Primeiro passo será criar as especificações da linguagem(a gramática), podemos usar apenas int e float,
depois extruturas condicionais if else, extrutura de repetição, escolher como finalizar essas extruturas, se com chaves ou end if ou end while.
Sobre as operações básicas +, -, *, e divisão, cuidado com a ordem de precedência.
Estruturas lógicas definir se pode misturar ou não.
OBS: Lembrando que o nossa gramática tá adaptada, ou seja um pouco misturada.
Garantir que seja LL1.
Seguindo essas orientações, ficará mais fácil e a geração de código.

**********Atualizando**********

Para especificar corretamente a gramática e garantir que ela seja LL(1), vamos revisar e ajustar a definição fornecida.
A gramática deve ser clara, não ambígua, e deve atender aos requisitos mencionados, como a manipulação de tipos de dados
(`int` e `float`), estruturas condicionais (`if-else`), estruturas de repetição (`while`), operadores aritméticos,
e lógica booleana. Vou organizar a gramática de forma a garantir que ela esteja em uma forma LL(1) e atenda às
especificações necessárias.

Gramática Ajustada

```python
import ply.yacc as yacc
from lexer import tokens

# Definição da classe Grammar (mantida)
```

```
class Grammar:
    # Classe mantida como no código original
    ...

# Definição das regras de gramática
def p_start(p):
    '''start : stmts'''

def p_stmts(p):
    '''stmts : stmt SEMI stmts
             | stmt'''

def p_stmt(p):
    '''stmt : assign
            | dccl
            | IF LPAREN expr RPAREN stmt else_opt FI
            | WHILE LPAREN expr RPAREN DO stmt OD
            | BEGIN stmts END'''

def p_else_opt(p):
    '''else_opt : ELSE stmt
                | empty'''

def p_dccl(p):
    '''dccl : INT ID
            | FLOAT ID'''

def p_assign(p):
    '''assign : ID ASSIGN expr'''

def p_expr(p):
    '''expr : expr_A expr_logic_prime'''

def p_expr_logic_prime(p):
    '''expr_logic_prime : LOGIC expr
                        | empty'''
```

```
def p_expr_A(p):
    '''expr_A : expr_T expr_A_prime'''

def p_expr_A_prime(p):
    '''expr_A_prime : PLUS expr_A
                    | MINUS expr_A
                    | empty'''

def p_expr_T(p):
    '''expr_T : expr_F expr_T_prime'''

def p_expr_T_prime(p):
    '''expr_T_prime : TIMES expr_T
                    | DIVIDE expr_T
                    | empty'''

def p_expr_F(p):
    '''expr_F : LPAREN expr RPAREN
              | ID
              | NUM'''

def p_empty(p):
    '''empty :'''
    pass

# Tratamento de erros de sintaxe
def p_error(p):
    print("Syntax error in input!")

# Construção do parser
parser = yacc.yacc()
```

Explicações e Ajustes

1. **Start Symbol**: A gramática começa com `start`, que agora define uma sequência de declarações `stmts`.

2. **Declarações e Atribuições**:
   - `dccl` é usado para declarações de variáveis (`int` e `float`).
   - `assign` lida com atribuições, permitindo atribuir expressões a identificadores.

3. **Operadores e Expressões**:
   - A gramática trata operações aritméticas (`+`, `-`, `*`, `/`) e garante a ordem de precedência correta.
   - `expr` é a expressão principal que pode envolver tanto operações aritméticas quanto lógicas.

4. **Estruturas Condicionais e de Repetição**:
   - As regras para `if-else` e `while` são definidas, com suporte para a opção de omitir o `else` (`else_opt`).

5. **Regras de Expressão Lógica**:
   - A lógica booleana é tratada na `expr_logic_prime`, que pode ser uma expressão lógica combinada ou vazia.

6. **Forma LL(1)**:
   - A gramática foi ajustada para evitar recursão à esquerda e garantir que cada produção seja adequada para um parser
    LL(1).

Anotações sobre LL(1)

Para garantir que a gramática seja LL(1), foi evitada recursão à esquerda, e todas as produções foram organizadas de
forma que o primeiro símbolo (ou conjunto de símbolos) seja único para cada produção da mesma regra. Isso permite a
construção de uma tabela de análise que pode ser utilizada por um parser LL(1).

Esta gramática deve atender às necessidades especificadas e ser adequada para a geração de código e análise sintática.

Verificação da propriedade LL1

Para verificar se uma gramática é LL(1), é necessário garantir que ela satisfaça duas condições principais:

1. **Não deve haver recursão à esquerda** em nenhuma de suas produções.
2. **Os conjuntos de "primeiro" e "seguinte"** (FIRST e FOLLOW) para as produções de um mesmo não terminal não devem ter interseções.

Passos para Verificação da Propriedade LL(1)

1. **Eliminação da Recursão à Esquerda**:
   - A gramática não deve ter produções do tipo A→AαA \rightarrow A \alphaA→Aα (recursão à esquerda direta) ou A→Bγ,B→AβA \rightarrow B \gamma, B \rightarrow A \betaA→Bγ,B→Aβ (recursão à esquerda indireta).
2. **Cálculo dos Conjuntos FIRST e FOLLOW**:
   - **Conjunto FIRST**: Para cada produção, determina o conjunto dos primeiros símbolos terminais que podem aparecer na cadeia derivada dessa produção.
   - **Conjunto FOLLOW**: Para cada não terminal, determina o conjunto de terminais que podem aparecer imediatamente após esse não terminal em alguma derivação da gramática.
3. **Condição LL(1)**:
   - Para cada não terminal AAA, para cada par de produções A→αA \rightarrow \alphaA→α e A→βA \rightarrow \betaA→β, os conjuntos FIRST(α)FIRST(\alpha)FIRST(α) e FIRST(β)FIRST(\beta)FIRST(β) devem ser disjuntos. Se α\alphaα ou β\betaβ pode derivar a palavra vazia (ε), então

FIRST(α)∩FOLLOW(A)FIRST(\alpha) \cap FOLLOW(A)FIRST(α)∩FOLLOW(A) e FIRST(β)∩FOLLOW(A)FIRST(\beta) \cap FOLLOW(A)FIRST(β)∩FOLLOW(A) também devem ser disjuntos.

Código Executável para Verificação LL(1)

Para automatizar a verificação da propriedade LL(1), pode-se implementar um analisador que calcula os conjuntos FIRST e FOLLOW e verifica se a gramática é LL(1). Abaixo está um exemplo básico em Python: