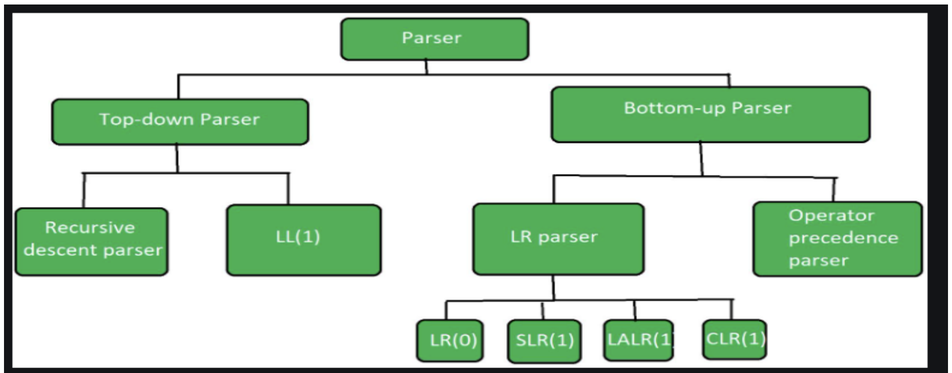
Tokenizer – takes source code as input and breaks it down into tokens, tokens represents a single unit of language. Tokenizer should identify all valid tokens, handle whitespace, and errors

[Building a compiler – working on the tokenizer - DEV Community]()

Has an error in the website*

```
def tokenize(source_code):
  tokens = []
  current_position = 0

  while current_position < len(source_code):
    current_character = source_code[current_position]

    # Ignore whitespace.
    while current_character.isspace():
      current_position += 1
      current_character = source_code[current_position]

    # If the current character is a digit, return a number token.
    if current_character.isdigit():
      number = ""
      while current_character.isdigit():
        number += current_character
        current_position += 1
        current_character = source_code[current_position]
      tokens.append(Token("NUMBER", int(number)))

    # If the current character is an alphabetic character, return an identifier token.
    elif current_character.isalpha():
      identifier = ""
      while current_character.isalpha():
        identifier += current_character
        current_position += 1
        current_character = source_code[current_position]
      tokens.append(Token("IDENTIFIER", identifier))

    # If the current character is a known operator, return an operator token.
    elif current_character in ["+", "-", "*", "/"]:
      operator = current_character
      current_position += 1
      tokens.append(Token("OPERATOR", operator))

    # Otherwise, return an unknown token.
    else:
      tokens.append(Token("UNKNOWN", current_character))
      current_position += 1

  return tokens
```

Parser – takes tokens from tokenizer and builds AST. Parsers should analyze into parts the valid programs, handle errors and generate the AST that is accurate. There are different types of parsers but are classified by two main categories. Top-down parser generates parse for the given input, there are two types of top down parsers. Recursive descent parser generates the parse tree by using brute force and backtracking. Non-recursive descent parser it uses a parsing table to generate the parse tree instead of back tracking. Bottom up parser generates the parse tree for the given input string with the help of grammar by compressing the terminals. LR Parser that generates the parse tree with given tree by using unambigiuos grammar, has four types: LR(0),SLR(1),LALR(1),CLR(1). Operator Precedence parser generates the tree from given grammar and string but by not having terminals and epsilon on the right side of the tree.



Classification of Parser

```
def parse(tokens):
  ast = Node()
  current_token = tokens[0]

  while current_token is not None:
    if current_token.type == "NUMBER":
      ast.children.append(NumberNode(current_token.value))
    elif current_token.type == "IDENTIFIER":
      ast.children.append(IdentifierNode(current_token.value))
    elif current_token.type == "OPERATOR":
      ast.children.append(OperatorNode(current_token.value))

    current_token = tokens[1:]

  return ast
```

TypeChecker – helps prevents errors, it analyzes the AST and checking the types of the expressions. The typechecker should be able to check the types of the expressions, report errors, and handle user defined types. There are two kinds of type checking. First one is Static Type Checking Is defined as type checking performed at compile time. Example of static checks, a compiler should report an error, must have a way to transfer the flow of control, defining certain objects once and see if two names are related. Dynamic Type Checking is being down at run time.

```
def typecheck(ast):
    for node in ast.children:
        if node.type == "BinaryExpressionNode":
            # Check the types of the left and right operands.
            left_operand_type = typecheck(node.left_operand)
            right_operand_type = typecheck(node.right_operand)

            # Make sure that the types of the operands are compatible.
            if left_operand_type != right_operand_type:
                raise TypeError("Type mismatch")

        elif node.type == "IdentifierNode":
            # Check the type of the identifier.
            identifier_type = get_type(node.name)

            # Make sure that the type of the identifier is valid.
            if identifier_type is None:
                raise UndeclaredIdentifierError(node.name)

        elif node.type == "NumberNode":
            # Check the type of the number.
            number_type = int

            # Make sure that the type of the number is valid.
            if not isinstance(node.value, int):
                raise TypeError("Invalid type for number")
```

CodeGenerator – generates machine code from the AST, the code is specific to the platform. The generator should be able to create machine code efficiently and handle users functions and data structures

```
def generate_code(ast):
  machine_code = []

  for node in ast.children:
    if node.type == "BinaryExpressionNode":
      # Generate machine code for the left and right operands.
      left_operand_code = generate_code(node.left_operand)
      right_operand_code = generate_code(node.right_operand)

      # Generate machine code for the binary operation.
      if node.operator == "+":
        machine_code.append("add")
      elif node.operator == "-":
        machine_code.append("sub")
      elif node.operator == "*":
        machine_code.append("mul")
      elif node.operator == "/":
        machine_code.append("div")

    elif node.type == "IdentifierNode":
      # Generate machine code to load the value of the identifier into a register.
      machine_code.append("load")

    elif node.type == "NumberNode":
      # Generate machine code to load the value of the number into a register.
```

```
        machine_code.append("load")

    return machine_code
```