| class LLParser:  def \_init\_(self, tokens):  self.tokens = tokens  self.current = 0    def parse(self):  self.E()  if self.current < len(self.tokens):  raise SyntaxError("Unexpected tokens at end of input")    def E(self):  self.T()  while self.current < len(self.tokens) and self.tokens[self.current] == '+':  self.current += 1  self.T()    def T(self):  self.F()  while self.current < len(self.tokens) and self.tokens[self.current] == '\*':  self.current += 1  self.F()    def F(self):  if self.tokens[self.current] == '(':  self.current += 1  self.E()  if self.tokens[self.current] == ')':  self.current += 1  else:  raise SyntaxError("Expected closing parenthesis")  elif self.tokens[self.current].isidentifier():  self.current += 1  else:  raise SyntaxError("Expected identifier or opening parenthesis")    # Example usage:  tokens = ['id', '+', 'id', '\*', 'id']  parser = LLParser(tokens)  parser.parse() |
| --- |

| class LRParser:  def \_\_init\_\_(self, parse\_table, grammar):  self.parse\_table = parse\_table  self.grammar = grammar    def parse(self, input\_tokens):  stack = [0] # Stack with initial state  i = 0    while i < len(input\_tokens):  state = stack[-1]  token = input\_tokens[i]    if (state, token) in self.parse\_table['shift']:  stack.append(self.parse\_table['shift'][(state, token)])  i += 1  elif (state, token) in self.parse\_table['reduce']:  production = self.grammar[self.parse\_table['reduce'][(state, token)]]  for \_ in range(len(production[1])):  stack.pop()  stack.append(self.parse\_table['goto'][(stack[-1], production[0])])  else:  raise ValueError("Syntax error")    return "Accepted" if stack == [0, 'E'] else "Rejected"    # Example parse table and grammar for the input `id + id \* id`  parse\_table = {  'shift': {(0, 'id'): 5, (4, 'id'): 5, (6, 'id'): 5},  'reduce': {(1, '+'): 2, (2, '+'): 4, (3, '+'): 1},  'goto': {(0, 'E'): 1, (0, 'T'): 2, (0, 'F'): 3, (4, 'E'): 6},  }    grammar = {  1: ('E', ['E', '+', 'T']),  2: ('E', ['T']),  3: ('T', ['T', '\*', 'F']),  4: ('T', ['F']),  5: ('F', ['id'])  }    # Instantiate the parser  parser = LRParser(parse\_table, grammar)    # Parse the input tokens  input\_tokens = ['id', '+', 'id', '\*', 'id']  result = parser.parse(input\_tokens)  print(result) # Output: Accepted |
| --- |

| # Define a simplified grammar for arithmetic expressions  productions = {  'E': ['T E\''],  'E\'': ['+ T E\'', 'ε'],  'T': ['F T\''],  'T\'': ['\* F T\'', 'ε'],  'F': ['( E )', 'id']  }    # Define a simple LR parser function  def lr\_parse(tokens):  stack = []  index = 0  while index < len(tokens):  token = tokens[index]  stack.append(token)  index += 1    # Simulate reduction (this example is highly simplified)  if stack[-2:] == ['id', '+']:  stack.pop() # Remove '+'  stack.pop() # Remove 'id'  stack.append('T') # Replace with 'T'  elif stack[-3:] == ['T', 'E\'', '+']:  stack.pop() # Remove '+'  stack.pop() # Remove 'E\''  stack.pop() # Remove 'T'  stack.append('E') # Replace with 'E'    return stack == ['E']    # Example usage  expression = "id + id \* id"  tokens = expression.split() # Simple tokenization  result = lr\_parse(tokens)  print("Valid expression" if result else "Invalid expression") |
| --- |

| int main() {  int a = 5  int b = 10;  return 0;  } |
| --- |

| // Example C-like pseudocode  void parse() {  while (currentToken != EOF) {  if (currentToken == SEMICOLON) {  // Resume parsing after finding the semicolon  continueParsing();  } else {  // Skip the current token  advanceToken();  }  }  } |
| --- |

| int main() {  if (x > 5 {  printf("x is greater than 5");  }  } |
| --- |

| %{  #include <stdio.h>  #include <stdlib.h>  %}    %token NUMBER  %left '+' '-'  %left '\*' '/'    %%    expression:  expression '+' term { printf("Addition\n"); }  | expression '-' term { printf("Subtraction\n"); }  | term  ;    term:  term '\*' factor { printf("Multiplication\n"); }  | term '/' factor { printf("Division\n"); }  | factor  ;    factor:  NUMBER  | '(' expression ')'  ;    %%    int main(void) {  yyparse();  return 0;  }    void yyerror(const char \*s) {  fprintf(stderr, "Error: %s\n", s);  } |
| --- |

| %{  #include <stdio.h>  #include <stdlib.h>  %}    %token NUMBER  %left '+' '-'  %left '\*' '/'    %%    expression:  expression '+' term { printf("Addition\n"); }  | expression '-' term { printf("Subtraction\n"); }  | term  ;    term:  term '\*' factor { printf("Multiplication\n"); }  | term '/' factor { printf("Division\n"); }  | factor  ;    factor:  NUMBER  | '(' expression ')'  ;    %%    int main(void) {  yyparse();  return 0;  }    void yyerror(const char \*s) {  fprintf(stderr, "Error: %s\n", s);  } |
| --- |