# Homework 2

2.1 1) Prove that following grammar is LL(1):

decl → ID decl_tail

decl_tail → , decl

→ : ID ;

A formal proof is unnecessary, just the reasoning must be sufficient.

For the production:
decl —> ID decl_tail
There is only one production, so there is no conflict.

For the productions:
decl_tail —> , decl
—> : ID ;
The first symbols in the two alternations are different. So when the parser applies the one token look-ahead, it'll know exactly which production to apply.

Additionally, there is no presence of left recursion.

So the grammar satisfies the LL(1) condition.

c) To check if it's LL(1), we can determine the FIRST/FOLLOW sets and see whether each nonterminal's productions are distinguishable by a single look-ahead token.

$M \to S \mid \varepsilon$:

- If the next input symbol is a, it could mean "start a new S (since S can begin with a)" or it could mean "go ε in M and let the next higher rule handle the a."

A similar FIRST/FOLLOW conflict occurs in E, which can produce a B, b A, or go empty. If the lookahead is a, it is not clear whether E should expand to a B or vanish (so the a is parsed by whatever follows E).

These ambiguities mean the grammar is not LL(1).

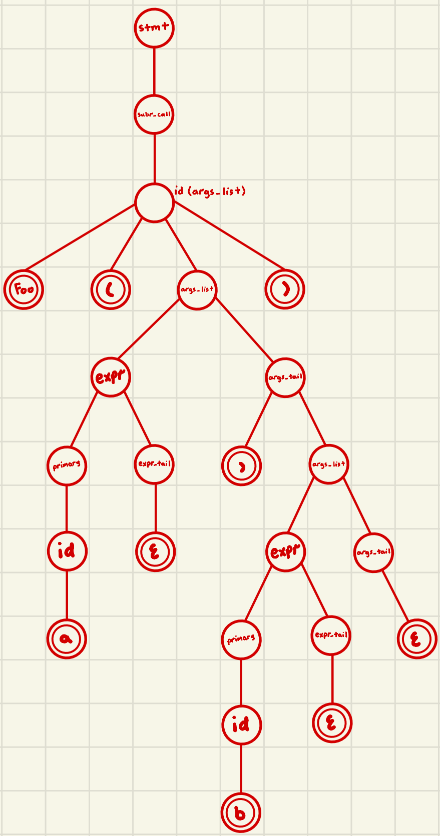
## 2.1 3) Consider the following grammar:

*stmt* ⟶ *assignment*
  ⟶ *subr_call*
*assignment* ⟶ `id` `:=` *expr*
*subr_call* ⟶ `id` `(` *arg_list* `)`
*expr* ⟶ *primary* *expr_tail*
*expr_tail* ⟶ *op* *expr*
  ⟶ $\epsilon$
*primary* ⟶ `id`
  ⟶ *subr_call*
  ⟶ `(` *expr* `)`
*op* ⟶ `+` | `-` | `*` | `/`
*arg_list* ⟶ *expr* *args_tail*
*args_tail* ⟶ `,` *arg_list*
  ⟶ $\epsilon$

a) Construct a parse tree for the input string foo(a, b).
c) Prove that the grammar is not LL(1).
d) Modify the grammar so that it is LL(1).

**NOTE:** Any LL(k) grammar (k > 1) can be converted to LL(1) using **left factorization** by restructuring or adding rules to group common prefixes, ensuring that a single token is sufficient to determine which rule to apply.

a)



b) The nonterminal primary has the following productions:

primary —> id

—> subr_call

and

subr_call→id(arg_list)

For id:

- FIRST(id) = { id }

For subr_call:

- Since subr_call starts with id, we get: FIRST(subr_call) = FIRST(id(arg_list)) = { id }

Since both productions have the same FIRST set, an LL(1) parser that looks ahead by only one token would not be able to distinguish whether it should expand primary as a single id or as a subr_call.

c) To make the grammar LL(1), we can factor out the common prefixes so that no two alternatives for the same nonterminal begin with the same token.

stmt —> id stmt_tail
stmt_tail —> := expr | ( arg_list )
expr —> primary expr_tail
expr_tail —> op expr | ε
primary —> id primary_tail | ( expr )
primary_tail —> ( arg_list ) | ε
arg_list —> expr args_tail
args_tail —> , arg_list | ε
op —> + | - | * | /

## 2.17) Extend the grammar of Figure 2.25 to include if statements and while loops, along the lines suggested by the following examples:

```
abs := n
if n < 0 then abs := 0 - abs fi

sum := 0
read count
while count > 0 do
    read n
    sum := sum + n
    count := count - 1
od
write sum
```

**Your grammar should support the six standard comparison operations in conditions, with arbitrary expressions as operands. It should also allow an arbitrary number of statements in the body of an if or while statement.**

## Figure 2.25:

1. *program* $\longrightarrow$ *stmt_list* `$$`
2. *stmt_list* $\longrightarrow$ *stmt_list stmt*
3. *stmt_list* $\longrightarrow$ *stmt*
4. *stmt* $\longrightarrow$ `id := ` *expr*
5. *stmt* $\longrightarrow$ `read id`
6. *stmt* $\longrightarrow$ `write` *expr*
7. *expr* $\longrightarrow$ *term*
8. *expr* $\longrightarrow$ *expr add_op term*
9. *term* $\longrightarrow$ *factor*
10. *term* $\longrightarrow$ *term mult_op factor*
11. *factor* $\longrightarrow$ `(` *expr* `)`
12. *factor* $\longrightarrow$ `id`
13. *factor* $\longrightarrow$ `number`
14. *add_op* $\longrightarrow$ `+`
15. *add_op* $\longrightarrow$ `-`
16. *mult_op* $\longrightarrow$ `*`
17. *mult_op* $\longrightarrow$ `/`

**Figure 2.25** **LR(1) grammar for the calculator language.** Productions have been numbered for reference in future figures.

1. program —> stmt_list $$
2. stmt_list —> stmt_list stmt
3. —> stmt
4. stmt —> id := expr
5. —> read id
6. —> write expr
7. —> if condition then stmt_list fi
8. —> while condition do stmt_list od
9. expr —> expr add_op term
10. —> term
11. term —> term mult_op factor
12. —> factor
13. factor —> ( expr )
14. —> id
15. —> number
16. add_op —> +
17. add_op —> -
18. mult_op —> *
19. mult_op —> /
20. condition —> expr rel_op expr
21. rel_op —> <
22. —> >
23. —> <=
24. —> >=
25. —> =
26. —> <>