# Lecture 12

COP3402 FALL 2015 – DR. MATTHEW GERBER – 10/14/2015

FROM EURIPIDES MONTAGNE, FALL 2014

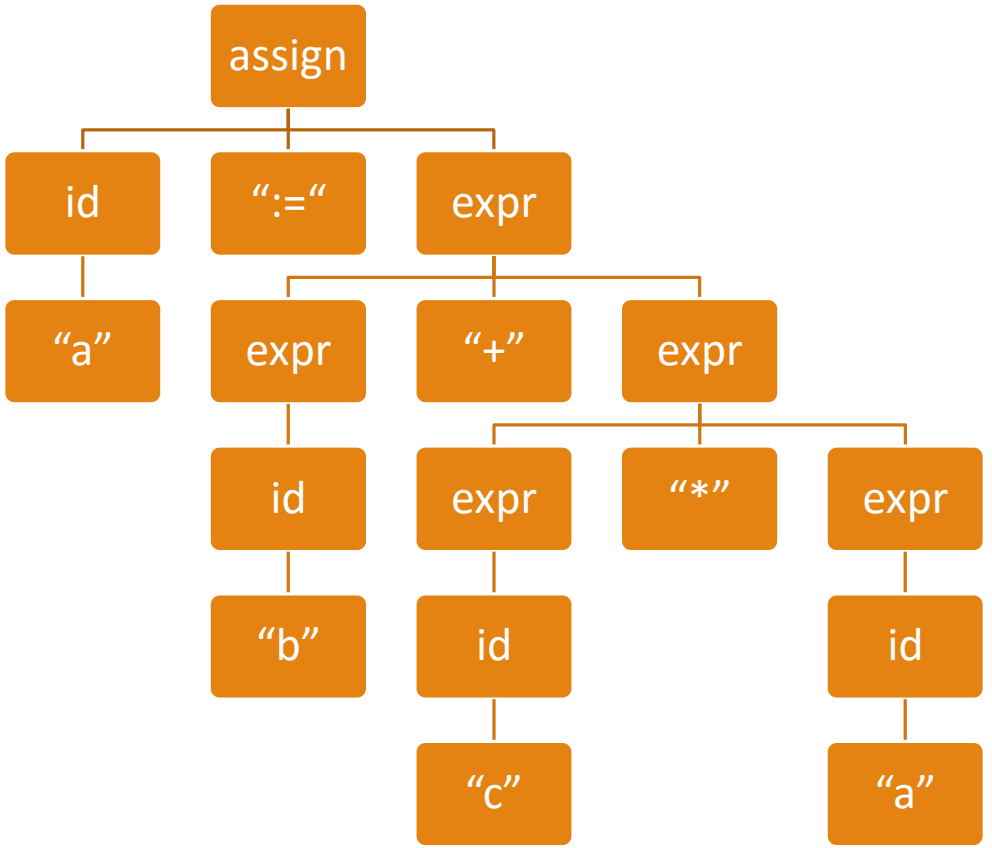
# Tonight

## Ambiguity in Parsing

# An Unfortunate Parse Tree Example

Shown here is the parse tree for the expression a := b + c * a in the language defined by the following grammar.

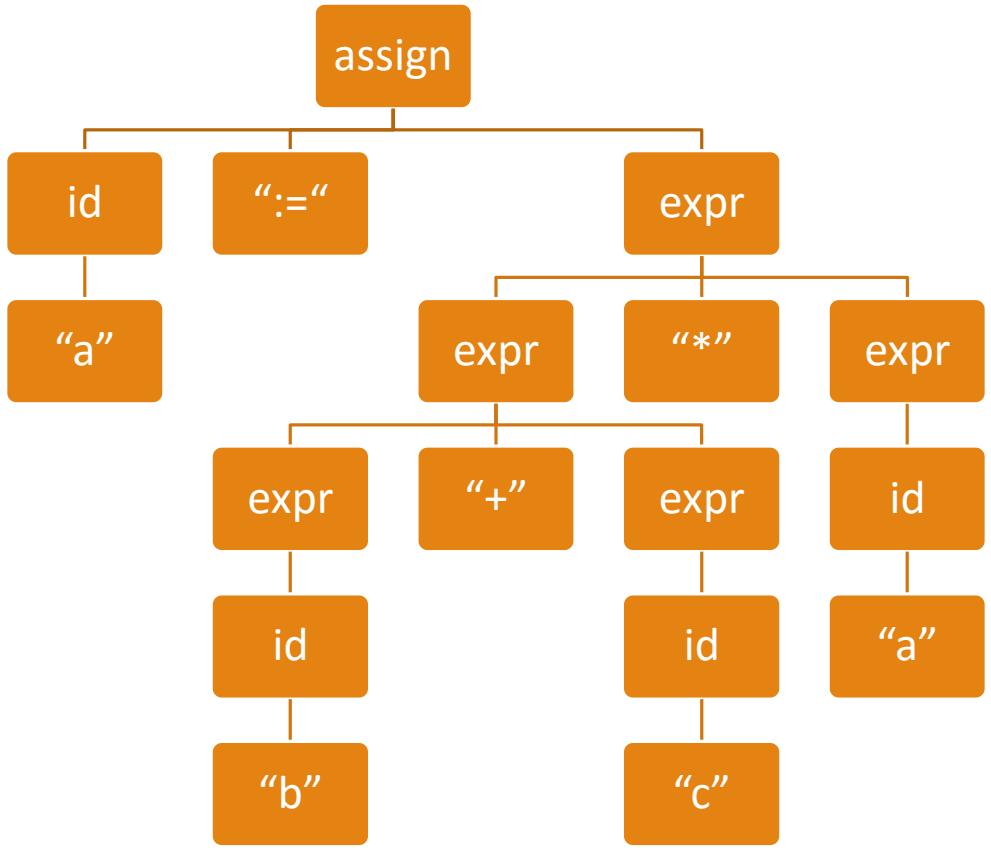
```
assgn   ::= id ":=" expr
id      ::= "a" | "b" | "c"
expr    ::= expr "+" expr
          | expr "*" expr
          | "(" expr ")"
          | id
```



# An Unfortunate Parse Tree Example

Shown here is **also** the parse tree for the expression a := b + c * a in the language defined by the following grammar.

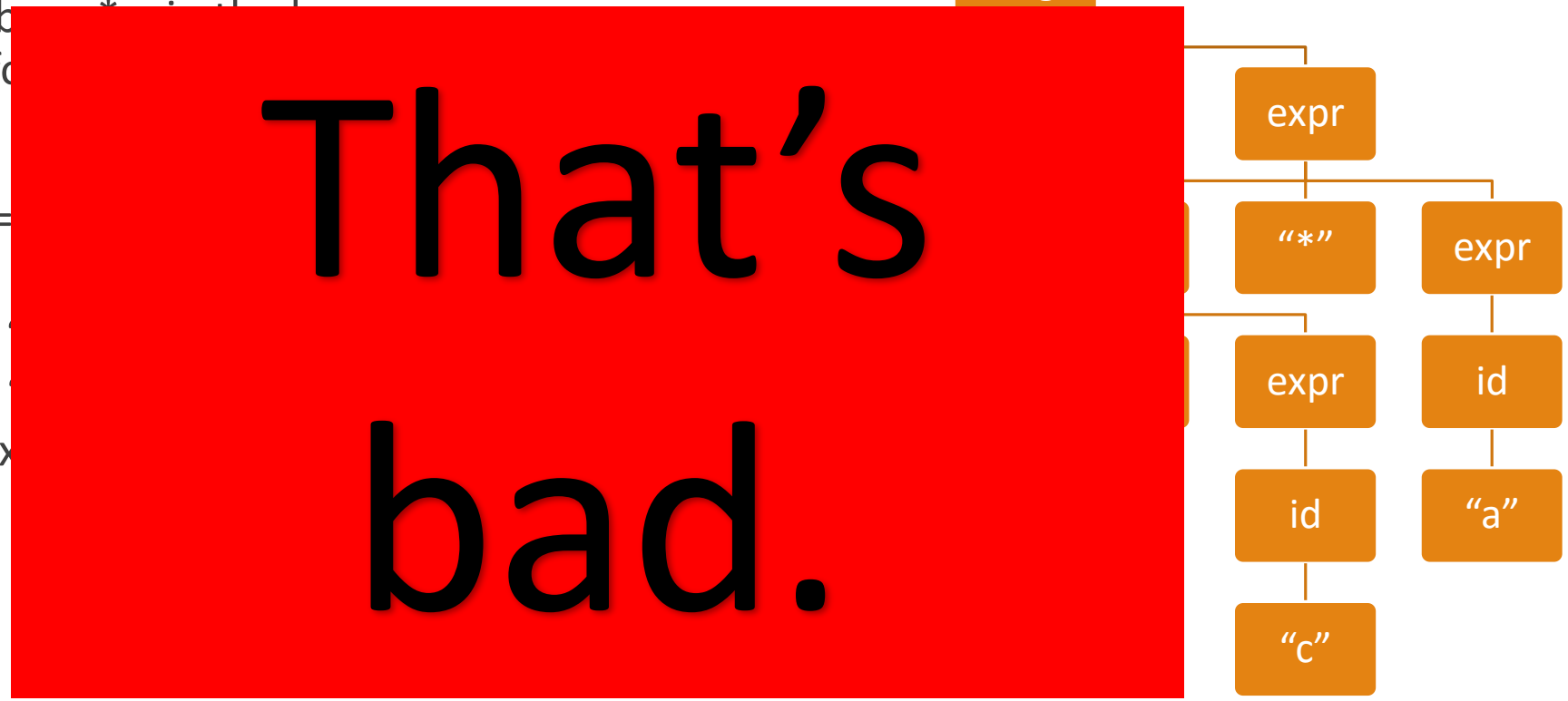
```
assgn   ::= id ":=" expr
id      ::= "a" | "b" | "c"
expr    ::= expr "+" expr
          | expr "*" expr
          | "(" expr ")"
          | id
```



# An Unfortunate Parse Tree Example

Shown here is **also** the parse tree for the expression a := b
defined by the fo

assgn ::= id ":=
id ::= "a" |
expr ::= expr
| expr
| "(" ex
| id



# Ambiguity

A grammar that can generate a statement for which there are two or more distinct parse trees is *ambiguous*.

So this grammar is ambiguous because it generates two different parse trees for the expression a = b + c * a.

This isn't a theoretical problem – it's real, and serious. We'd never use this grammar in a real language.

```
assgn   ::= id ":=" expr
id      ::= "a" | "b" | "c"
expr    ::= expr "+" expr
          | expr "*" expr
          | "(" expr ")"
          | id
```

# Ambiguity

In *this* case, fortunately, it's easy to fix.

- The problem is just operator predecence
- The grammar on the right enforces the usual order
- **Note:** Operators *lower* in the tree have *superior* precedence!

```
assgn   ::= id ":=" expr
id      ::= "a" | "b" | "c"
expr    ::= expr "+" expr
          | expr "*" expr
          | "(" expr ")"
          | id
```

```
assgn   ::= id ":=" expr
id      ::= "a" | "b" | "c"
expr    ::= expr "+" term
          | term
term    ::= term "*" factor
          | factor
factor  ::= "(" expr ")"
          | id
```

# With The New Grammar

## LEFT MOST DERIVATION

<assgn> → <id> := <expr>
→ a := <expr>
→ a := <expr> + <term>
→ a := <term> + <term>
→ a := <factor> + <term>
→ a := <id> + <term>
→ a := b + <term>
→ a := b + <term> *<factor>
→ a := b + <factor> * <factor>
→ a := b + <id> * <factor>
→ a := b + c * <factor>
→ a := b + c * <id>
→ a := b + c * a

## RIGHT MOST DERIVATION

<assgn> → <id> := <expr>
→ <id> := <expr> + <term>
→ <id> := <expr> + <term> * <factor>
→ <id> := <expr> + <term> * <id>
→ <id> := <expr> + <term> * a
→ <id> := <expr> + <factor> * a
→ <id> := <expr> + <id> * a
→ <id> := <expr> + c * a
→ <id> := <term> + c * a
→ <id> := <factor> + c * a
→ <id> := <id> + c * a
→ <id> := b + c * a
→ a := b + c * a

# With The New Grammar

LEFT MOST DERIVATION

<assgn> → <id> :=
→ a := <ex
→ a := <ex
→ a := <te
→ a := <fa
→ a := <id
→ a := b +
→ a := b +
→ a := b +
→ a := b +
→ a := b +
→ a := b +
→ a := b +

RIGHT MOST DERIVATION

rm>
rm> * <factor>
rm> * <id>
rm> * a
ctor> * a
> * a
a
a
* a

That's good.

# The General Rules

Multiplication and division have higher precedence than addition and subtraction.

- a + b * 3 → a + (b * 3)

Operators of equal precedence associate to the left.

- a + b + 3 → (a + b) + 3

You know this already! So write the precedence *into the grammar.*

```
expr    ::= expr op expr | id | number | (expr)        NO!
op      ::= "+" | "-" | "*" | "/"

expr    ::= term | expr "+" term | expr "-" term
term    ::= factor | term "*" factor | term "/" factor   YES!
factor  ::= id | number | (expr)
```

# Exercises (Try This at Home)

## Are these grammars ambiguous?

**E ::= T | E + T | E - T**

**T ::= F | T * F | T / F**

**F ::= id | num | ( E )**

Try parsing id + id - id

**E ::= T | E + T**

**T ::= F | T * F**

**F ::= id | ( E )**

Try parsing id + id - id

**E ::= E + E | id**

Try parsing id + id + id

**E ::= E + id | id**

Try parsing id + id + id

# One More Exercise

**E ::= E + E | E * E | ( E ) | id**

…*is* ambiguous. We can rewrite it as:

**E ::= E + T | T**

**T ::= T * F | F**

**F ::= id | ( E )**

Now find the parse tree for:

id + id * id

# Another Problem: Left-Recursion

A grammar is *left-recursive* if it has a non-terminal symbol that can be written to itself followed by something else – that is, if it has any derivation of the form
A ::= A $\alpha$

- Sadly, most grammars useful for anything are left-recursive in their simplest forms
- Top-down parsers can't easily handle left-recursive grammars
- We have to transform the grammar to eliminate left-recursion

For example, we could rewrite A ::= A $\alpha$ | $\beta$ as:

A ::= $\beta$ A'

A' ::= $\alpha$ A' | $\varepsilon$

# Another Left-Recursion Example

E ::= E “+” E | T

T ::= T “*” F | F

F ::= id | “(“ E “)”

The key is always the same: find a way to move the iteration to the right instead of the left.

What you’ll usually do is create an intermediate rule that resolves to either a rightward expansion of itself or the empty string.

E → T E’

E’ → “+” T E’ | ε

T → F T’

T’ → “*” F T’ | ε

F → “(“ E “)” | id

# One More Problem: Left Factoring

We also want to avoid multiple rewrite rules whose substitution side starts the same way – and unfortunately, rules created by the OR operator count. So this…

$$A ::= \alpha\ \beta_1 \mid \alpha\ \beta_2$$

…is a problem. The good news is, it's easy to get rid of:

$$A \rightarrow \alpha\ A'$$

$$A' \rightarrow \beta_1 \mid \beta_2$$

…and that approach readily extends to all cases.

# Next Time: Parsing