Lecture 8: Recursive-descent parsing

- **Recursive-descent formalized**
  - **FIRST sets**
  - **LL(1) condition**
  - **Transformations to LL(1) form**
- **Grammars for expressions - a difficult case**

(Next week: LR(1) parsing, ocamlyacc)

Top-down parsing

- **For each non-terminal with productions:**

$$A \rightarrow \vec{X} \mid \vec{Y} \mid \ldots \mid \vec{Z}$$

$\vec{X} = X_1 X_2 \ldots X_n$, $n \geq 0$, $X_i \in G$

define parseA:

`parseA toklis` = choose production based on hd toklis:

if $A \rightarrow \vec{X}$: handle $\vec{X}$; produce $A \Rightarrow X_1 X_2 \ldots X_n$

else if $A \rightarrow \vec{Y}$: handle $\vec{Y}$, etc

handle $X_1 X_2 \ldots X_n$: handle X_1; handle X_2; ...; handle X_n

where handle `t` : `if hd toklis = t`
`then` remove `t` and continue
`else` error

handle B : `parseB toklis`

"choose production based on hd toklis"

derives in one step

- Need to formalize some things...
- Define "$\Rightarrow$": $X_1 \ldots X_n \Rightarrow w_1 \ldots w_n$, where $X_i \in G$ and $w_i \in G^*$, if there exists j such that $X_j \to w_j$ is a production in G, and for all $i \neq j$, $X_i = w_i$.
- $\Rightarrow^+$ and $\Rightarrow^*$ are the transitive and reflexive-transive closures of $\Rightarrow$. (Say $\vec{X}$ *derives* α if $\vec{X} \Rightarrow^* \alpha$.)

  (E.g. α is a sentence of G if the start symbol of G derives α and α consists solely of tokens.)

A → (B)
B → id | A

A ⇒ (B) ⇒ (A) ⇒ ((B))
⇒ ((id))

B ⇒ id
B ⇒ A ⇒ (A)

"choose production based on hd toklis" (cont.)

$A \rightarrow A + id$

- $\vec{X}$ **is** *nullable* **if it can derive** ϵ**.**
- **Define:** **FIRST(**$\vec{X}$**)** $= \{t \in T \mid \vec{X} \Rightarrow^* t\alpha$ **for some** $\alpha\}$ $\cup$ $\{\bullet \mid \vec{X}\ nullable\}$**.**
- **Define: G is** *left-recursive* **if** $\exists A : A \Rightarrow^+ A\alpha$ **for some** α**.**

  ↑ sequence of grammar symbols
- **Define: G is** $LL(1)$ **if**
  1. **G is not left-recursive, and**
  2. **For all non-terminals** A**, if the productions of** A **are** $A \rightarrow \vec{X} \mid \ldots \mid \vec{Y}$**, the sets FIRST(**$\vec{X}$**), ..., FIRST(**$\vec{Y}$**) are pairwise disjoint.**

FIRST(A) = { (} , FIRST(B) = { id, (}

Top-down parsing revisited

If G is LL(1), then for each non-terminal A with productions

$A \rightarrow \vec{X} \mid \vec{Y} \mid \ldots \mid \vec{Z}$

construct `parseA`:

```
parseA toklis = let t = hd toklis in
      if t ∈ FIRST(X⃗) then handle X⃗
      else if t ∈ FIRST(Y⃗) then handle Y⃗
      ...else if t ∈ FIRST(Z⃗) or • ∈ FIRST(Z⃗) then handle Z⃗
                (Z⃗ is the unique nullable right-hand side of A, if any)
           else error
```

handle $X_1, X_2, \ldots, X_n$: handle X_1; handle X_2; ...; handle X_n

handle `t` : `if hd toklis = t`
`then` remove `t` and continue
`else` error

handle B : `parseB toklis`

Transformation to LL(1)

- **Left refactoring:**

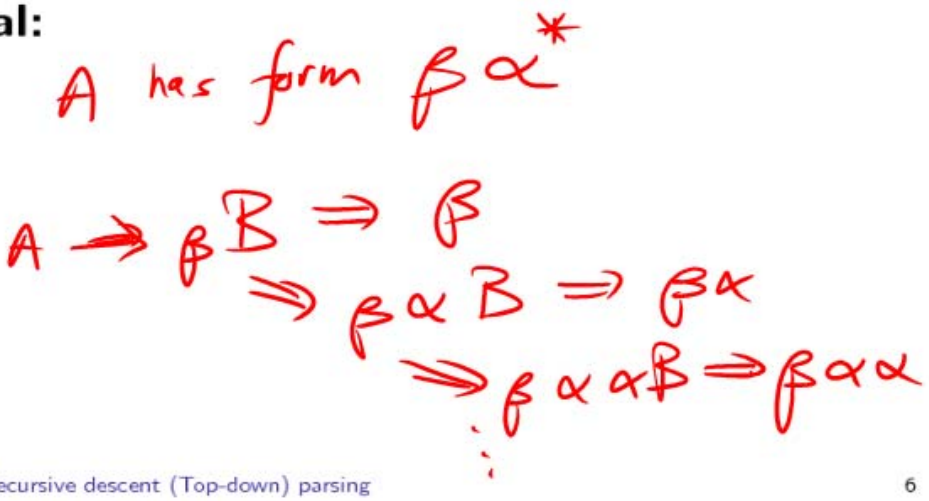
$$A \rightarrow \alpha\beta \mid \alpha\gamma$$

$$\Rightarrow \quad A \rightarrow \alpha B$$
$$B \rightarrow \beta \mid \gamma$$

- **Left-recursion removal:**

$$A \rightarrow A\alpha \mid \beta$$

$$\Rightarrow \quad A \rightarrow \beta B$$
$$B \rightarrow \epsilon \mid \alpha B$$



Example

- **Consider non-LL(1) grammar 3 from the previous class:**

  A $\rightarrow$ **id** | **'('** **B** **')'**
  B $\rightarrow$ **A** | **A** **'+'** **B**

- **Grammar 3 transformed to LL(1) form:**

  A $\rightarrow$ **id** | **'('** **B** **')'**
  B $\rightarrow$ **A** **C**
  C $\rightarrow$ **'+'** **A** **C** | ϵ

  } left-factoring

Ambiguity

- No test for ambiguity
- Recursive descent and LR(1) parsing not applicable to ambiguous grammar (possible to "cheat" with LR parser - will see how next week)

Expression grammars

- Expressions are challenging for several reasons:
  - Should be LL(1) and LR(1)
  - Grammar should enforce precedence, if possible
  - Grammar should enforce associativity, if possible
  - Grammar shouldn't be ambiguous
  - Should be easy to construct *abstract* syntax tree
- Especially hard to write LL(1) parser for expressions. Not so hard for LR(1).

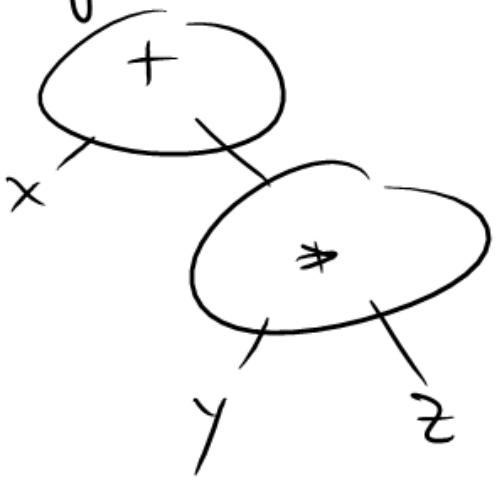
Enforcing precedence

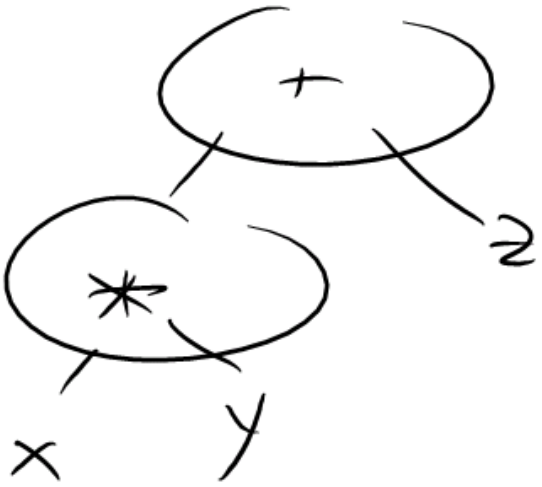
Consider

x + y * z

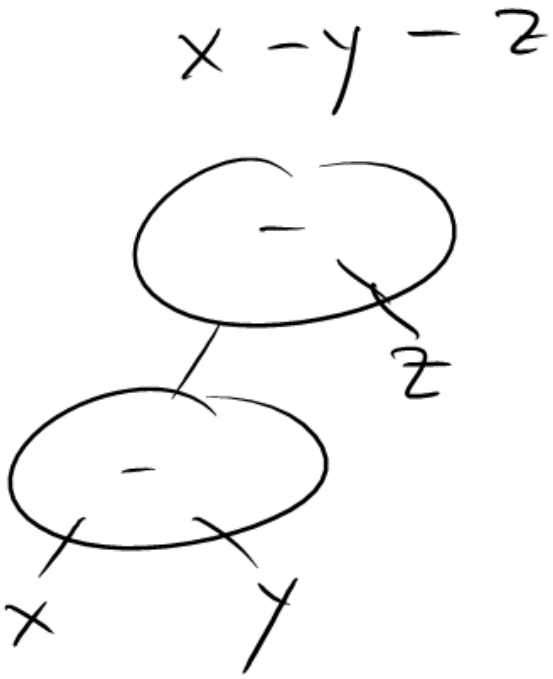
x * y + z

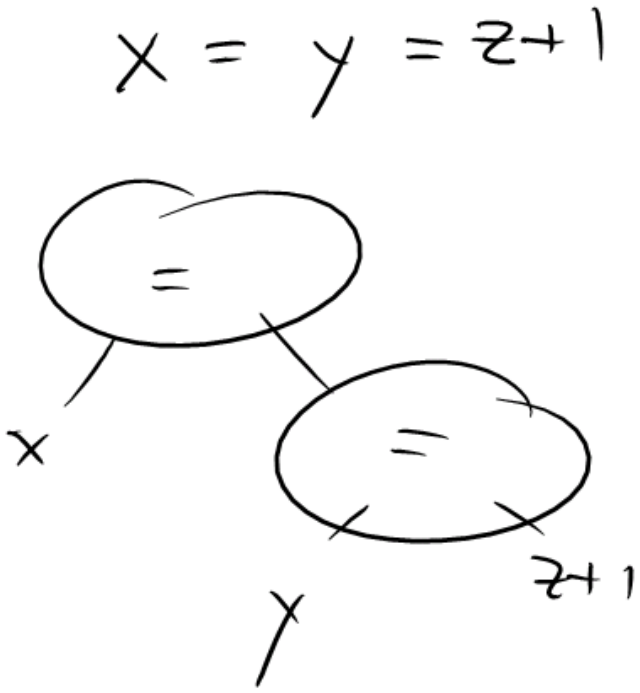
Should parse





Enforcing associativity





Some expression grammars

G_A: E → id | E - E | E * E

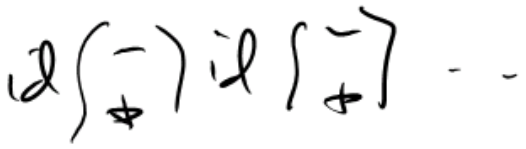
G_B: E → id | id - E | id * E

G_C: E → id | E - id | E * id

G_D:
E → T - E | T
T → id | id * T

G_E:
E → E - T | T
T → id | T * id

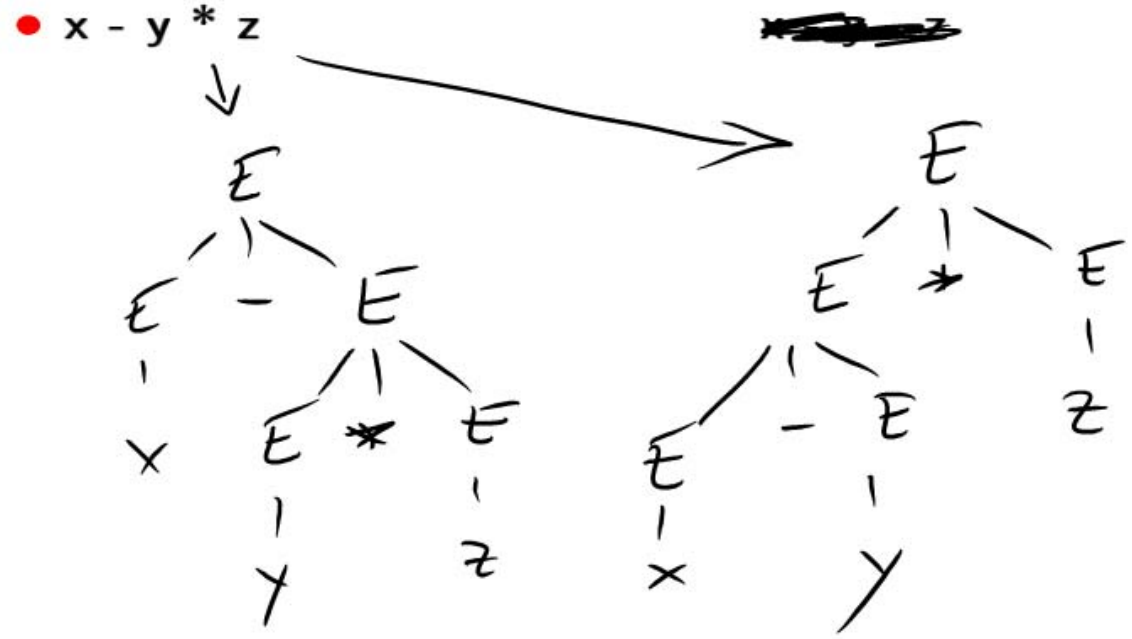
G_F:
E → T E′
E′ → ϵ | - E
T → id T′
T′ → ϵ | * T



- G_A: **E → id | E - E | E * E**

- Ambiguous
- Does not enforce precedence or associativity

- **x - y * z**

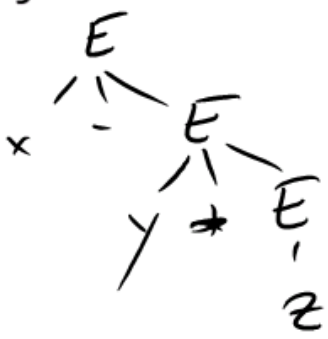


- G_B: **E → id | id - E | id * E**

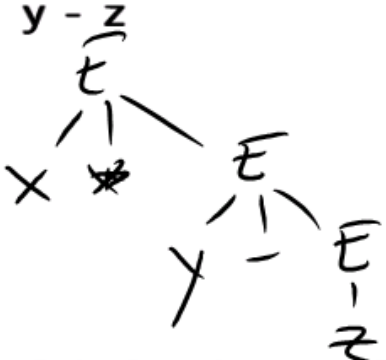
. Unambiguous
. LR(1)
. Not LL(1), but could be left-factored

. No precedence
. Right associativity

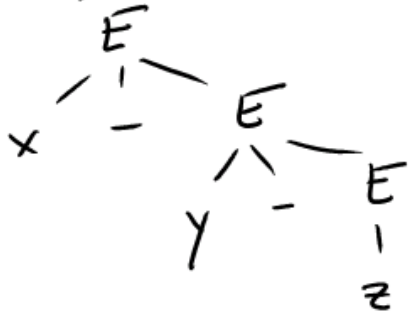
- **x - y * z**



- **x * y - z**



- x - y - z

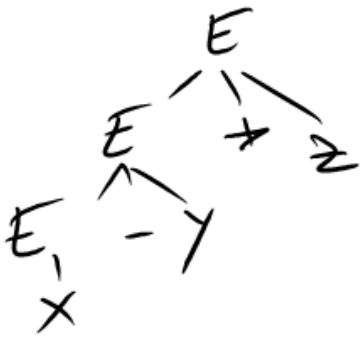


- G_C: **E → id | E - id | E * id**

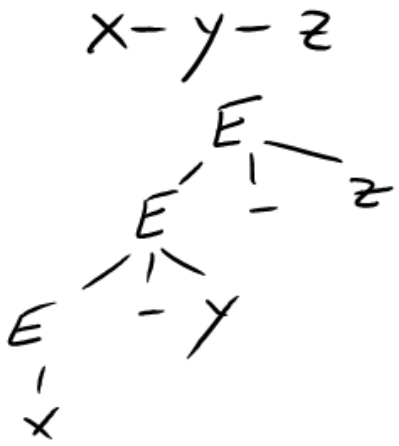
- Unambiguous
- LR(1), not LL(1) because left-recursive

- No precedence
- Enforces left-associativity

- **x - y * z**



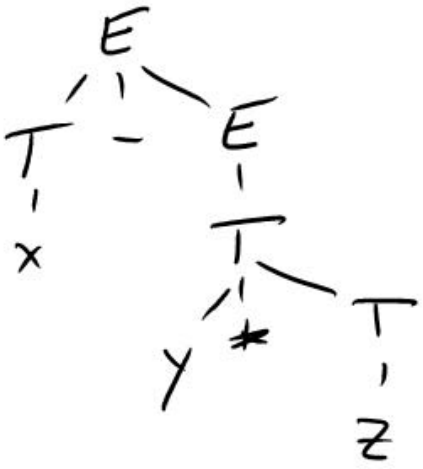
- **x * y - z**



- G_D: **E → T - E | T**
  T → id | id * T

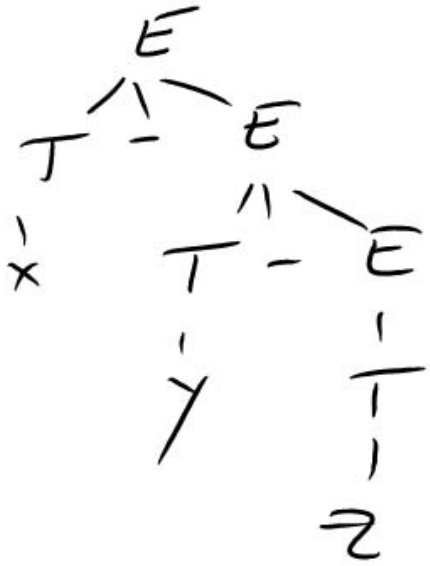
. Unambiguous
. LR(1), not LL(1) (but left-factorable)
. Enforces precedence
. Enforces right-assoc.

- **x - y * z**



x * y - z

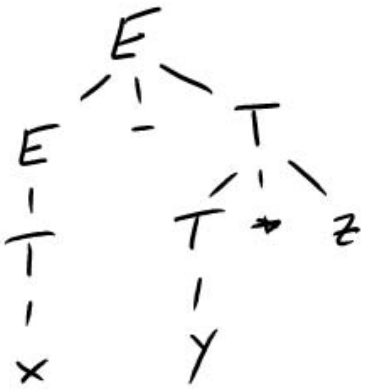
x - y - z



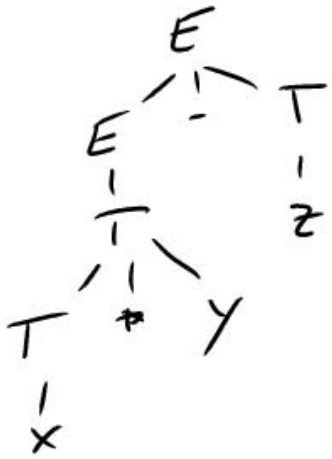
- G_E: **E → E - T | T**
  T → id | T * id

- Unambiguous
- LR(1), not LL(1) (left-recursive)
- Enforces prec.
- Enforces left-assoc.

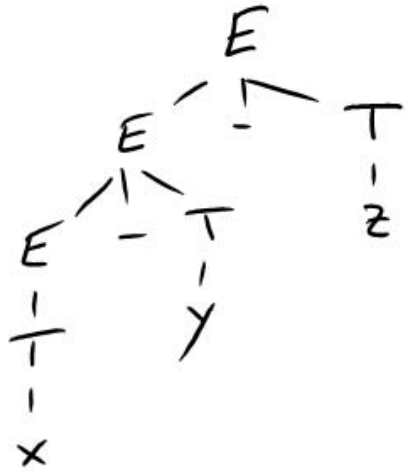
- **x - y * z**



x * y - z



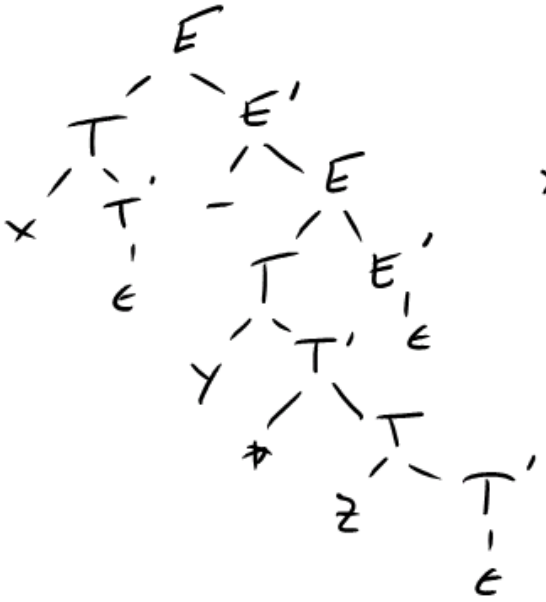
x - y - z



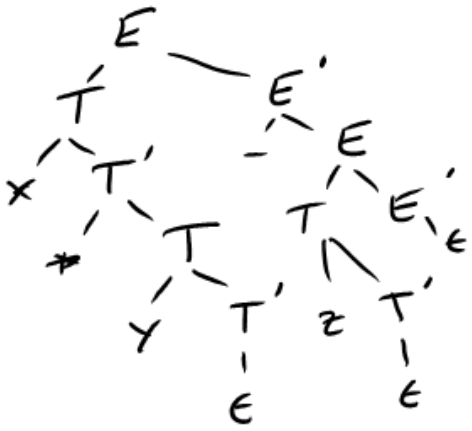
- G_F: **E** $\rightarrow$ **T E′**
  E′ $\rightarrow \epsilon \mid$ **- E**
  T $\rightarrow$ **id T′**
  T′ $\rightarrow \epsilon \mid$ *** T**

Left-factored version of G_D
- Unambiguous
- LR(1), LL(1)
- Enforces precedence
- Enforces right-asso

- **x - y * z**



x * y - z



x - y - z