📚 Detailed Role Explanation - Mini-Python Compiler Visualizer

Complete Breakdown of Work Done in Each Role

This document provides an in-depth explanation of the work done in each of the four roles, including specific code implementations, technical decisions, and step-by-step explanations.

🎨 Role 1: Frontend Developer & UI/UX Designer

What I Did: Building the Interactive Web Interface

I created a complete Streamlit web application that serves as the visual interface for the compiler. This involved designing the user experience, implementing interactive components, and integrating all backend compiler phases into a cohesive visualization platform.

---

1. Application Architecture & Setup

Page Configuration (app.py lines 8-13)

```
st.set_page_config(
    page_title="Mini-Python Compiler",
    page_icon="🔮",
    layout="wide",
    initial_sidebar_state="expanded"
)
```

What This Does:
- Sets browser tab title and icon
- Uses wide layout for better visualization space
- Keeps sidebar expanded by default for easy access to controls

Why This Matters:
- Professional appearance in browser
- Maximizes screen real estate for code and visualizations
- Improves user experience with immediate access to examples

Custom CSS Styling (app.py lines 16-28)

```
st.markdown("""
<style>
    .stTextArea textarea {
        font-family: 'Consolas', 'Monaco', 'Courier New', monospace;
    }
    .reportview-container {
        background: #0e1117;
    }
    .sidebar .sidebar-content {
        background: #262730;
    }
</style>
""", unsafe_allow_html=True)
```

What This Does:
- Applies monospace font to code editor for better readability
- Sets dark theme colors for professional look
- Customizes sidebar appearance

Technical Decision:
- Used monospace fonts because they align characters vertically, making code easier to read
- Dark theme reduces eye strain during extended use
- Matches modern IDE aesthetics

---

2. Two-Column Layout Design

Code Editor Column (app.py lines 137-144)

```
col1, col2 = st.columns([1, 1])

with col1:
    st.subheader("📝 Source Code Editor")
    initial_code = examples[selected_example] if selected_example != "Select an Example" else ""
    code = st.text_area("Input Code", value=initial_code, height=400,
                        placeholder="Type your python code here...",
```

```
                          help="Support for: variables, loops, functions, recursion, and try-except.")
    analyze_btn = st.button("🚀 Run / Analyze Pipeline", type="primary", use_container_width=True)
```

What This Does:
- Creates left column for code input
- Provides 400px height text area for comfortable coding
- Shows helpful placeholder text
- Adds tooltip explaining supported features
- Creates prominent "Run" button

Implementation Details:
- `value=initial_code` pre-fills editor when example is selected
- `type="primary"` makes button stand out visually
- `use_container_width=True` makes button span full column width

---

3. Example Loader System

Pre-built Examples (app.py lines 108-115)

```
examples = {
    "Select an Example": "",
    "Hello World": 'print("Hello Mini-Python!")',
    "Variables & Math": 'x = 10\ny = 5\nprint(x + y * 2)',
    "Conditionals": 'x = 42\nif x > 50:\n    print("Big")\nelse:\n    print("Small")',
    "Loops": 'print("Counting:")\nfor i in range(5):\n    print(i)',
    "Functions": 'def greet(name):\n    print("Hello " + name)\n\ngreet("User")'
}

selected_example = st.selectbox("Load Example", list(examples.keys()))
```

What This Does:
- Provides 5 ready-to-run code examples
- Uses dropdown for easy selection
- Automatically loads selected code into editor

Why This Approach:
- New users can immediately see the compiler in action
- Examples demonstrate different language features
- Reduces barrier to entry for learning

---

4. Tabbed Visualization System

Five-Phase Pipeline Tabs (app.py lines 156-158)

```
tab1, tab2, tab3, tab4, tab5 = st.tabs([
    "📊 Lexer", "🌳 Parser (AST)", "🌓 Semantic", "⚙️ ICG", "🏁 Output"
])
```

What This Does:
- Creates 5 separate tabs for each compiler phase
- Uses emojis for visual appeal and quick recognition
- Organizes complex information into digestible sections

Tab 1: Lexer - Token Display (app.py lines 161-171)

```
with tab1:
    if results.get('lexer_error'):
        st.error(f"Lexical Error: {results['lexer_error']}")
    elif results.get('tokens'):
        tokens = results['tokens']
        df = pd.DataFrame(tokens)
        st.dataframe(df, use_container_width=True)

        # Download Tokens CSV
        csv = df.to_csv(index=False).encode('utf-8')
        st.download_button("📥 Download Tokens (CSV)", csv, "tokens.csv", "text/csv")
```

What This Does:
- Displays tokens in a structured table using Pandas DataFrame
- Shows error messages if tokenization fails
- Provides CSV export functionality

Technical Implementation:
- `pd.DataFrame(tokens)` converts token list to table format
- `use_container_width=True` makes table responsive
- `to_csv().encode('utf-8')` prepares data for download
- Download button allows saving results for analysis

Tab 2: Parser - AST Visualization (app.py lines 174-186)

```
with tab2:
    if results.get('parser_error'):
        st.error(f"Syntax Error: {results['parser_error']}")
    elif results.get('ast'):
        st.success("Syntax Analysis Complete")
        visualizer = ASTVisualizer()
        try:
            visualizer.visualize(results['ast'])
            st.graphviz_chart(visualizer.dot)
            st.download_button("📥 Download AST (DOT)", visualizer.dot.source, "ast.dot", "text/plain")
        except Exception as e:
            st.error(f"Visualization Error: {e}")
```

What This Does:

- Renders Abstract Syntax Tree using Graphviz
- Shows success message when parsing succeeds
- Handles visualization errors gracefully
- Provides DOT file export

Technical Decisions:

- Used try-except to prevent crashes if Graphviz isn't installed
- `st.graphviz_chart()` renders interactive tree diagram
- DOT format export allows opening in external tools

Tab 5: Execution Output (app.py lines 208-217)

```
with tab5:
    if results.get('exec_error'):
        st.error(f"Execution Error: {results['exec_error']}")

    output = results.get('exec_output', '')
    if output:
        st.markdown("#### Program Output")
        st.code(output, language="text")
    elif not results.get('exec_error'):
         st.info("Program finished with no output.")
```

What This Does:

- Displays program execution results
- Shows errors in red if execution fails
- Handles empty output gracefully
- Uses code block for formatted output

User Experience Design:
- Different colors for errors (red), info (blue), and output (gray)
- Clear messaging when program runs but produces no output
- Formatted code blocks preserve whitespace and formatting

---

5. Educational Tooltips

Compiler Theory Explanations (app.py lines 126-134)

```
st.markdown("### 📚 Compiler 101")
with st.expander("What is a Lexer?"):
    st.write("The Lexical Analyzer (Lexer) breaks the source code into a stream of **tokens**...")
with st.expander("What is a Parser?"):
    st.write("The Parser takes tokens and arranges them into an **Abstract Syntax Tree (AST)**...")
with st.expander("Semantic Analysis?"):
    st.write("This phase ensures that the code makes sense...")
with st.expander("What is ICG?"):
    st.write("Intermediate Code Generation (ICG) creates a platform-independent representation...")
```

What This Does:
- Provides collapsible educational content
- Explains each compiler phase in simple terms
- Helps users understand what they're seeing

Educational Value:
- Makes the tool useful for learning compiler theory
- Reduces confusion about technical terms
- Encourages exploration and understanding

---

6. Pipeline Execution Logic

Compiler Pipeline Runner (app.py lines 42-98)

```
def run_compiler_pipeline(code):
    results = {}

    # 1. Lexical Analysis
    try:
        tokens = tokenize(code)
        results['tokens'] = tokens
        results['lexer_error'] = None
```

```
    except Exception as e:
        results['tokens'] = []
        results['lexer_error'] = str(e)
        return results  # Stop if lexer fails

    # 2. Parsing (Syntax & AST)
    try:
        ast = parser.parse(code)
        results['ast'] = ast
        results['parser_error'] = None
    except Exception as e:
        results['ast'] = None
        results['parser_error'] = str(e)
        return results  # Stop if parser fails

    # ... continues for semantic, ICG, and execution
```

What This Does:
- Orchestrates all compiler phases in sequence
- Catches errors at each stage
- Stops pipeline if early stage fails
- Returns comprehensive results dictionary

Error Handling Strategy:
- Each phase wrapped in try-except
- Errors stored in results dictionary
- Pipeline stops at first failure (cascading prevention)
- Allows UI to show exactly where compilation failed

Key Achievements in Frontend Role:

☑ **Created Professional UI** - Modern, dark-themed interface matching IDE aesthetics
☑ **Implemented 5-Phase Visualization** - Clear separation of compiler stages
☑ **Built Example System** - 5 pre-loaded examples for immediate testing
☑ **Added Export Features** - CSV, DOT, and TXT downloads for all outputs
☑ **Integrated Error Handling** - Graceful error display at each stage
☑ **Educational Content** - Built-in compiler theory explanations
☑ **Responsive Design** - Two-column layout optimized for wide screens

---

🔤 Role 2: Lexical Analyzer Developer

What I Did: Building the Tokenizer

I implemented the lexical analysis phase using PLY (Python Lex-Yacc), which breaks source code into meaningful tokens. This is the first and foundational step of the compilation process.

---

1. Token Definition

Complete Token Set (src/lexer.py lines 5-48)

```
tokens = (
    'NUMBER',      # 123, 456
    'STRING',      # "hello", 'world'
    'IDENTIFIER',  # variable_name, function_name
    'PLUS',        # +
    'MINUS',       # -
    'TIMES',       # *
    'DIVIDE',      # /
    'MODULO',      # %
    'EQUALS',      # =
    'LPAREN',      # (
    'RPAREN',      # )
    'LBRACKET',    # [
    'RBRACKET',    # ]
    'COLON',       # :
    'COMMA',       # ,
    'DOT',         # .
    'GT',          # >
    'LT',          # <
    'GE',          # >=
    'LE',          # <=
    'EQ',          # ==
    'NE',          # !=
    'AND',         # and
    'OR',          # or
    'NOT',         # not
    # ... keywords
)
```

What This Does:
- Defines all possible token types the lexer can recognize
- Covers operators, delimiters, literals, and keywords
- Total of 47 different token types

Design Decision:
- Separated operators by function (arithmetic, comparison, logical)
- Included both single-char (`+`) and multi-char (`>=`) tokens
- Distinguished between `=` (assignment) and `==` (equality)

---

2. Regular Expression Rules

Simple Token Patterns (src/lexer.py lines 51-69)

```
t_PLUS = r'\+'
t_MINUS = r'-'
t_TIMES = r'\*'
t_DIVIDE = r'/'
t_MODULO = r'%'
t_EQUALS = r'='
t_GE = r'>='
t_LE = r'<='
t_EQ = r'=='
t_NE = r'!='
```

What This Does:
- Maps token names to regex patterns
- `t_` prefix tells PLY this is a token rule
- Backslashes escape special regex characters

Technical Details:
- `\+` escapes `+` because it means "one or more" in regex
- `\*` escapes `*` because it means "zero or more" in regex
- Multi-character operators (`>=` , `==`) must be defined before single-character ones to avoid conflicts

---

Number Recognition (src/lexer.py lines 112-115)

```
def t_NUMBER(t):
    r'\d+'
    t.value = int(t.value)
    return t
```

What This Does:
- Matches one or more digits (`\d+`)
- Converts string to integer
- Returns token with numeric value

Example:
- Input: `"123"`
- Regex matches: `"123"`
- Converted to: `123` (integer)
- Token: `{'type': 'NUMBER', 'value': 123}`

String Literal Parsing (src/lexer.py lines 117-121)

```
def t_STRING(t):
    r'"[^"\\]*(\\.[^"\\]*)*"|\'[^\'\\]*(\\.[^\'\\]*)*\''
    # Remove quotes and handle escape sequences
    t.value = t.value[1:-1].encode().decode('unicode_escape')
    return t
```

What This Does:
- Matches strings in double quotes (`"..."`) or single quotes (`'...'`)
- Handles escape sequences like `\n` , `\t` , `\\`
- Removes surrounding quotes
- Processes escape characters

Regex Breakdown:
- `"[^"\\]*` - Match opening quote, then any non-quote, non-backslash chars
- `(\\.[^"\\]*)*` - Match backslash + any char (escape sequence), repeat
- Same pattern for single quotes

Example:
- Input: `"Hello\nWorld"`
- After processing: `"Hello` + newline + `World"`
- Escape sequence `\n` converted to actual newline character

Identifier & Keyword Recognition (src/lexer.py lines 72-92, 123-126)

```
reserved = {
    'True': 'TRUE',
    'False': 'FALSE',
    'if': 'IF',
    'else': 'ELSE',
    'while': 'WHILE',
    'for': 'FOR',
    'def': 'DEF',
    'return': 'RETURN',
    # ... more keywords
}

def t_IDENTIFIER(t):
    r'[a-zA-Z_][a-zA-Z_0-9]*'
    t.type = reserved.get(t.value, 'IDENTIFIER')
    return t
```

What This Does:

- Matches valid Python identifiers (letters, underscores, numbers)
- Checks if identifier is a reserved keyword
- Returns appropriate token type

How It Works:

1. Regex matches: `variable_name`, `if`, `for`, `my_var_123`
2. Looks up value in `reserved` dictionary
3. If found → returns keyword token (e.g., `IF`)
4. If not found → returns `IDENTIFIER` token

Example:

- Input: `if` → Token type: `IF` (keyword)
- Input: `count` → Token type: `IDENTIFIER` (variable)

---

3. Special Handling

Newline Tracking (src/lexer.py lines 97-100)

```
def t_NEWLINE(t):
    r'\n+'
    t.lexer.lineno += t.value.count('\n')
    return t
```

What This Does:
- Matches one or more newline characters
- Updates line number counter
- Returns newline token

Why This Matters:
- Enables accurate error reporting with line numbers
- Python is whitespace-sensitive
- Helps track code structure

Comment Handling (src/lexer.py lines 132-135)

```
def t_COMMENT(t):
    r'\#.*'
    # No return value. Token discarded
    pass
```

What This Does:
- Matches `#` followed by any characters until end of line
- Doesn't return a token (comments are ignored)

Example:
- Input: `x = 5 # This is a comment`
- Tokens generated: `IDENTIFIER(x)`, `EQUALS`, `NUMBER(5)`
- Comment is silently discarded

Error Handling (src/lexer.py lines 138-140)

```
def t_error(t):
    # Instead of raising an error, we'll skip the character and continue
    t.lexer.skip(1)
```

What This Does:
- Called when lexer encounters unrecognized character
- Skips the character and continues
- Prevents crashes on invalid input

Design Choice:
- Graceful degradation instead of hard failure

- Allows partial tokenization of code with errors
- Better user experience in interactive environment

---

4. Tokenization Function

Main Tokenizer (src/lexer.py lines 145-164)

```
def tokenize(code):
    """Tokenize the input code and return a list of tokens with their details."""
    lexer.lineno = 1
    lexer.input(code)
    tokens = []

    while True:
        tok = lexer.token()
        if not tok:
            break
        token_info = {
            "type": tok.type,
            "value": str(tok.value),
            "line": tok.lineno,
            "lexpos": tok.lexpos
        }
        tokens.append(token_info)

    return tokens
```

What This Does:
- Resets lexer state
- Feeds code into lexer
- Extracts all tokens in a loop
- Returns list of token dictionaries

Token Information Captured:
- `type` - Token category (NUMBER, IDENTIFIER, etc.)
- `value` - Actual text or converted value
- `line` - Line number in source code
- `lexpos` - Character position in source

Example Output:

```
Input: "x = 5"
Output: [
    {'type': 'IDENTIFIER', 'value': 'x', 'line': 1, 'lexpos': 0},
    {'type': 'EQUALS', 'value': '=', 'line': 1, 'lexpos': 2},
```

```
    {'type': 'NUMBER', 'value': '5', 'line': 1, 'lexpos': 4}
]
```

---

5. Token Formatting for Display

Formatted Output (src/lexer.py lines 166-225)

```
def format_token_output(tokens):
    """Format the tokens into a structured, readable output."""
    # Create formatted table
    max_type_len = max(len(str(token['type'])) for token in tokens) + 2
    max_value_len = max(len(str(token['value'])) for token in tokens) + 2

    header = f"{'Token Type'.ljust(max_type_len)}{'Token Value'.ljust(max_value_len)}..."
    # ... creates aligned table

    # Add token summary
    token_types = {}
    for token in tokens:
        token_types[token['type']] = token_types.get(token['type'], 0) + 1

    # Categorize tokens
    categories = {
        "Keywords": ["IF", "ELSE", "WHILE", "FOR", ...],
        "Operators": ["PLUS", "MINUS", "TIMES", ...],
        "Delimiters": ["LPAREN", "RPAREN", ...],
        "Literals": ["NUMBER", "STRING"],
        "Identifiers": ["IDENTIFIER"]
    }
```

What This Does:

- Creates aligned table output
- Calculates column widths dynamically
- Adds summary statistics
- Categorizes tokens by type

Output Example:

```
Token Type      Token Value     Line  Position
------------------------------------------------
IDENTIFIER      x               1     0
EQUALS          =               1     2
NUMBER          5               1     4

Token Summary:
- IDENTIFIER: 1 tokens
- EQUALS: 1 tokens
```

```
- NUMBER: 1 tokens

Token Categories:
- Identifiers: 1 tokens
- Operators: 1 tokens
- Literals: 1 tokens
```

Key Achievements in Lexer Role:

- ☑ **Defined 47 Token Types** - Complete coverage of Python subset
- ☑ **Implemented Regex Patterns** - Accurate matching for all constructs
- ☑ **String Parsing with Escapes** - Handles `\n`, `\t`, `\\`, etc.
- ☑ **Keyword Recognition** - Distinguishes reserved words from identifiers
- ☑ **Line Number Tracking** - Enables precise error reporting
- ☑ **Comment Handling** - Properly ignores Python comments
- ☑ **Error Recovery** - Graceful handling of invalid characters
- ☑ **Formatted Output** - Beautiful table display with statistics

---

🌳 Role 3: Parser & Semantic Analyzer Developer

What I Did: Building the Syntax Analyzer and Type Checker

I implemented the parsing phase using PLY's yacc module to build Abstract Syntax Trees, defined all AST node classes, performed semantic analysis for type checking and scope validation, and created AST visualization tools.

---

1. AST Node Definitions

Basic Expression Nodes (src/ast_nodes.py lines 4-18)

```
class Number:
    def __init__(self, value):
        self.value = value

class String:
    def __init__(self, value):
        self.value = value
```

```
class Boolean:
    def __init__(self, value):
        self.value = value

class Identifier:
    def __init__(self, name):
        self.name = name
```

What This Does:
- Defines nodes for literal values and variables
- Each node stores its value or name
- Forms the leaf nodes of the AST

Example:
- Code: `42` → AST: `Number(value=42)`
- Code: `"hello"` → AST: `String(value="hello")`
- Code: `x` → AST: `Identifier(name="x")`

Assignment Node (src/ast_nodes.py lines 20-23)

```
class Assign:
    def __init__(self, name, expr):
        self.name = name
        self.expr = expr
```

What This Does:
- Represents variable assignment
- Stores variable name and expression being assigned

Example:
- Code: `x = 5 + 3`
- AST: `Assign(name="x", expr=BinaryOp(left=Number(5), op="+", right=Number(3)))`

Binary Operation Node (src/ast_nodes.py lines 93-97)

```
class BinaryOp:
    def __init__(self, left, op, right):
        self.left = left
```

```
        self.op = op
        self.right = right
```

What This Does:
- Represents operations with two operands
- Stores left operand, operator, and right operand
- Handles arithmetic, comparison, and logical operations

Example:
- Code: `5 + 3`
- AST: `BinaryOp(left=Number(5), op="+", right=Number(3))`

Tree Structure:

```
    BinaryOp(+)
    /         \
Number(5)   Number(3)
```

Control Flow Nodes (src/ast_nodes.py lines 104-119)

```
class IfElse:
    def __init__(self, condition, if_body, else_body=None):
        self.condition = condition
        self.if_body = if_body
        self.else_body = else_body

class WhileLoop:
    def __init__(self, condition, body):
        self.condition = condition
        self.body = body

class ForLoop:
    def __init__(self, var, iterable, body):
        self.var = var
        self.iterable = iterable
        self.body = body
```

What This Does:
- Represents conditional and loop structures
- Stores conditions and code blocks
- Enables control flow analysis

Example - If Statement:

```
Code:
if x > 10:
    print("Big")
else:
    print("Small")

AST:
IfElse(
    condition=BinaryOp(left=Identifier("x"), op=">", right=Number(10)),
    if_body=[Print(String("Big"))],
    else_body=[Print(String("Small"))]
)
```

Function Nodes (src/ast_nodes.py lines 121-130)

```
class FunctionDef:
    def __init__(self, name, params, body):
        self.name = name
        self.params = params
        self.body = body

class FunctionCall:
    def __init__(self, name, args):
        self.name = name
        self.args = args
```

What This Does:

- Represents function definitions and calls
- Stores function name, parameters, and body
- Enables function analysis and execution

Example:

```
Code:
def add(a, b):
    return a + b

result = add(5, 3)

AST:
FunctionDef(
    name="add",
    params=["a", "b"],
    body=[Return(BinaryOp(Identifier("a"), "+", Identifier("b")))]
)
FunctionCall(name="add", args=[Number(5), Number(3)])
```

Data Structure Nodes (src/ast_nodes.py lines 29-36)

```
class ListNode:
    def __init__(self, elements):
        self.elements = elements

class IndexNode:
    def __init__(self, expr, index):
        self.expr = expr
        self.index = index
```

What This Does:
- Represents lists and indexing operations
- Stores list elements or index expressions

Example:

```
Code: numbers = [1, 2, 3]
AST: Assign(
    name="numbers",
    expr=ListNode(elements=[Number(1), Number(2), Number(3)])
)

Code: numbers[0]
AST: IndexNode(expr=Identifier("numbers"), index=Number(0))
```

---

2. Grammar Rules (Parser Implementation)

The parser uses context-free grammar rules to build the AST. Here's how key constructs are parsed:

Assignment Statement

```
def p_statement_assign(p):
    '''statement : IDENTIFIER EQUALS expression'''
    p[0] = Assign(p[1], p[3])
```

What This Does:
- Matches pattern: `identifier = expression`
- Creates `Assign` node with variable name and expression
- `p[1]` is identifier, `p[3]` is expression (p[2] is `=`)

Binary Operations with Precedence

```
def p_expression_binop(p):
    '''expression : expression PLUS expression
                  | expression MINUS expression
                  | expression TIMES expression
                  | expression DIVIDE expression'''
    p[0] = BinaryOp(p[1], p[2], p[3])
```

What This Does:

- Matches arithmetic operations
- Creates `BinaryOp` node
- PLY handles operator precedence automatically

Precedence Rules:

```
precedence = (
    ('left', 'OR'),
    ('left', 'AND'),
    ('left', 'EQ', 'NE'),
    ('left', 'LT', 'LE', 'GT', 'GE'),
    ('left', 'PLUS', 'MINUS'),
    ('left', 'TIMES', 'DIVIDE', 'MODULO'),
    ('right', 'NOT'),
)
```

Effect:

- `2 + 3 * 4` parsed as `2 + (3 * 4)` not `(2 + 3) * 4`
- Multiplication has higher precedence than addition

If-Else Statement

```
def p_statement_if(p):
    '''statement : IF expression COLON suite
                 | IF expression COLON suite ELSE COLON suite'''
    if len(p) == 5:
        p[0] = IfElse(p[2], p[4], None)
    else:
        p[0] = IfElse(p[2], p[4], p[7])
```

What This Does:

- Handles both `if` and `if-else` forms

- `len(p) == 5` means no else clause
- `len(p) == 8` means else clause present
- Creates `IfElse` node with condition and bodies

Function Definition

```
def p_statement_funcdef(p):
    '''statement : DEF IDENTIFIER LPAREN params RPAREN COLON suite'''
    p[0] = FunctionDef(p[2], p[4], p[7])
```

What This Does:
- Matches `def name(params): body`
- Extracts function name, parameter list, and body
- Creates `FunctionDef` node

---

3. Semantic Analysis

Symbol Table Management

```
class SymbolTable:
    def __init__(self):
        self.scopes = [{}]  # Stack of scopes

    def enter_scope(self):
        self.scopes.append({})

    def exit_scope(self):
        self.scopes.pop()

    def define(self, name, type_info):
        self.scopes[-1][name] = type_info

    def lookup(self, name):
        for scope in reversed(self.scopes):
            if name in scope:
                return scope[name]
        return None
```

What This Does:
- Tracks variable declarations across scopes

- Implements scope stack (global, function, block scopes)
- Enables variable lookup and redefinition detection

Example:

```
x = 5          # Global scope
def foo():
    y = 10     # Function scope
    x = 20     # Shadows global x
```

Symbol table after parsing:

```
Scope 0 (global): {x: int}
Scope 1 (foo):    {y: int, x: int}
```

Type Checking

```
def check_binary_op(left_type, op, right_type):
    if op in ['+', '-', '*', '/', '%']:
        if left_type == 'number' and right_type == 'number':
            return 'number'
        elif left_type == 'string' and right_type == 'string' and op == '+':
            return 'string'
        else:
            raise TypeError(f"Cannot apply {op} to {left_type} and {right_type}")
```

What This Does:
- Validates type compatibility in operations
- Allows `number + number` → `number`
- Allows `string + string` → `string`
- Rejects `number + string`

---

4. AST Visualization

Graphviz Tree Generation (src/utils.py)

```
class ASTVisualizer:
    def __init__(self):
        self.dot = Digraph()
```

```
        self.counter = 0

    def visualize(self, node):
        return self._visit(node)

    def _visit(self, node):
        node_id = f"node{self.counter}"
        self.counter += 1

        if isinstance(node, Number):
            self.dot.node(node_id, f"Number\\n{node.value}")
        elif isinstance(node, BinaryOp):
            self.dot.node(node_id, f"BinaryOp\\n{node.op}")
            left_id = self._visit(node.left)
            right_id = self._visit(node.right)
            self.dot.edge(node_id, left_id)
            self.dot.edge(node_id, right_id)
        # ... more node types

        return node_id
```

What This Does:
- Traverses AST recursively
- Creates Graphviz nodes for each AST node
- Draws edges between parent and child nodes
- Generates visual tree diagram

Example Output:

```
Code: x = 5 + 3

Visual Tree:
      Assign
      /    \
  "x"      BinaryOp(+)
           /         \
       Number(5)   Number(3)
```

Key Achievements in Parser Role:

- ☑ **Defined 20+ AST Node Classes** - Complete language representation
- ☑ **Implemented Grammar Rules** - 30+ production rules for Python subset
- ☑ **Built Symbol Table** - Nested scope support with stack-based design
- ☑ **Type Checking System** - Validates type compatibility in operations
- ☑ **Scope Resolution** - Ensures variables declared before use
- ☑ **AST Visualization** - Interactive Graphviz tree diagrams

☑ **Error Recovery** - Graceful handling of syntax errors
☑ **Function Support** - Full function definition and call analysis

---

⚙ Role 4: Code Generator & Interpreter Developer

What I Did: Building the Execution Engine

I implemented intermediate code generation (Three-Address Code) and built a tree-walking interpreter that directly executes the AST, handling variables, functions, recursion, loops, and exception handling.

---

1. Intermediate Code Generation (ICG)

Three-Address Code Concept

Three-Address Code (TAC) is an intermediate representation where each instruction has at most three operands:

```
result = operand1 operator operand2
```

Example:

```
Code: x = 5 + 3 * 2

TAC:
t1 = 3
t2 = 2
t3 = t1 * t2
t4 = 5
t5 = t4 + t3
x = t5
```

Temporary Variable Management (src/icg_generator.py lines 8-14)

```
temp_counter = [0]
label_counter = [0]
```

```
def new_temp():
    temp_counter[0] += 1
    return f"t{temp_counter[0]}"

def new_label():
    label_counter[0] += 1
    return f"L{label_counter[0]}"
```

What This Does:

- Generates unique temporary variable names (`t1` , `t2` , `t3` , ...)
- Generates unique labels for control flow (`L1` , `L2` , `L3` , ...)
- Uses list to maintain counter across function calls

Why Lists?

- Python closures can't modify primitive values
- Lists are mutable, so `temp_counter[0] += 1` works
- Alternative would be `nonlocal` keyword

Expression Code Generation (src/icg_generator.py lines 29-60)

```
def visit(node):
    if isinstance(node, Number):
        temp = new_temp()
        code_lines.append(f"{temp} = {node.value}")
        return temp

    elif isinstance(node, BinaryOp):
        left = visit(node.left)
        right = visit(node.right)
        temp = new_temp()
        code_lines.append(f"{temp} = {left} {node.op} {right}")
        return temp
```

What This Does:

- Recursively generates code for expressions
- Returns temporary variable holding result
- Builds code bottom-up (leaves first, then parents)

Example:

```
Code: 5 + 3

Execution:
1. visit(Number(5)) → generates "t1 = 5", returns "t1"
```

```
2. visit(Number(3)) → generates "t2 = 3", returns "t2"
3. visit(BinaryOp) → generates "t3 = t1 + t2", returns "t3"

TAC Output:
t1 = 5
t2 = 3
t3 = t1 + t2
```

Assignment Code Generation (src/icg_generator.py lines 24-27)

```
elif cname == "Assign":
    rhs = visit(node.expr)
    code_lines.append(f"{node.name} = {rhs}")
    return node.name
```

What This Does:

- Generates code for right-hand side expression
- Assigns result to variable
- Returns variable name

Example:

```
Code: x = 5 + 3

TAC:
t1 = 5
t2 = 3
t3 = t1 + t2
x = t3
```

Control Flow - If Statement (src/icg_generator.py lines 67-81)

```
elif cname == "IfElse":
    cond = visit(node.condition)
    else_label = new_label()
    end_label = new_label()

    code_lines.append(f"if {cond} == False goto {else_label}")
    for stmt in node.if_body:
        visit(stmt)
    code_lines.append(f"goto {end_label}")
    code_lines.append(f"{else_label}:")
    if node.else_body:
```

```
        for stmt in node.else_body:
            visit(stmt)
    code_lines.append(f"{end_label}:")
```

What This Does:

- Generates conditional jump instructions
- Uses labels for control flow
- Implements if-else logic with gotos

Example:

```
Code:
if x > 10:
    print("Big")
else:
    print("Small")

TAC:
t1 = x
t2 = 10
t3 = t1 > t2
if t3 == False goto L1
t4 = "Big"
print t4
goto L2
L1:
t5 = "Small"
print t5
L2:
```

Control Flow - While Loop (src/icg_generator.py lines 83-94)

```
elif cname == "WhileLoop":
    start_label = new_label()
    end_label = new_label()

    code_lines.append(f"{start_label}:")
    cond = visit(node.condition)
    code_lines.append(f"if {cond} == False goto {end_label}")
    for stmt in node.body:
        visit(stmt)
    code_lines.append(f"goto {start_label}")
    code_lines.append(f"{end_label}:")
```

What This Does:

- Creates loop with start and end labels

- Checks condition at loop start
- Jumps back to start after body
- Exits loop when condition false

Example:

```
Code:
while x > 0:
    x = x - 1

TAC:
L1:
t1 = x
t2 = 0
t3 = t1 > t2
if t3 == False goto L2
t4 = x
t5 = 1
t6 = t4 - t5
x = t6
goto L1
L2:
```

Function Definition (src/icg_generator.py lines 121-127)

```
elif cname == "FunctionDef":
    code_lines.append(f"function {node.name}:")
    for param in node.params:
        code_lines.append(f"param {param}")
    for stmt in node.body:
        visit(stmt)
    return None
```

What This Does:
- Marks function start with label
- Lists parameters
- Generates code for function body

Example:

```
Code:
def add(a, b):
    return a + b

TAC:
function add:
```

```
param a
param b
t1 = a
t2 = b
t3 = t1 + t2
return t3
```

---

2. Tree-Walking Interpreter

Interpreter Architecture

```
class Interpreter:
    def __init__(self, output_buffer=None):
        self.globals = {}
        self.locals_stack = [{}]
        self.functions = {}
        self.output_buffer = output_buffer or sys.stdout
        self.recursion_depth = 0
        self.max_recursion = 1000
```

What This Does:
- Maintains global and local variable scopes
- Stores function definitions
- Tracks recursion depth to prevent stack overflow
- Redirects output to buffer for capture

Expression Evaluation

```
def eval_expr(self, node):
    if isinstance(node, Number):
        return node.value

    elif isinstance(node, String):
        return node.value

    elif isinstance(node, Identifier):
        return self.get_variable(node.name)

    elif isinstance(node, BinaryOp):
        left = self.eval_expr(node.left)
        right = self.eval_expr(node.right)
        return self.apply_op(left, node.op, right)
```

What This Does:
- Recursively evaluates expressions
- Returns actual Python values
- Handles variables, literals, and operations

Example:

```
Code: x = 5; y = x + 3

Execution:
1. eval_expr(Number(5)) → returns 5
2. Assign: self.globals['x'] = 5
3. eval_expr(Identifier('x')) → returns 5 (from globals)
4. eval_expr(Number(3)) → returns 3
5. eval_expr(BinaryOp) → returns 5 + 3 = 8
6. Assign: self.globals['y'] = 8
```

Operator Application

```
def apply_op(self, left, op, right):
    if op == '+':
        return left + right
    elif op == '-':
        return left - right
    elif op == '*':
        return left * right
    elif op == '/':
        if right == 0:
            raise ZeroDivisionError("Division by zero")
        return left / right
    elif op == '>':
        return left > right
    elif op == '==':
        return left == right
    # ... more operators
```

What This Does:
- Implements actual operation logic
- Handles division by zero
- Returns computed result

Type Polymorphism:
- `5 + 3` → `8` (integer addition)
- `"Hello" + "World"` → `"HelloWorld"` (string concatenation)
- Python's duck typing handles this automatically

Function Execution with Recursion

```
def execute_function(self, name, args):
    if name not in self.functions:
        raise NameError(f"Function '{name}' not defined")

    func = self.functions[name]

    # Check recursion depth
    self.recursion_depth += 1
    if self.recursion_depth > self.max_recursion:
        raise RecursionError("Maximum recursion depth exceeded")

    # Create new local scope
    self.locals_stack.append({})

    # Bind parameters
    for param, arg in zip(func.params, args):
        arg_value = self.eval_expr(arg)
        self.locals_stack[-1][param] = arg_value

    # Execute function body
    result = None
    for stmt in func.body:
        result = self.execute_stmt(stmt)
        if isinstance(stmt, Return):
            break

    # Clean up
    self.locals_stack.pop()
    self.recursion_depth -= 1

    return result
```

What This Does:
- Checks if function exists
- Prevents infinite recursion
- Creates new scope for local variables
- Binds arguments to parameters
- Executes function body
- Handles return values
- Cleans up scope after execution

Example - Factorial:

```
Code:
def factorial(n):
    if n <= 1:
```

```
        return 1
    else:
        return n * factorial(n - 1)

result = factorial(5)

Execution Trace:
factorial(5):
  locals: {n: 5}
  5 > 1, so return 5 * factorial(4)

  factorial(4):
    locals: {n: 4}
    4 > 1, so return 4 * factorial(3)

    factorial(3):
      locals: {n: 3}
      3 > 1, so return 3 * factorial(2)

      factorial(2):
        locals: {n: 2}
        2 > 1, so return 2 * factorial(1)

        factorial(1):
          locals: {n: 1}
          1 <= 1, so return 1

        returns 2 * 1 = 2
      returns 3 * 2 = 6
    returns 4 * 6 = 24
  returns 5 * 24 = 120

Final result: 120
```

Loop Execution

```
def execute_for_loop(self, node):
    iterable = self.eval_expr(node.iterable)

    for value in iterable:
        self.set_variable(node.var, value)

        for stmt in node.body:
            result = self.execute_stmt(stmt)
            if isinstance(stmt, Break):
                return
            if isinstance(stmt, Continue):
                break
```

What This Does:
- Evaluates iterable expression
- Iterates over values
- Sets loop variable
- Executes loop body
- Handles break and continue

Example:

```
Code:
for i in range(3):
    print(i)

Execution:
1. eval_expr(RangeCall(0, 3, 1)) → returns [0, 1, 2]
2. Iteration 1: i = 0, print(0)
3. Iteration 2: i = 1, print(1)
4. Iteration 3: i = 2, print(2)

Output:
0
1
2
```

Exception Handling

```
def execute_try_except(self, node):
    try:
        for stmt in node.try_body:
            self.execute_stmt(stmt)
    except Exception as e:
        for stmt in node.except_body:
            self.execute_stmt(stmt)
```

What This Does:
- Executes try block
- Catches any exception
- Executes except block on error
- Prevents program crash

Example:

```
Code:
try:
```

```
    x = 10 / 0
except:
    print("Error occurred")

Execution:
1. Execute try block
2. Division by zero raises exception
3. Catch exception
4. Execute except block
5. Print "Error occurred"

Output:
Error occurred
```

---

3. Built-in Functions

Print Function

```
def execute_print(self, node):
    value = self.eval_expr(node.expr)
    print(value, file=self.output_buffer)
```

What This Does:
- Evaluates expression
- Prints to output buffer
- Allows output capture for web interface

Range Function

```
def eval_range(self, node):
    start = self.eval_expr(node.start) if node.start else 0
    stop = self.eval_expr(node.stop)
    step = self.eval_expr(node.step) if node.step else 1
    return list(range(start, stop, step))
```

What This Does:
- Implements Python's `range()` function
- Handles 1, 2, or 3 arguments
- Returns list of integers

Examples:

- `range(5)` → `[0, 1, 2, 3, 4]`
- `range(2, 5)` → `[2, 3, 4]`
- `range(0, 10, 2)` → `[0, 2, 4, 6, 8]`

Len Function

```
def eval_len(self, node):
    value = self.eval_expr(node.expr)
    return len(value)
```

What This Does:

- Evaluates expression (list, string, etc.)
- Returns length using Python's built-in `len()`

---

4. Data Structure Support

List Operations

```
# List creation
def eval_list(self, node):
    return [self.eval_expr(elem) for elem in node.elements]

# List indexing
def eval_index(self, node):
    lst = self.eval_expr(node.expr)
    idx = self.eval_expr(node.index)
    return lst[idx]

# List assignment
def execute_list_assign(self, node):
    lst = self.get_variable(node.name)
    idx = self.eval_expr(node.index)
    value = self.eval_expr(node.value)
    lst[idx] = value
```

What This Does:

- Creates lists from elements
- Supports indexing with `[]`
- Allows item assignment

Example:

```
Code:
numbers = [1, 2, 3]
numbers[1] = 10
print(numbers[1])

Execution:
1. Create list: [1, 2, 3]
2. Assign to 'numbers'
3. Get list from 'numbers'
4. Set index 1 to 10: [1, 10, 3]
5. Get index 1: returns 10
6. Print: 10
```

Key Achievements in Interpreter Role:

- ☑ **Generated Three-Address Code** - Clean intermediate representation
- ☑ **Built Tree-Walking Interpreter** - Direct AST execution
- ☑ **Implemented Recursion** - With depth limiting for safety
- ☑ **Function Call Stack** - Proper scope management
- ☑ **Exception Handling** - Try/except block execution
- ☑ **Built-in Functions** - print, range, len, str, int
- ☑ **Data Structures** - Lists with indexing and assignment
- ☑ **Control Flow** - if/else, while, for, break, continue
- ☑ **Runtime Safety** - Division by zero, undefined variables
- ☑ **Output Capture** - Redirectable output for web interface

---

🔄 How All Roles Work Together

Complete Compilation Flow Example

Let's trace a simple program through all four roles:

```
def factorial(n):
    if n <= 1:
        return 1
    else:
        return n * factorial(n - 1)
```

```
print(factorial(5))
```

Step 1: Frontend (Role 1)

1. User types code in Streamlit editor
2. Clicks "Run / Analyze Pipeline" button
3. Frontend calls `run_compiler_pipeline(code)`
4. Orchestrates all phases and collects results
5. Displays results in tabbed interface

Step 2: Lexer (Role 2)

```
tokens = tokenize(code)
```

Output:

```
[
  {'type': 'DEF', 'value': 'def', 'line': 1},
  {'type': 'IDENTIFIER', 'value': 'factorial', 'line': 1},
  {'type': 'LPAREN', 'value': '(', 'line': 1},
  {'type': 'IDENTIFIER', 'value': 'n', 'line': 1},
  {'type': 'RPAREN', 'value': ')', 'line': 1},
  {'type': 'COLON', 'value': ':', 'line': 1},
  {'type': 'IF', 'value': 'if', 'line': 2},
  ...
]
```

Frontend displays: Token table in "Lexer" tab

Step 3: Parser (Role 3)

```
ast = parser.parse(code)
```

Output (AST):

```
[
  FunctionDef(
    name='factorial',
    params=['n'],
    body=[
      IfElse(
        condition=BinaryOp(Identifier('n'), '<=', Number(1)),
        if_body=[Return(Number(1))],
        else_body=[
          Return(
            BinaryOp(
              Identifier('n'),
              '*',
              FunctionCall('factorial', [
                BinaryOp(Identifier('n'), '-', Number(1))
              ])
            )
          )
        ]
      )
    ]
  ),
  Print(FunctionCall('factorial', [Number(5)]))
]
```

Frontend displays: Visual tree diagram in "Parser" tab

Step 4: Semantic Analysis (Role 3)

```
is_valid, output = semantic_analysis(ast)
```

Checks:
- ☑ Function 'factorial' is defined before being called
- ☑ Parameter 'n' is in scope within function
- ☑ Return statements are inside function
- ☑ Types are compatible in operations

Output:

```
Semantic Analysis Complete
✓ All variables declared before use
✓ All functions defined before calls
✓ Type checking passed
```

Frontend displays: Validation results in "Semantic" tab

Step 5: Intermediate Code Generation (Role 4)

```
icg = generate_icg(ast)
```

Output (Three-Address Code):

```
function factorial:
param n
t1 = n
t2 = 1
t3 = t1 <= t2
if t3 == False goto L1
t4 = 1
return t4
goto L2
L1:
t5 = n
t6 = n
t7 = 1
t8 = t6 - t7
t9 = call factorial(t8)
t10 = t5 * t9
return t10
L2:
t11 = 5
t12 = call factorial(t11)
print t12
```

Frontend displays: TAC in "ICG" tab

Step 6: Execution (Role 4)

```
interpreter = Interpreter()
interpreter.execute(code)
```

Execution Trace:

```
1. Define function 'factorial'
2. Call factorial(5)
   3. n = 5, 5 <= 1? No
   4. Return 5 * factorial(4)
      5. n = 4, 4 <= 1? No
      6. Return 4 * factorial(3)
```

```
         7. n = 3, 3 <= 1? No
         8. Return 3 * factorial(2)
            9. n = 2, 2 <= 1? No
            10. Return 2 * factorial(1)
                11. n = 1, 1 <= 1? Yes
                12. Return 1
            13. Return 2 * 1 = 2
         14. Return 3 * 2 = 6
      15. Return 4 * 6 = 24
   16. Return 5 * 24 = 120
17. Print 120
```

Output:

```
120
```

Frontend displays: "120" in "Output" tab

📊 Summary of Contributions

Role	Lines of Code	Key Files	Main Achievement
Frontend	~220	app.py	Interactive web interface with 5-phase visualization
Lexer	~250	lexer.py	47 token types, regex patterns, formatted output
Parser	~350	myparser.py, ast*nodes.py,* *semantic*analyzer.py, utils.py	30+ grammar rules, 20+ AST nodes, type checking, visualization
Interpreter	~400	icg_generator.py, interpreter.py	TAC generation, tree-walking execution, recursion support

Total: ~1,220 lines of production code

🎯 What Makes This Project Special

Educational Value

- **Visualizes Abstract Concepts** - Makes compiler theory tangible
- **Interactive Learning** - Immediate feedback on code changes
- **Complete Pipeline** - Shows all compilation stages

Technical Depth

- **Real Parser** - Uses industry-standard PLY framework
- **Proper AST** - Not just string manipulation
- **Type Checking** - Actual semantic analysis
- **Recursion Support** - Handles complex programs

Professional Quality

- **Modern UI** - Streamlit-based web interface
- **Error Handling** - Graceful degradation at each stage
- **Export Features** - Download all intermediate representations
- **Documentation** - Extensive inline comments and tooltips

🚀 Future Enhancements

Frontend

- Add step-by-step debugger
- Syntax highlighting in editor
- Performance metrics visualization

Lexer

- Support for comments in output
- Better error messages with suggestions
- Unicode identifier support

Parser

- Class and object support
- List comprehensions
- Lambda functions
- Decorator syntax

Interpreter

- Bytecode compilation
- JIT optimization
- Garbage collection
- Async/await support

🎓 Conclusion This project demonstrates mastery of: - **Compiler Design Theory** - Lexical analysis, parsing, semantic analysis, code generation - **Software Architecture** - Modular design, separation of concerns - **Web Development** - Interactive UI with Streamlit - **Python Programming** - Advanced features, OOP, recursion - **Visualization** - Graphviz integration, data presentation **Each role contributes essential functionality, and together they create a complete, educational compiler visualizer.** *Built with dedication to compiler theory and software craftsmanship* 🚀