CS 344 - LL and LR grammars

Note Title 2/3/2012

Announcements

- HW3 - due Sunday by midnight
- Next HW up this weekend

Other parsing algorithms

CYK is still pretty slow, especially for large programs.

After it was developed, a lot of work was put into figuring out what grammars could have faster algorithms.

Two big (& useful) classes have $O(n)$ time parsers: LL & LR.

LL & LR grammars

"LL" is left-to-right, leftmost derivation

"LR" is left-to-right, rightmost derivation

- So parser will scan left to right either way.
- LL will make a leftmost derivation (so right-leaning tree)

LL versus LR

- LL are a bit simpler so we'll start with them
  - Note: LR is a larger class (so more grammers are LR than are LL)
- Both are used in production compilers today

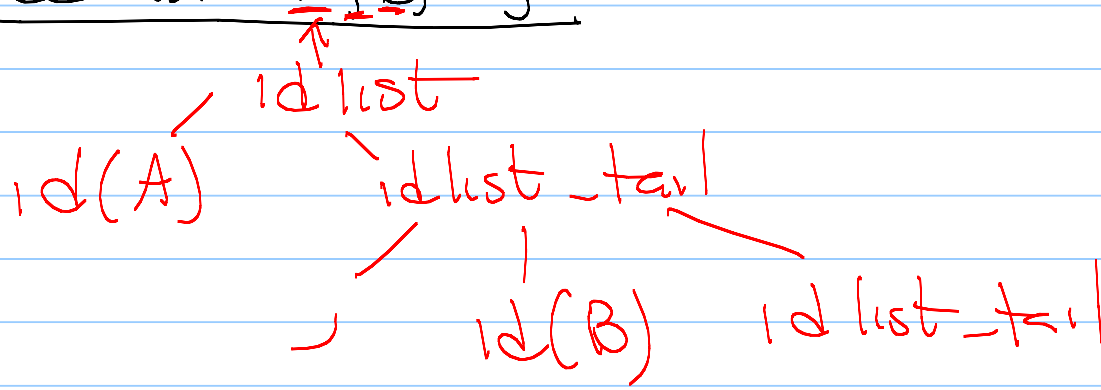
Example: LL parsing

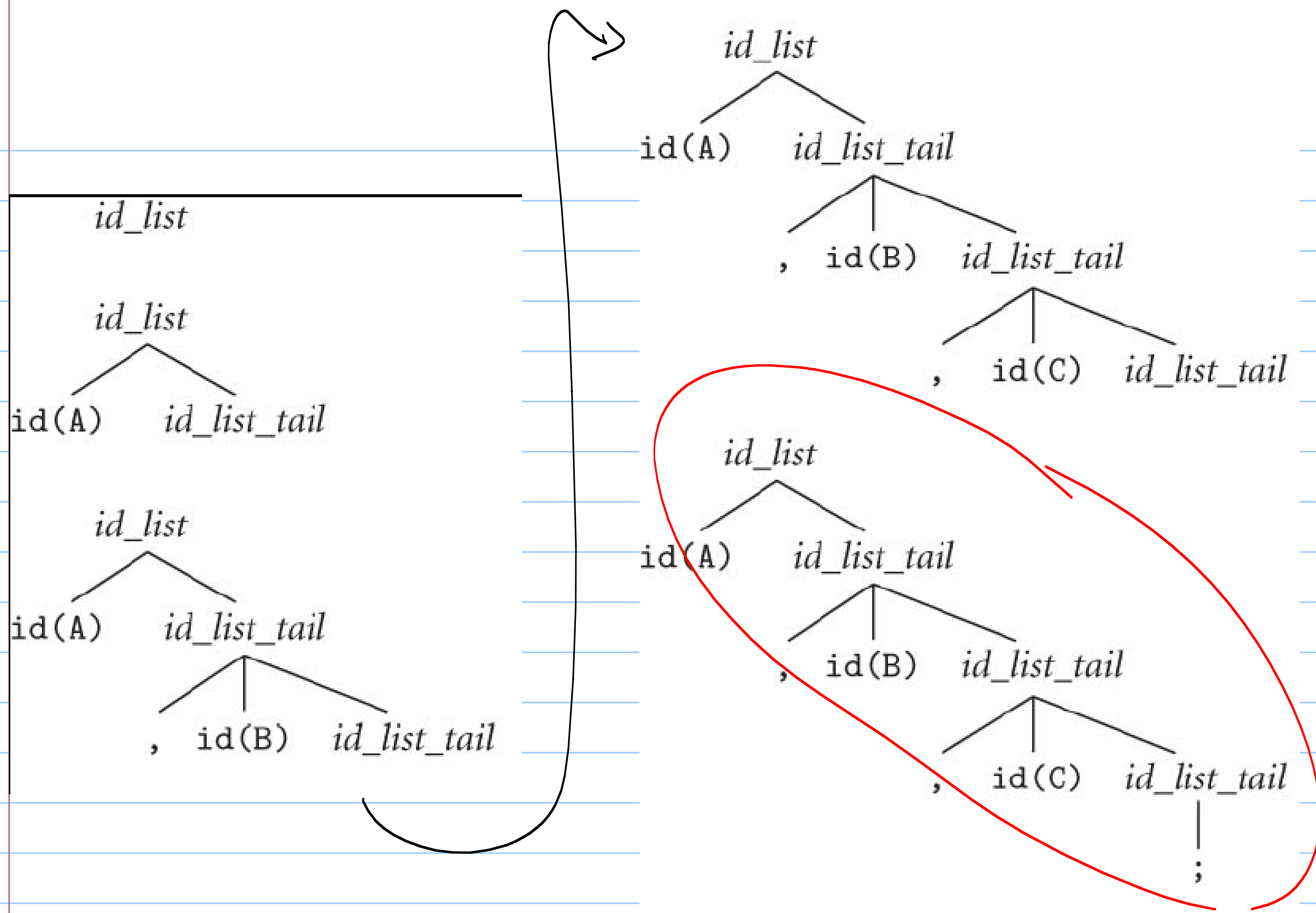
idlist → id idlist_tail

idlist_tail → , id idlist_tail

idlist_tail → ;

Parse tree for "A, B, C;"




id_list
id_list
id(A)
id_list_tail
id_list
id(A)
id_list_tail
,
id(B)
id_list_tail
id_list
id(A)
id_list_tail
,
id(B)
id_list_tail
,
id(C)
id_list_tail
id_list
id(A)
id_list_tail
,
id(B)
id_list_tail
,
id(C)
id_list_tail
;

LL(k) & LR(k)

When LL or LR is written with (1), (2), etc, it refers to how much look-ahead is allowed.

LL(1) means we can only look 1 token ahead when making our decision of which rule to match

Most commercial ones are LR(1), but exceptions exist (such as ANTLR).

A non LL(1) example: Left recursion

id_list → id
→ id_list , id

left recursion is bad in LL

Imagine: Scanning left to right, & encounter an id token.

Which parse tree do we build?

id(A), id(B), id(C)

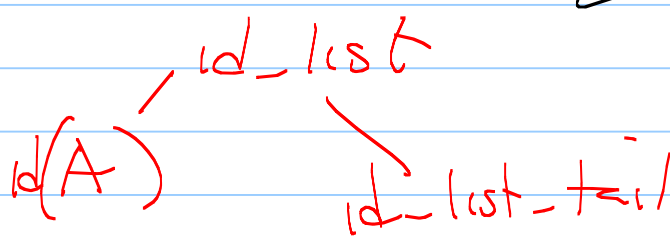
(this is LL(2))

Making the grammar LL(1):

id_list → id id_list_tail

id_list_tail → , id id_list_tails

→ ε



Another non-LL(0) example: common prefixes

stmt → id := expr
stmt → id (arguement_list)

So when next token is an id, don't know which rule to use.

Fix?

stmt → id stmt_tail

stmt_tail → := expr
→ (- - - -)

Some grammars are non-LL:

- Eliminating left recursion and common prefixes is a very mechanical procedure which can be applied to any grammar.
- However, might not work! There are examples of inherently non-LL grammars.
- In these cases generally add some heuristic to deal with odd cases (or use CYK)

Example: non-LL language: optional else

stmt → if condition then_clause else_clause

then_clause → then stmt

else_clause → else stmt

→ ε

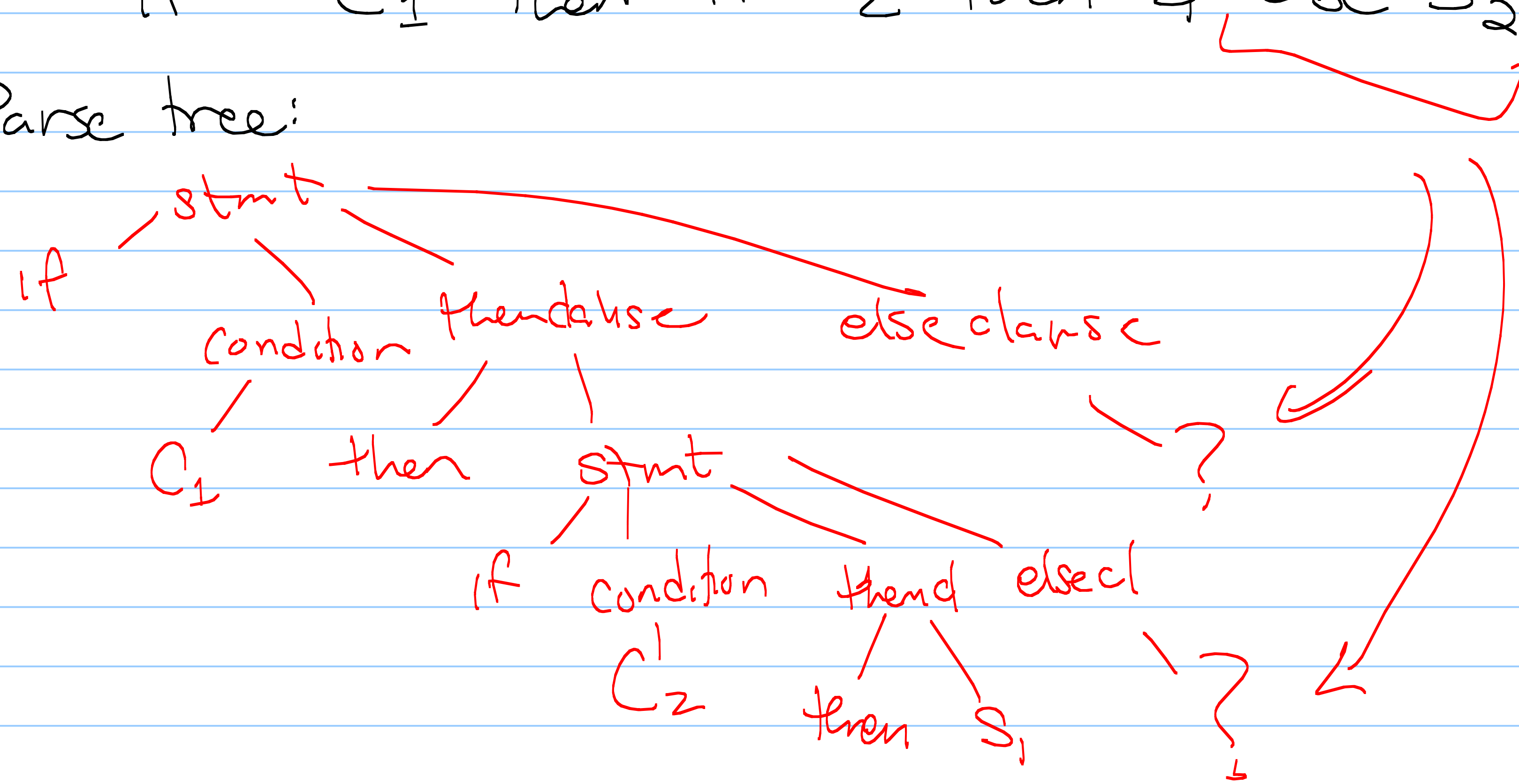
What syntax?

if-else statement

(PASCAL)

Ex: if C_1 then if C_2 then S_1 else S_2

Parse tree:



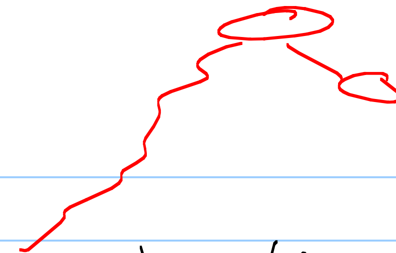
Back to LL-parsing

We have seen mostly **top-down** parsing.

Start with So, the start token, & try to construct the tree based on the next input.

Also called predictive parsing – matches the rule based on current token/state plus the next input.

LR grammers



Bottom-up parsing starts at the leaves (here, the tokens), & tries to build the tree upward.

Continues scanning & shifting tokens onto a forest, then builds up when it finds a valid production.

Never predicts – when it recognizes right hand side of a rule, simplifies to left hand side.

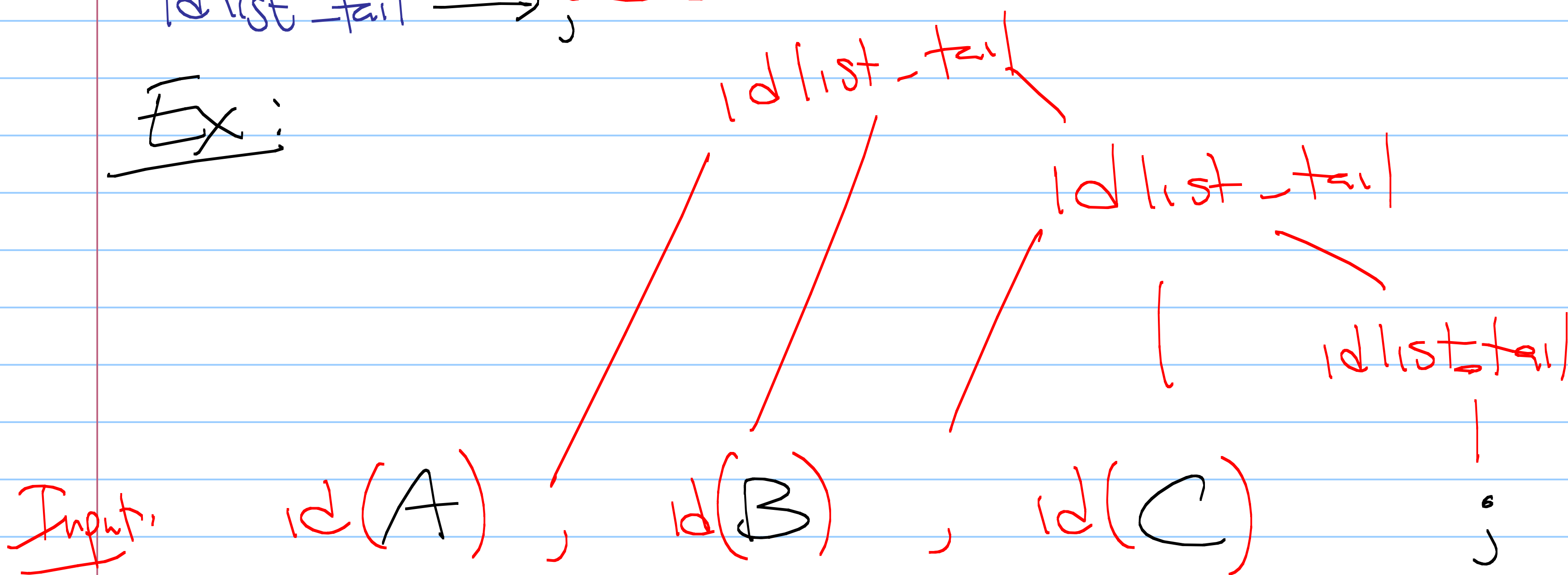
Bottom-up parsing (LR parsing)

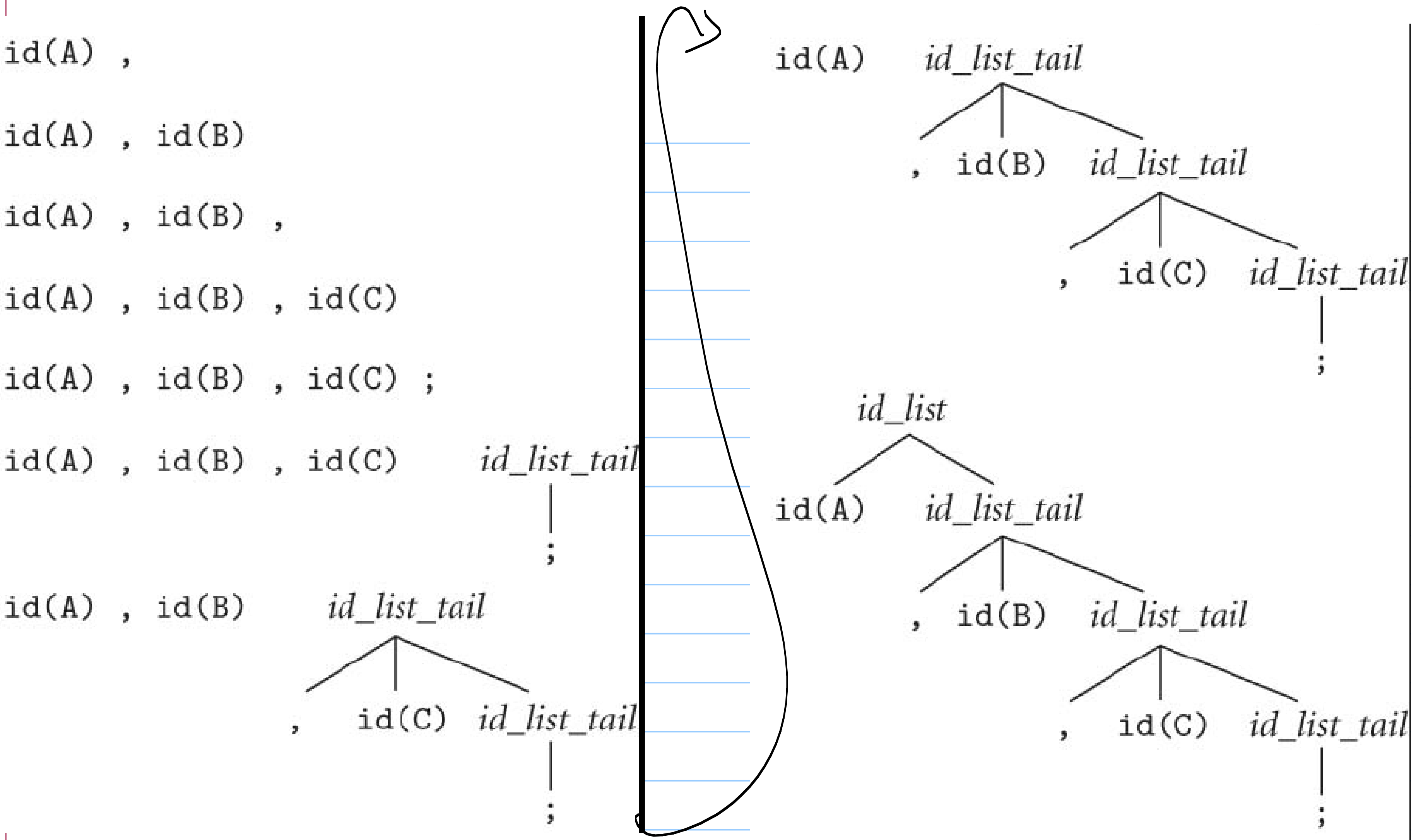
idlist → id idlist_tail

idlist_tail → , id idlist_tail

idlist_tail → ;

Ex:




id(A) ,
id(A) , id(B)
id(A) , id(B) ,
id(A) , id(B) , id(C)
id(A) , id(B) , id(C) ;
id(A) , id(B) , id(C) id_list_tail
;
id(A) , id(B) id_list_tail
, id(C) id_list_tail
;
id(A) id_list_tail
, id(B) id_list_tail
, id(C) id_list_tail
;
id_list
id(A) id_list_tail
, id(B) id_list_tail
, id(C) id_list_tail
;

Shift-reduce:

- Bottom up parsers are also called shift-reduce:
  - shift token onto stack (in a forest)
  - when a rule is recognized, reduce to left-hand side

- Problem with last example: must shift <u>all</u> tokens onto the forest before reducing. What could happen in a large program?
  overflow memory

- Sometimes unavoidable. However, sometimes other options...

Bottom-up parsings: another example

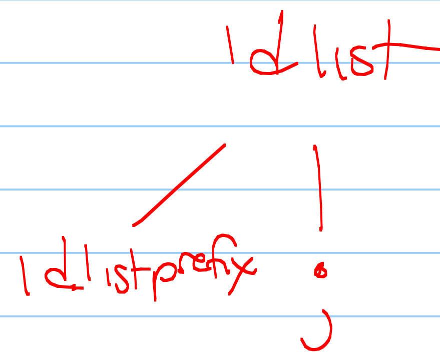
left recursion!

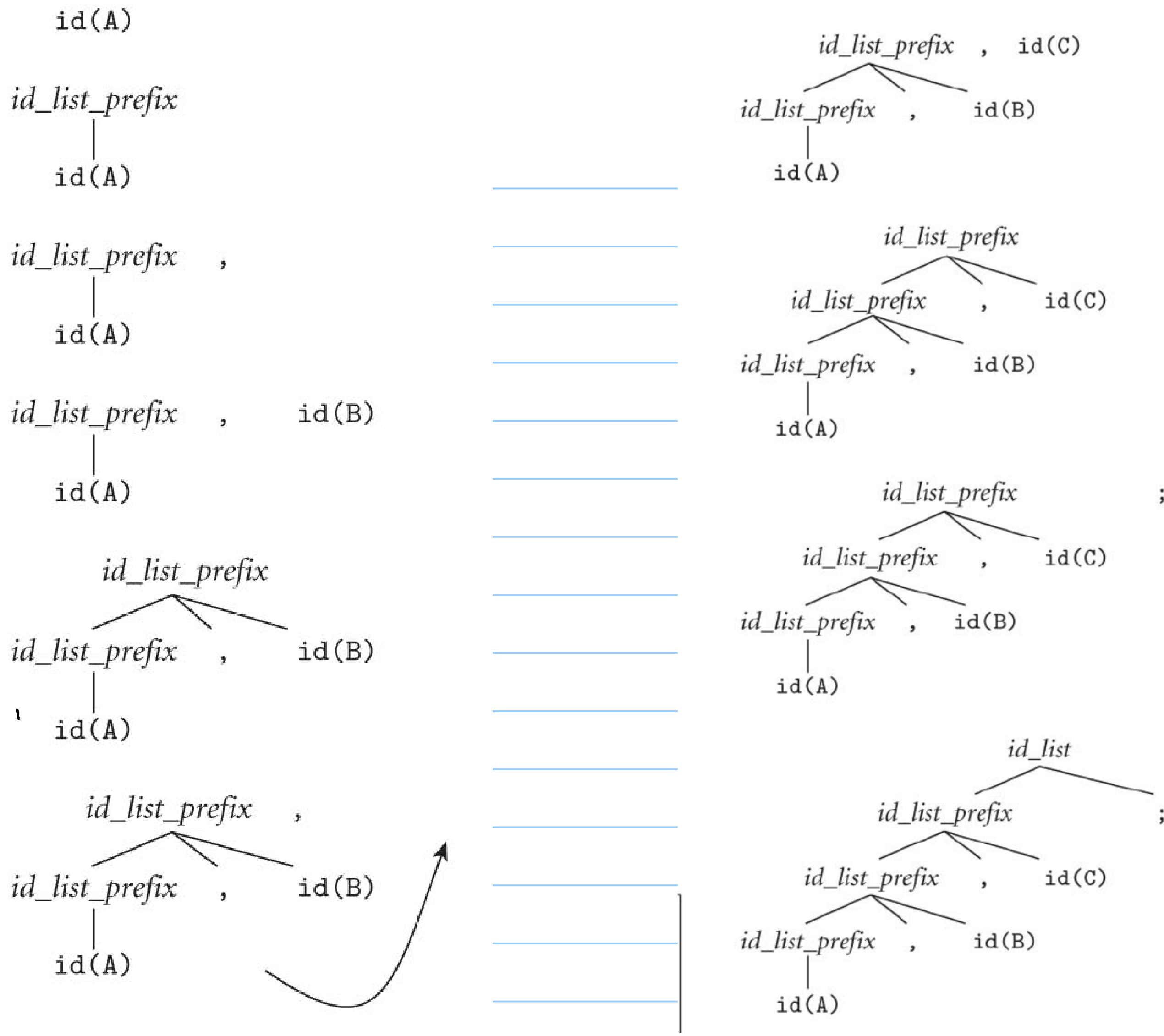
id_list → id_list_prefix ;

id_list_prefix → id_list_prefix , id

→ id

Parse A , B , C ; again, bottom-up:





never put all tokens on stack!

much better in terms of space.

Bottom-up parsing: some notes

- The previous example cannot be parsed top-down. Why? left recursion!
- Note that it also is not an LL grammar, although the language is LL.
- There is a distinction between a language & a grammar. Remember, any language can be generated by an infinite number of grammars.

LR grammers: An old example

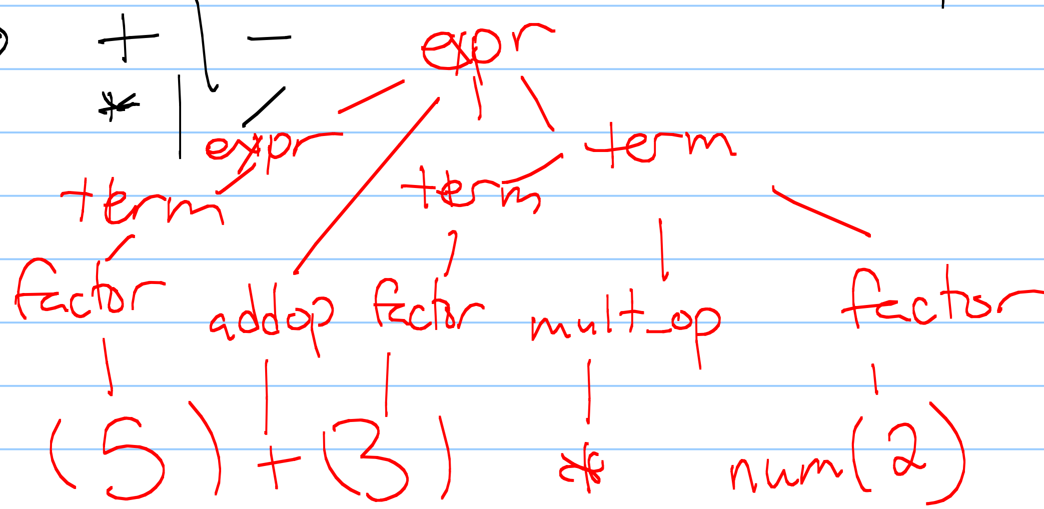
expr → term | expr add_op term

term → factor | term multop factor

factor → id | number | − factor | (expr)

add_op → + | −

mult_op → * | /



This grammar is not LL!

- If we get an id as input when expecting an expr, no way to choose between the 2 possible productions.
- It suffers from the common prefix issue we saw before.

(We can fix this ⟶)

Another LL example:

expr → term term_tail

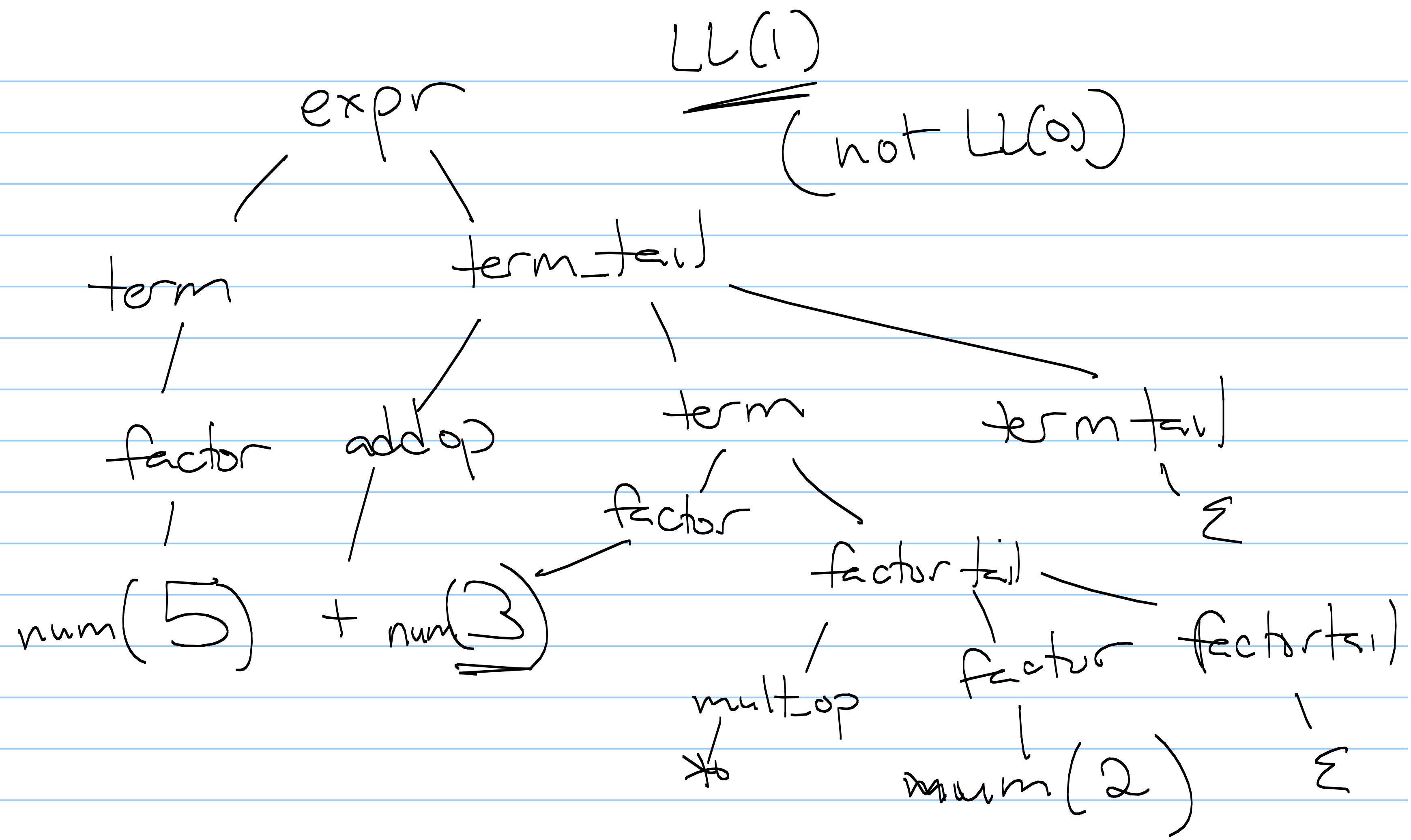
term_tail → add_op term term_tail
→ ε

→ term → factor factor_tail

factor_tail → mult_op factor factor_tail
→ ε

factor → (expr) | id | number

add_op → + | -
mult_op → * | /


LL(1)
(not LL(0))
expr
term
term_tail
factor
addop
term
term_tail
num(5)
+
num(3)
factor
factor_tail
ε
mult_op
factor
factor_tail
*
num(2)
ε

Now can add this as part of a simple calculator language:

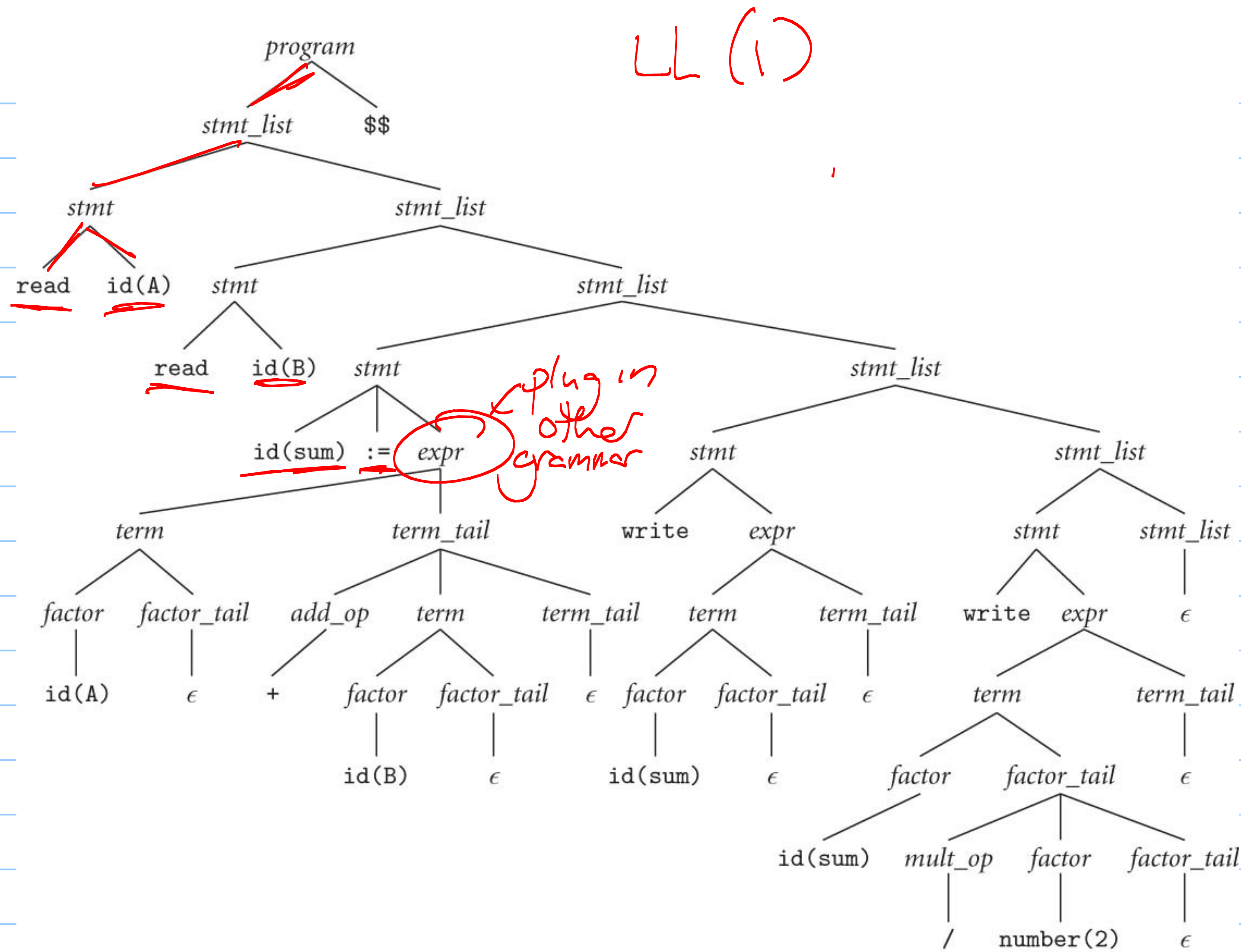
$$
\begin{aligned}
program &\to stmt\_list \ \$\$ \quad \leftarrow \text{end of file} \\
stmt\_list &\to stmt \ \ stmt\_list \\
&\to \varepsilon \\
stmt &\to id := expr \\
&\to read \ \ id \\
&\to write \ \ expr
\end{aligned}
$$

Program: What does it do?

```
read A
read B
sum := A + B
write sum
write sum / 2
```

expr (A + B)

average (sum / 2)


program
stmt_list
$$
stmt
stmt_list
read
id(A)
stmt
stmt_list
read
id(B)
stmt
stmt_list
id(sum)
:=
expr
stmt
stmt_list
term
term_tail
write
expr
stmt
stmt_list
factor
factor_tail
add_op
term
term_tail
term
term_tail
write
expr
ϵ
id(A)
ϵ
+
factor
factor_tail
ϵ
factor
factor_tail
ϵ
term
term_tail
id(B)
ϵ
id(sum)
ϵ
factor
factor_tail
ϵ
id(sum)
mult_op
factor
factor_tail
/
number(2)
ϵ
LL(1)
plug in other grammar