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:**

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

define parseA:

`parseA toklis` = choose production based on hd toklis:
if $A \rightarrow \vec{X}$ chosen: handle $\vec{X}$
else if $A \rightarrow \vec{Y}$ chosen: handle $\vec{Y}$,
else if *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"

- Need to formalize some things...
- Define "$\Rightarrow$": $X_1 \ldots X_n \Rightarrow X_1 \ldots X_{i-1} \alpha X_{i+1} \ldots X_n$ (for any $1 \leq i \leq n$) if the grammar has production $X_i \rightarrow \alpha$.
- $\Rightarrow^+$ and $\Rightarrow^*$ are the transitive and reflexive-transive closures of $\Rightarrow$. (Say $\vec{X}$ *derives* α if $\vec{X} \Rightarrow^* \alpha$.)
- α is a *sentential form* of G if the start symbol of G derives α. If, furthermore, α consists solely of tokens, then it is a *sentence*. (These notions correspond to being the "frontier" of a syntax tree; some care is needed in defining "frontier" to account for ϵ-productions.)

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

- $\vec{X}$ **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: FOLLOW(**A**)** $= \{t \in T \mid \exists$ **a sentential form** $\alpha A t \beta\}$

∪ { eof | A can appear as the last symbol in a sentential form }

There are well-known algorithms for calculating FIRST and FOLLOW sets, but we will consider only simple cases where they can be calculated by inspection.

$L \rightarrow (\; M \;)$

$M \rightarrow \epsilon \mid id \; N$

$N \rightarrow \epsilon \mid , \; id \; N$

$L \Rightarrow (M)$

$\Rightarrow (id \; N)$

$\Rightarrow (id , id \; N)$

FIRST((M)) = { (}

FIRST(ϵ) = { • }

FIRST (id N) = { id }

FIRST(, id N) = { , }

FOLLOW (L) = { eof }

FOLLOW (M) = {) }

FOLLOW (N) = {) }

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

- **Define: G is *left-recursive* if $\exists A : A \Rightarrow^{+} A\alpha$ for some α.**
- **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 $A \rightarrow \alpha_1 \mid \ldots \mid \alpha_n$:**
   (a) **The sets FIRST(α_1), ..., FIRST(α_n) are pairwise disjoint. ($\forall i, j . i \neq j \Rightarrow \alpha_i \cap \alpha_j = \emptyset$)**
   (b) **If A is nullable, then suppose α_i is the unique right-hand side such that $\bullet \in$ FIRST(α_i). Then, for all $j \neq i$, $\alpha_j \cap$ FOLLOW(A) $= \emptyset$.**

FIRST(α_j)

should be: FIRST(α_i) $\cap$ FIRST(α_j) $= \emptyset$

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) and t ∈ FOLLOW(A))
            then handle Z⃗ (Z⃗ 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:**

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

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

$A \Rightarrow \beta$
$\Rightarrow A\alpha \Rightarrow \beta\alpha$
$\Rightarrow A\alpha \Rightarrow A\alpha\alpha \Rightarrow \beta\alpha\alpha$

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 $\rightarrow$ '+' **B** | ϵ

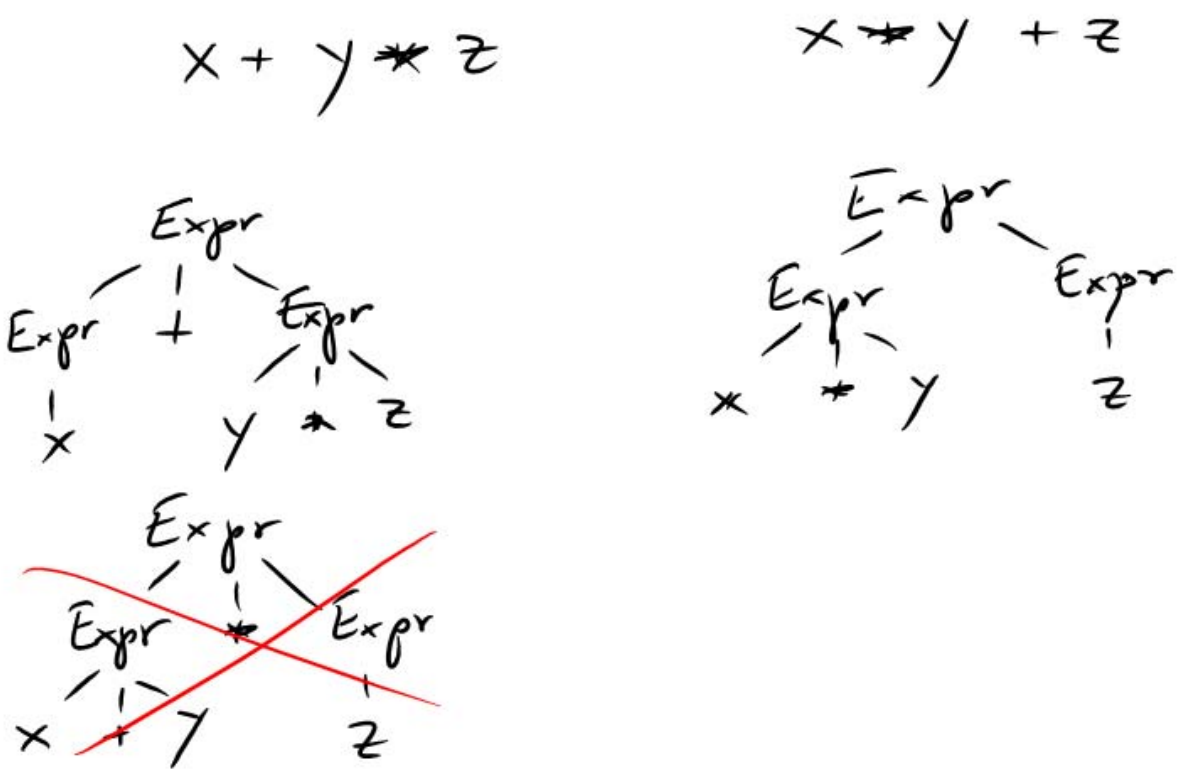
Ambiguity

- 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)

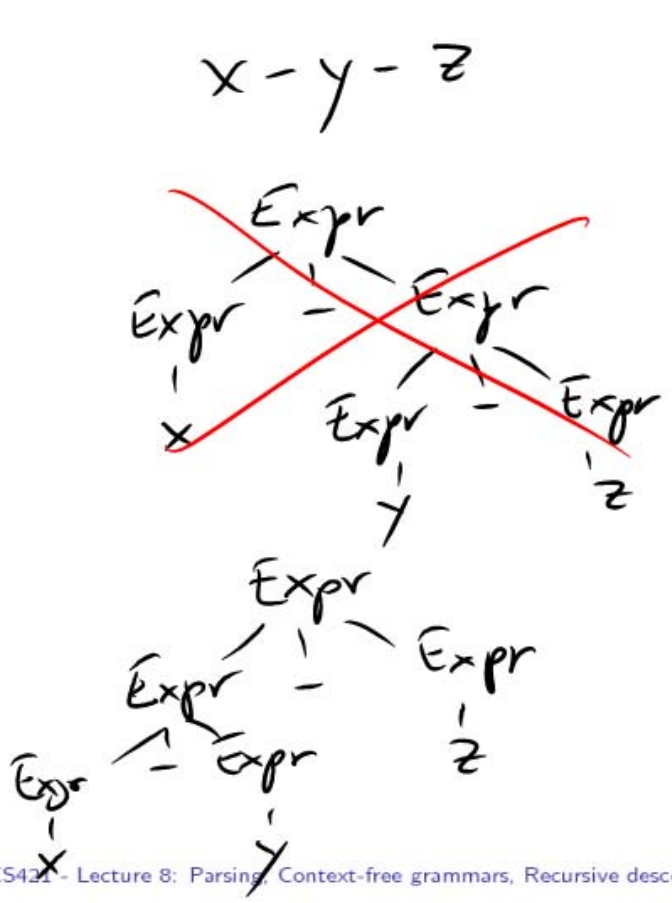
Expression grammars

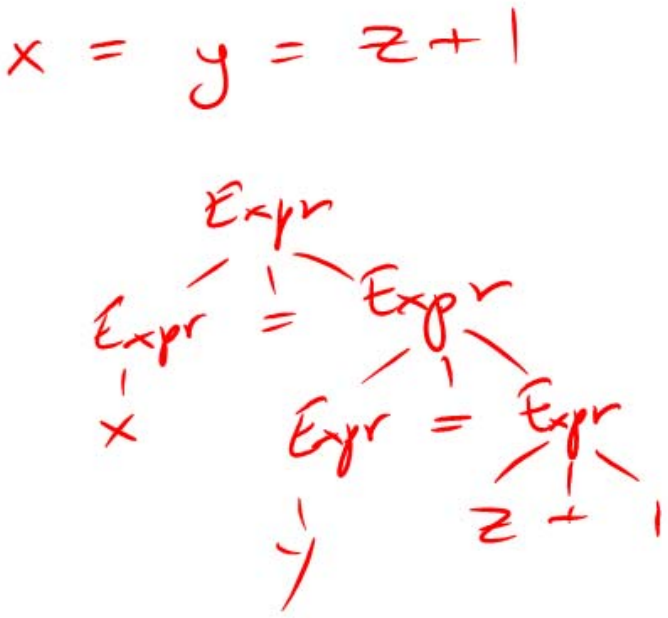
- Expressions are challenging for several reasons:
  - Grammar should enforce precedence, if possible
  - Grammar should enforce associativity, if possible
  - Grammar shouldn't be ambiguous
  - Should be easy to construct *abstract* syntax tree
- Especially hard to write LL(1) parser for expressions.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

Enforcing precedence



Enforcing associativity





Some expression grammars

G_A: E $\rightarrow$ id | E - E | E * E

G_B: E $\rightarrow$ id | id - E | id * E

G_C: E $\rightarrow$ id | E - id | E * id

G_D: E $\rightarrow$ T - E | T
T $\rightarrow$ id | id * T

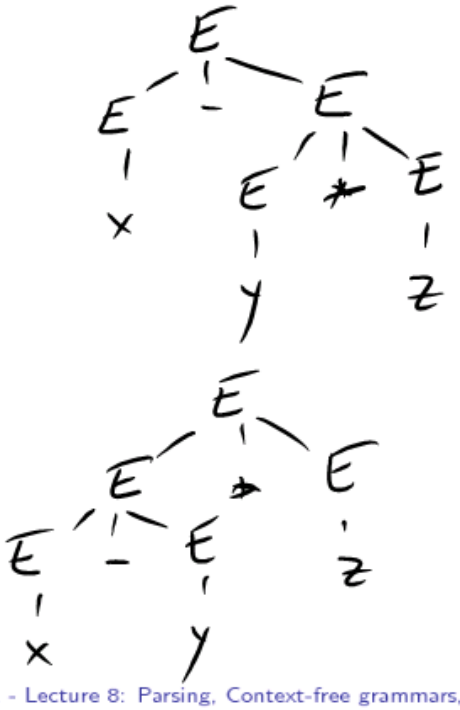
G_E: E $\rightarrow$ E - T | T
T $\rightarrow$ id | T * id

G_F: E $\rightarrow$ T E$'$
E$'$ $\rightarrow$ ϵ | - E
T $\rightarrow$ id T$'$
T$'$ $\rightarrow$ ϵ | * T

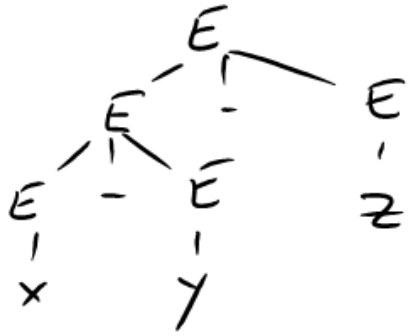
- G_A: **E** → **id** | **E - E** | **E * E**

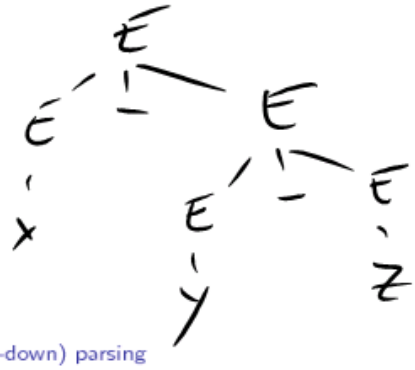
- Ambiguous
- Does not enforce anything
- Not LL(1) or LR(1)

- **x - y * z**



x - y - z



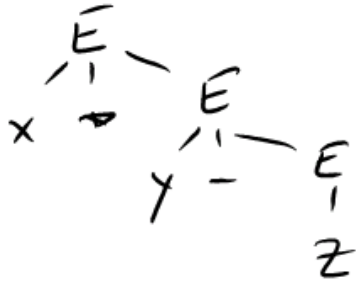


- G_B: **E → id | id - E | id * E**

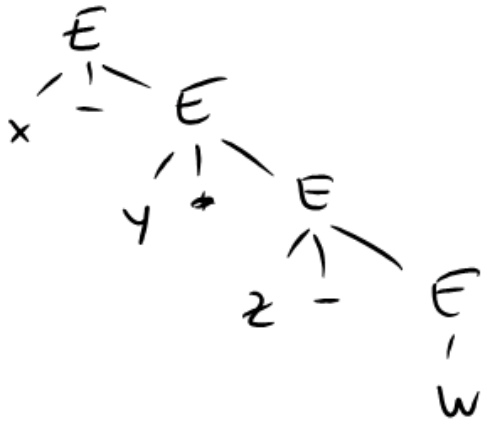
- Unambiguous
- Does not enforce anything
- Not LL(1) - but can left factor

- **x - y * z**

- **x * y - z**



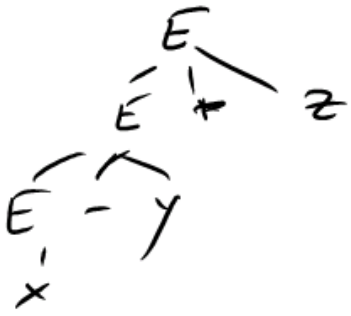
x - y * z - w



- G_C: **E → id | E - id | E * id**

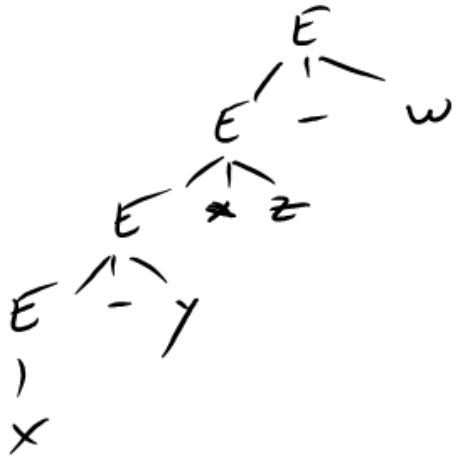
- Unambig.
- Does not enforce anything
- Not LL(1) (left recursive)

- **x - y * z**



- **x * y - z**

x - y * z - w

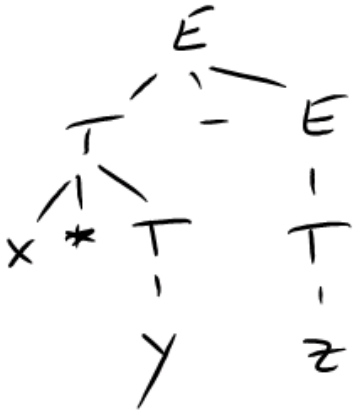


- G_D: **E → T - E | T**
  T → id | id * T

- Unamb.
- Enforces precedence
- Enforces right assoc.
- LL(1)? No - can left-factor.

- **x - y * z**

x * y - z

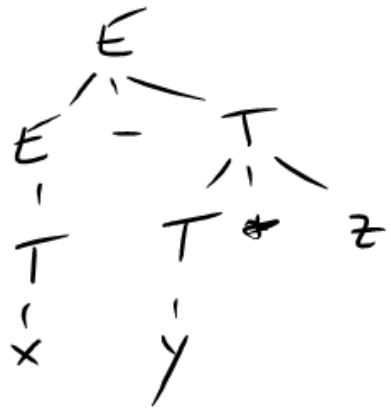


x - y - z

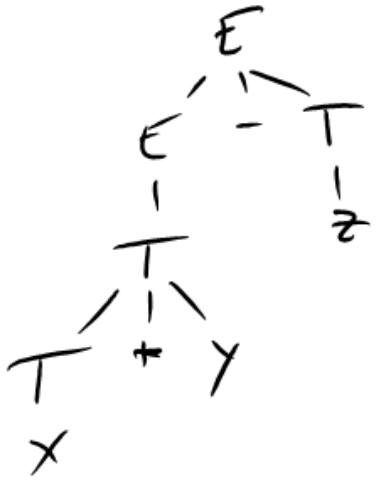
- G_E: **E → E - T | T**
  T → id | T * id

- Unamb.
- Enforces prec. & left assoc.
- LL(1)? No - left recursive

- **x - y * z**



x * y - z

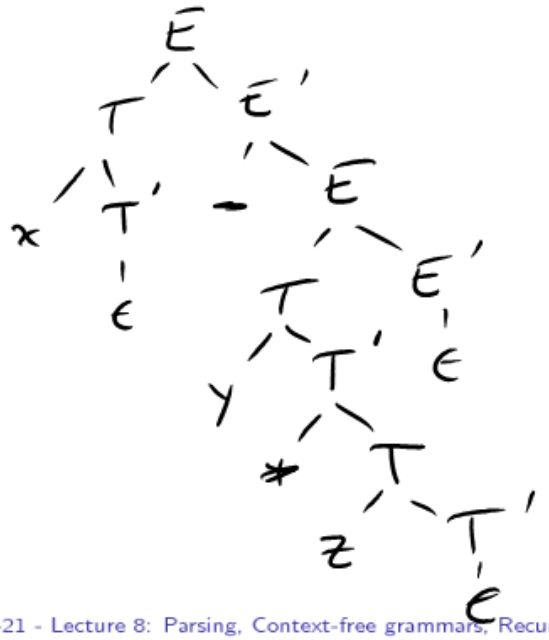


x - y - z

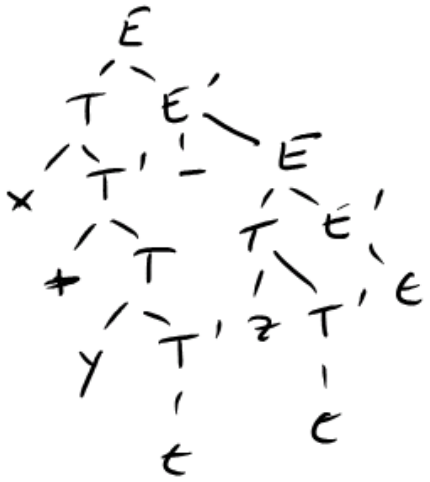
- G_F: **E** $\rightarrow$ **T E′**
  E′ $\rightarrow \epsilon \mid$ **- E**
  T $\rightarrow$ **id T′**
  T′ $\rightarrow \epsilon \mid$ *** T**

- Unamb.
- Enforce prec, right assoc.
- LL(1) ? Yes

- **x - y * z**



x * y - z



x - y - z