| import re  # Define token types using regular expressions token\_specification = [  ('NUMBER', r'\d+'), # Integer  ('ASSIGN', r'='), # Assignment operator  ('ID', r'[A-Za-z]+'), # Identifiers  ('PLUS', r'\+'), # Addition operator  ('MINUS', r'-'), # Subtraction operator  ('SKIP', r'[ \t\n]+'), # Skip over spaces, tabs, and newlines  ('MISMATCH', r'.'), # Any other character (error) ]  # Combine all the regex patterns into one master pattern master\_pattern = '|'.join(f'(?P<{pair[0]}>{pair[1]})' for pair in token\_specification)  # Lexer function to match tokens def lexer(code):  for match in re.finditer(master\_pattern, code):  type\_ = match.lastgroup  value = match.group(type\_)  if type\_ == 'SKIP':  continue # Ignore spaces and newlines  elif type\_ == 'MISMATCH':  raise ValueError(f'Illegal character: {value}')  yield type\_, value  # Test the lexer code = "x = 5 + 3" for token in lexer(code):  print(token) |
| --- |

| import re **#Define some common patterns** patterns = [ (r'[A-Za-z\_][A-Za-z0-9\_]\*', 'identifier'),  (r'\d+(.\d+)?', 'number'),  (r'+|-|=', 'operator'),  ] **#Sample code to test the regex patterns** test\_cases = [  "variable1", # should match identifier  "123.45", # should match number  "+", # should match operator  ] **#Test function** def test\_regex():  for pattern, label in patterns:  print(f"Testing {label} pattern: {pattern}")  for case in test\_cases:  if re.match(pattern, case):  print(f"Match found for '{case}'")  else:  print(f"No match for '{case}'")  test\_regex() |
| --- |

| Import re  # Define regex patterns for the tokens TOKEN\_PATTERNS = [  ('NUMBER', r'\d+'),  ('PLUS', r'\+'),  ('MINUS', r'-'),  ('TIMES', r'\\*'),  ('DIVIDE', r'/'),  ('LPAREN', r'\('),  ('RPAREN', r'\)'),  ('ID', r'[a-zA-Z\_][a-zA-Z0-9\_]\*'),  ('WHITESPACE', r'\s+'), ]  def lexer(code):  tokens = []  position = 0  while position < len(code):  match = None  for token\_type, pattern in TOKEN\_PATTERNS:  regex = re.compile(pattern)  match = regex.match(code, position)  if match:  if token\_type != 'WHITESPACE': # Ignore whitespaces  tokens.append((token\_type, match.group(0)))  position = match.end()  break  if not match:  raise SyntaxError(f"Unexpected character at position {position}")  return tokens |
| --- |

| class ASTNode:  def \_\_init\_\_(self, value, children=None):  self.value = value  self.children = children if children else []  def parse\_expression(tokens):  token = tokens.pop(0)  if token[0] == 'NUMBER':  return ASTNode('NUMBER', [token[1]])  elif token[0] == 'ID':  return ASTNode('ID', [token[1]])  else:  raise SyntaxError(f"Unexpected token: {token}") |
| --- |

| def evaluate\_ast(node):  if node.value == 'NUMBER':  return int(node.children[0]) # Return the numeric value  elif node.value == 'ID':  return variables[node.children[0]] # Retrieve value from variables  else:  raise RuntimeError(f"Unknown AST node type: {node.value}") |
| --- |

| variables = {}  def parse\_assignment(tokens):  var\_name = tokens.pop(0)[1] # Get variable name  tokens.pop(0) # Skip '='  expr\_node = parse\_expression(tokens)  variables[var\_name] = evaluate\_ast(expr\_node) |
| --- |

| def interpret(code):  tokens = lexer(code)  while tokens:  if tokens[0][0] == 'ID': # If we encounter an identifier, it's an assignment  parse\_assignment(tokens)  else:  expr\_node = parse\_expression(tokens)  result = evaluate\_ast(expr\_node)  print(result) |
| --- |

| import re   TOKEN\_PATTERNS = [   ('SET', r'SET'),   ('PRINT', r'PRINT'),   ('NUMBER', r'\d+'),   ('ID', r'[a-zA-Z\_][a-zA-Z0-9\_]\*'),   ('EQUALS', r'='),   ('PLUS', r'\+'),   ('MINUS', r'-'),   ('TIMES', r'\\*'),   ('DIVIDE', r'/'),   ('WHITESPACE', r'\s+'),  ]  def lexer(code):   tokens = []   position = 0   while position < len(code):   match = None   for token\_type, pattern in TOKEN\_PATTERNS:   regex = re.compile(pattern)   match = regex.match(code, position)   if match:   if token\_type != 'WHITESPACE':   tokens.append((token\_type, match.group(0)))   position = match.end()   break   if not match:   raise SyntaxError(f"Unexpected character at position {position}")   return tokens |
| --- |

### 

| class ASTNode:  def **init**(self, value, children=None):  self.value = value  self.children = children if children else []  def parse(tokens):  token = tokens.pop(0)  if token[0] == 'SET':  var\_name = tokens.pop(0)[1]  tokens.pop(0) # Skip '='  expr\_node = parse\_expression(tokens)  return ('SET', var\_name, expr\_node)  elif token[0] == 'PRINT':  expr\_node = parse\_expression(tokens)  return ('PRINT', expr\_node)  else:  raise SyntaxError(f"Unexpected token: {token}")  def parse\_expression(tokens):  token = tokens.pop(0)  if token[0] == 'NUMBER':  return ASTNode('NUMBER', [token[1]])  elif token[0] == 'ID':  return ASTNode('ID', [token[1]])  else:  raise SyntaxError(f"Unexpected token in expression: {token}") |
| --- |

### 

| variables = {}    def evaluate(node):  if node[0] == 'SET':  variables[node[1]] = evaluate(node[2])  elif node[0] == 'PRINT':  print(evaluate(node[1]))  elif isinstance(node, ASTNode):  if node.value == 'NUMBER':  return int(node.children[0])  elif node.value == 'ID':  return variables.get(node.children[0], 0)  else:  raise RuntimeError(f"Unknown node type: {node}") |
| --- |

| def run(code):  tokens = lexer(code)  while tokens:  ast = parse(tokens)  evaluate(ast)    dsl\_code = """  SET x = 10  SET y = 5  PRINT x + y  """    run(dsl\_code) |
| --- |