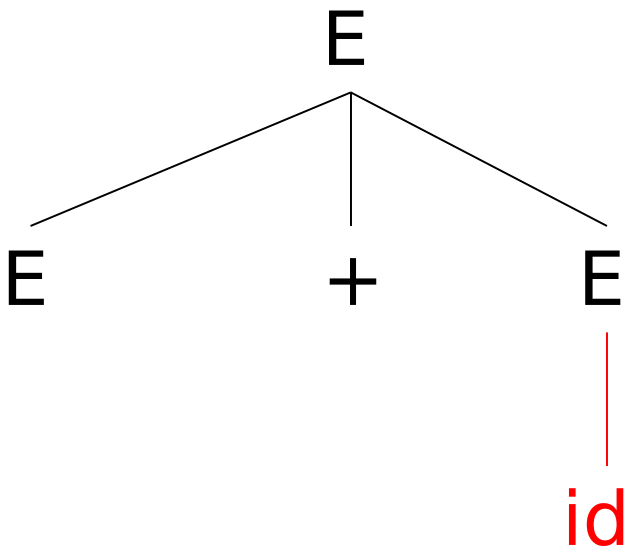


E

$\rightarrow E+E$

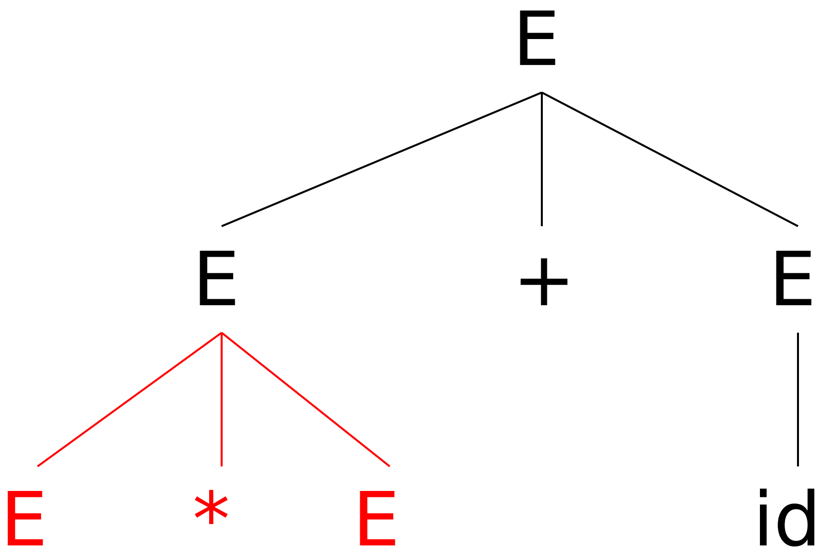
Right-most Derivation in Detail (3)

$$\begin{aligned} & E \\ \rightarrow\ & E+E \\ \rightarrow\ & E+id \end{aligned}$$



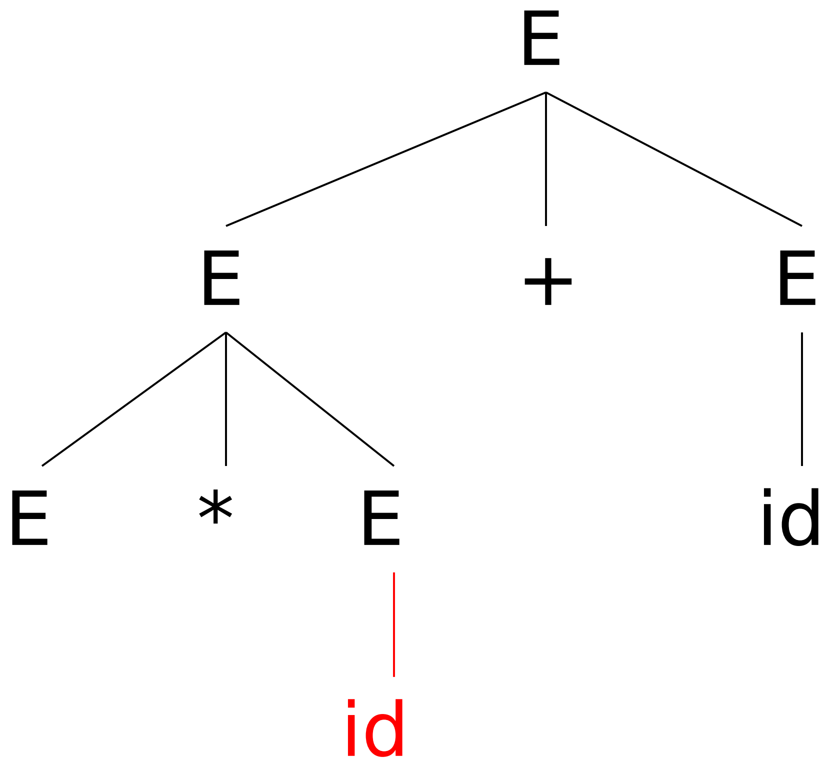
Right-most Derivation in Detail (4)

$$
\begin{aligned}
& E \\
\rightarrow\ & E+E \\
\rightarrow\ & E+id \\
\rightarrow\ & E * E + id
\end{aligned}
$$



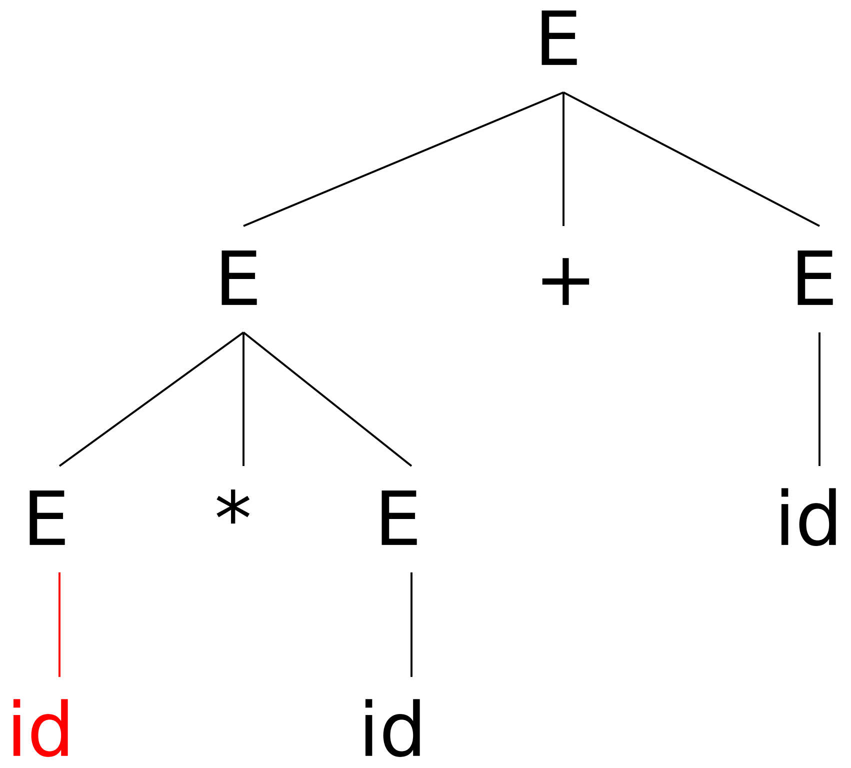
Right-most Derivation in Detail (5)

$$
\begin{aligned}
& E \\
\rightarrow\quad & E+E \\
\rightarrow\quad & E+\mathrm{id} \\
\rightarrow\quad & E * E + \mathrm{id} \\
\rightarrow\quad & E * \mathrm{id} + \mathrm{id}
\end{aligned}
$$



Right-most Derivation in Detail (6)

$$
\begin{aligned}
& E \\
\rightarrow\ & E+E \\
\rightarrow\ & E+\mathrm{id} \\
\rightarrow\ & E * E + \mathrm{id} \\
\rightarrow\ & E * \mathrm{id} + \mathrm{id} \\
\rightarrow\ & \mathrm{id} * \mathrm{id} + \mathrm{id}
\end{aligned}
$$



Derivations and Parse Trees

- Note that for each parse tree there is a left-most and a right-most derivation

- The difference is the order in which branches are added

- We will start with a parsing technique that yields left-most derivations
  - Later we'll move on to right-most derivations

Summary of Derivations

- We are not just interested in whether

  $$s \in L(G)$$

  - We also need a parse tree for s

- A derivation defines a parse tree
  - But one parse tree may have many derivations

- Left-most and right-most derivations are important in parser implementation

Review

- A parser consumes a sequence of tokens s and produces a parse tree
- Issues:
  - How do we **recognize** that $s \in L(G)$?
  - A **parse tree** of s describes how $s \in L(G)$
  - **Ambiguity**: more than one parse tree (interpretation) for some string s
  - **Error**: no parse tree for some string s
  - How do we **construct** the parse tree?

Ambiguity

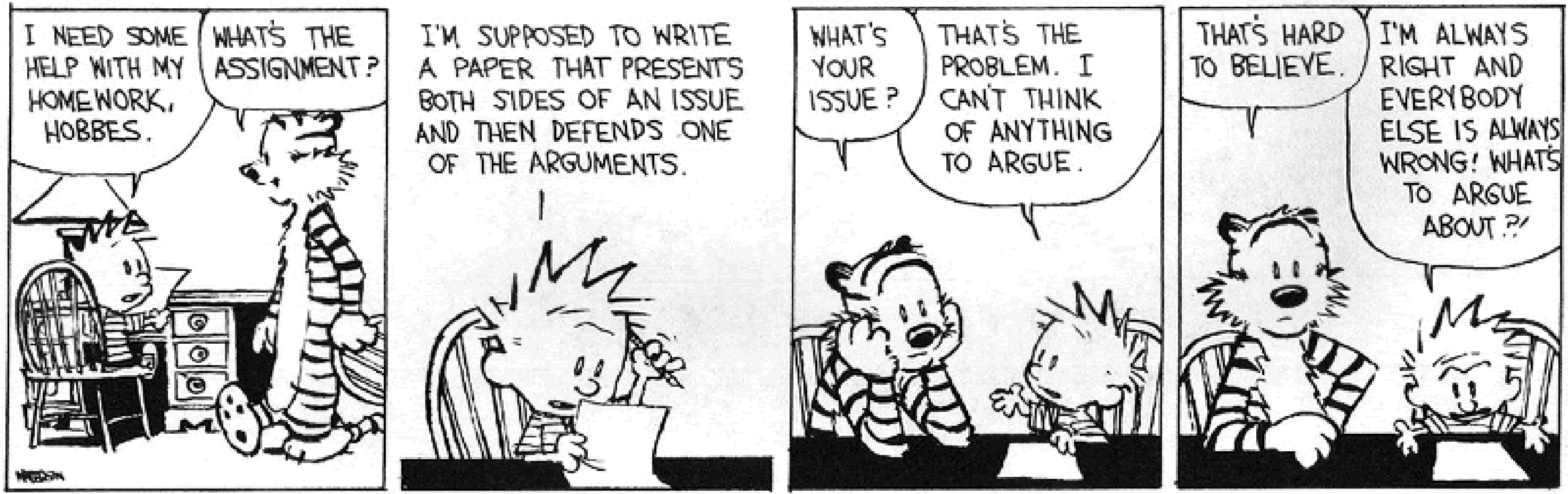
- (Ambiguous) Grammar G

  E → E + E | E * E | (E) | int

- Some strings in L(G)

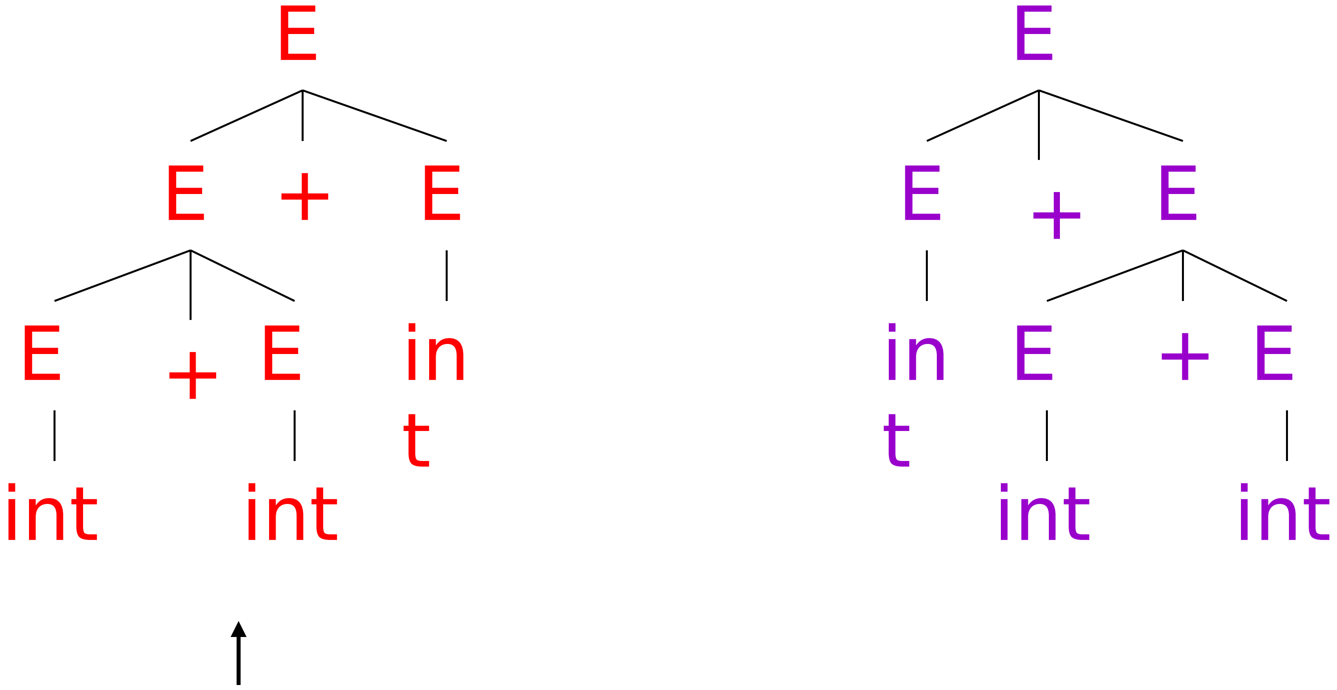
  int + int + int

  int * int + int


I NEED SOME HELP WITH MY HOMEWORK, HOBBES.
WHAT'S THE ASSIGNMENT?
I'M SUPPOSED TO WRITE A PAPER THAT PRESENTS BOTH SIDES OF AN ISSUE AND THEN DEFENDS ONE OF THE ARGUMENTS.
WHAT'S YOUR ISSUE?
THAT'S THE PROBLEM. I CAN'T THINK OF ANYTHING TO ARGUE.
THAT'S HARD TO BELIEVE.
I'M ALWAYS RIGHT AND EVERYBODY ELSE IS ALWAYS WRONG! WHAT'S TO ARGUE ABOUT?!

Ambiguity. Example

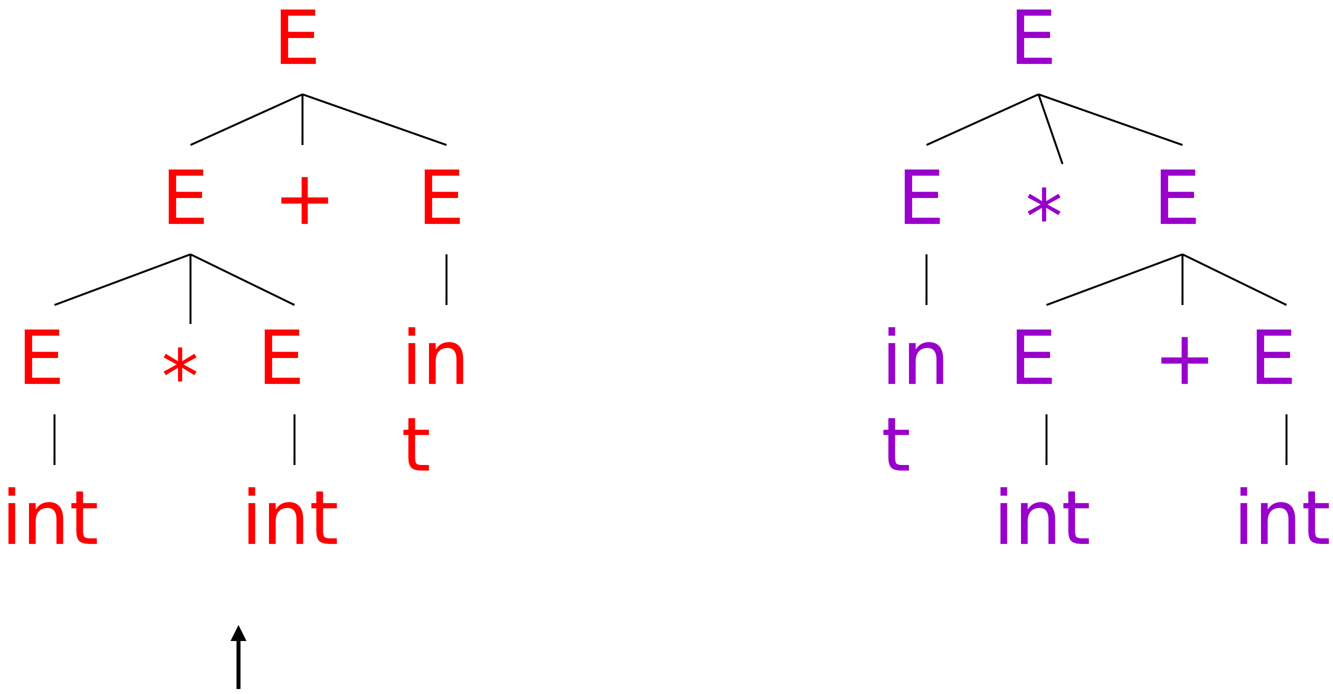
The string int + int + int has two parse trees



+ is left-associative

Ambiguity. Example

The string int * int + int has two parse trees



* has higher precedence than +

Ambiguity (Cont.)

- A grammar is ***ambiguous*** if it has more than one parse tree for some string
  - Equivalently, there is more than one right-most or left-most derivation for some string
- Ambiguity is bad
  - Leaves meaning of some programs ill-defined
- Ambiguity is common in programming languages
  - Arithmetic expressions
  - IF-THEN-ELSE

Dealing with Ambiguity

- There are several ways to handle ambiguity
- Most direct method is to *rewrite the grammar* unambiguously

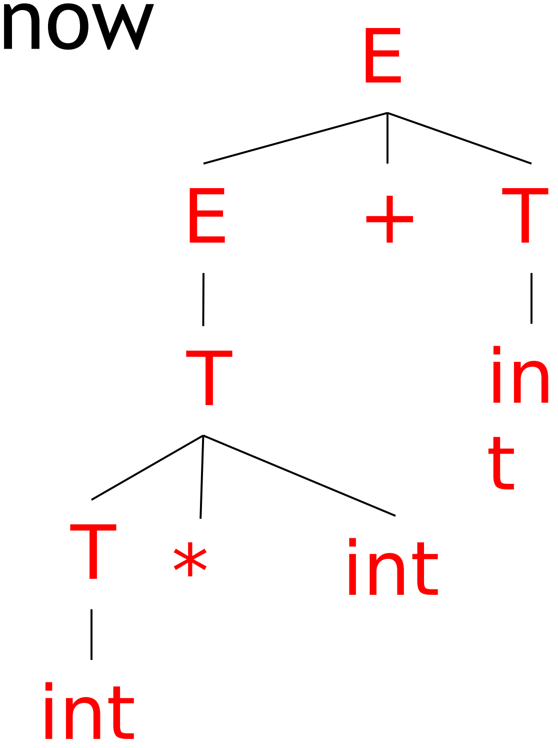
  E → E + T | T

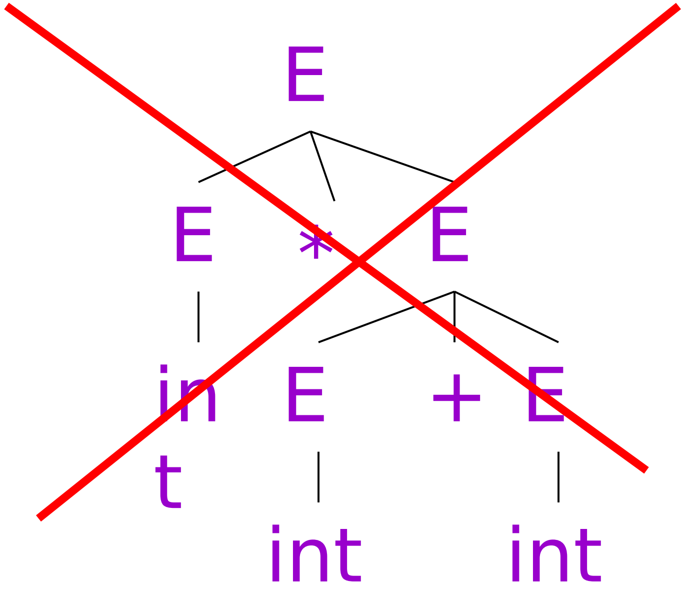
  T → T * int | int | (E)

- Enforces precedence of * over +
- Enforces left-associativity of + and *

Ambiguity. Example

The int * int + int has ony one parse tree now





Ambiguity: The Dangling Else

- Consider this new grammar

$$
\begin{aligned}
E \rightarrow\; & \text{if } E \text{ then } E \\
\mid\; & \text{if } E \text{ then } E \text{ else } E \\
\mid\; & \text{OTHER}
\end{aligned}
$$

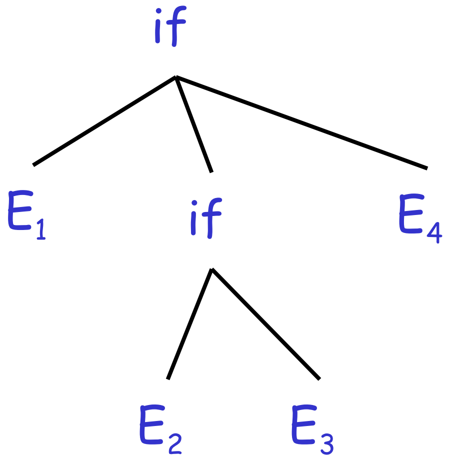
- This grammar is also ambiguous

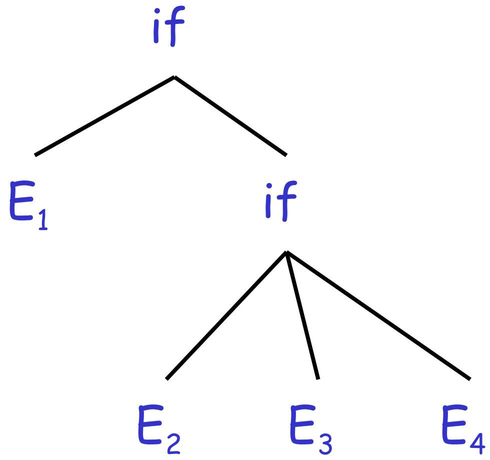
The Dangling Else: Example

- The string

if E_1 then if E_2 then E_3 else E_4

has two parse trees





- Typically we want the second form

The Dangling Else: A Fix

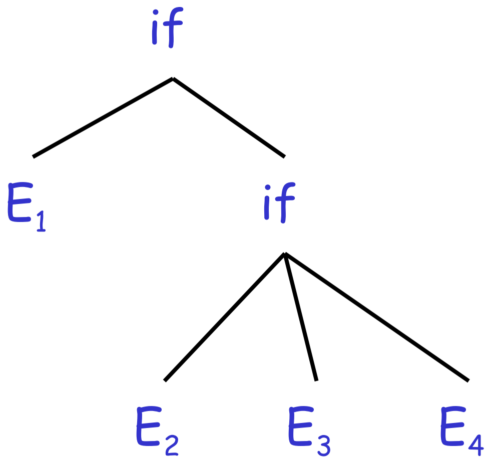
- else matches the closest unmatched then
- We can describe this in the grammar (distinguish between matched and unmatched “then”)

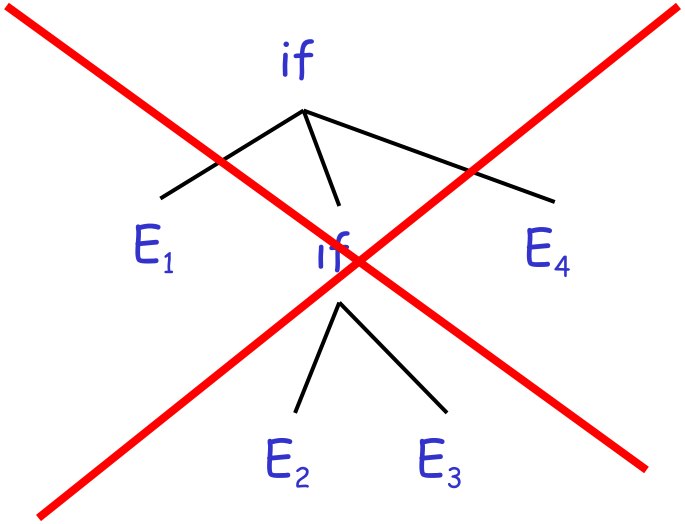
```
E →    MIF                    /* all then are matched */
   |   UIF                    /* some then are unmatched */
MIF →  if E then MIF else MIF
    |  OTHER
UIF → if E then E
    |  if E then MIF else UIF
```

- Describes the same set of strings

The Dangling Else: Example Revisited

- The expression if E_1 then if E_2 then E_3 else E_4





- A valid parse tree (for a UIF)

- Not valid because the then expression is not a MIF

Ambiguity

- No general techniques for handling ambiguity
- Impossible to convert automatically every ambiguous grammar to an unambiguous one
- Used with care, ambiguity can simplify the grammar
  - Sometimes allows more natural definitions
  - So we need disambiguation mechanisms
  - As shown next ...

Precedence and Associativity Declarations

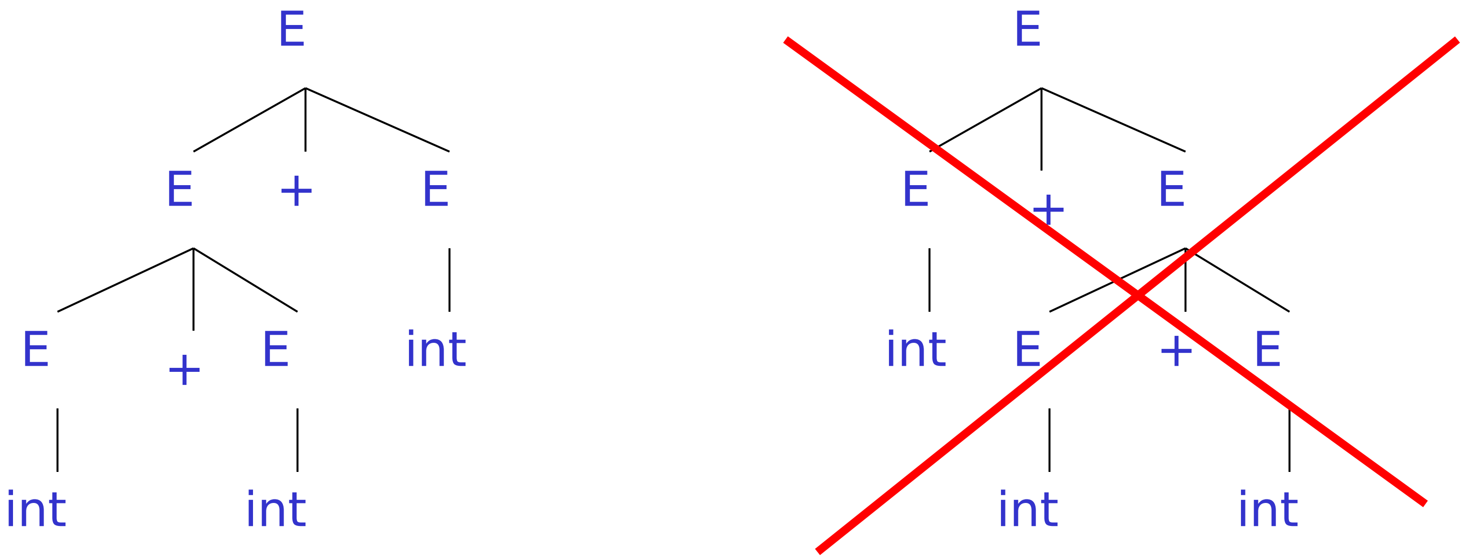
- Instead of rewriting the grammar
  - Use the more natural (ambiguous) grammar
  - Along with disambiguating declarations

- Most tools allow ***precedence and associativity declarations*** to disambiguate grammars

- Examples …

Associativity Declarations

- Consider the grammar E → E + E | int
  - And the string: int + int + int



- Left-associativity declaration: **%left +**

  %left *

Review

- We can specify language syntax using CFG
- A parser will answer whether $s \in L(G)$
- … and will build a parse tree
- … and pass on to the rest of the compiler

- Next:
  - How do we answer $s \in L(G)$ and build a parse tree?

Homework

- WA1 (written homework) due
  - You must work alone.
  - Write or print out your answers.
  - Turn in before class or in drop-box (423 Rice).
- PA2 (Lexer) Due
  - You may work in pairs.