Interpreter – Part 2
Example

\[ \text{index} = 2 \times \text{count} + 17; \]

- **Lexemes** and **tokens** of this statement are:

<table>
<thead>
<tr>
<th>Lexemes</th>
<th>Tokens</th>
</tr>
</thead>
<tbody>
<tr>
<td>index</td>
<td>Identifier</td>
</tr>
<tr>
<td>=</td>
<td>Equal sign</td>
</tr>
<tr>
<td>2</td>
<td>int_literal</td>
</tr>
<tr>
<td>*</td>
<td>mult_op</td>
</tr>
<tr>
<td>count</td>
<td>identifier</td>
</tr>
<tr>
<td>+</td>
<td>add_op</td>
</tr>
<tr>
<td>17</td>
<td>int_literal</td>
</tr>
<tr>
<td>;</td>
<td>semicolon</td>
</tr>
</tbody>
</table>
Lexemes/Tokens

• A **lexeme** is the lowest level syntactic unit of a language (e.g., *, sum, begin)

• A **token** is a category of lexemes (e.g., identifier)
Lexical Analysis

- Lexical analysis is the process of converting a sequence of characters into a sequence of tokens.
- A program or function which performs lexical analysis is called a **lexical analyzer**, **lexer**, or **scanner**.
- A **lexer** often exists as a single function which is called by a parser or another function.
Translation Stages

• Lexical analysis (scanning)
  o Process of converting a sequence of characters into a sequence of tokens

• Parsing
  o Find all syntax errors; for each, produce an appropriate diagnostic message and recover quickly
Interpreter

The interpreter operates in a **read-eval-print** loop.

1. The **Scanner** reads the input program
2. The **Parser** gets a statement worth of lexemes from the Scanner and builds a parse tree.
3. **eval** is a method that evaluates the parse tree.
4. The result of each statement is printed.
Expression Evaluation

- For binary operators (PLUS, MINUS, TIMES, DIVIDE)
  - Evaluate the left child
  - Evaluate the right child
  - Perform the specified operation on the values
  - Return a Lexeme containing the result

- For unary operations, there is only one child
  - Evaluate that child
  - Perform the operation
Evaluating an expression

- From top
  - Evaluate left child of + (recursive call to eval)
    - Evaluate left child of / (look up b)
    - Evaluate right child of /
      - Evaluate left child of - (look up a)
      - Evaluate right child of - (look up b)
        - Subtract b from a
    - Divide b by rhs value
      - Evaluate right child of + to get 1
      - Add lhs and rhs to get result
Evaluation and Control Structures

• How do we build parse trees for control structures?

• We need some extra node types (structural tokens) to glue the parts of these statements together
  o An \texttt{if} statement can have three parts
  o We need to be able to make lists of statements.

• The \texttt{eval} method needs new cases for these statements.
Boolean expressions

• The test conditions for control structures need boolean expressions
• These are binary so the parse trees look like those for arithmetic expressions
  o Except for the NOT operator
• The NOT operator has only one child
Lists in Parse Trees

• To represent a list, you need some kind of backbone node to separate different elements in the list

• The token could be **LIST**
**if statement**

- An **if** statement has a test condition, a **then** part and possibly an **else** part.
- The parser needs to build a tree with places for these three parts
  - use a structural node to separate the **then** part and the **else** part
- To evaluate, evaluate the test condition
  - if **true**, evaluate the **then** part
  - if **false**, evaluate the **else** part or do nothing if there isn’t one
Parse tree for `if` statement
while loop

• A **while** statement has
  o a test condition
  o followed by a list of one or more statements.

• The parser needs to build a parse tree for the whole **while** which will include
  o a parse tree for the test condition and
  o a list of parse trees for the statements in the body of the loop
while loop

- The evaluator needs a loop that evaluates the test condition and then evaluates the body of the loop if the condition is true
Parse Tree for while loop
Implementing Functions

• Need to be able to define functions and call them
• A function definition is a new kind of statement
  o Token is FUNCTION
• A function call can be a statement or it can be an expression
  o Token is FN_CALL
Evaluating a Function Definition

• Build a *parse tree* containing the parameter list and the statement list.

• For *static scope*, you also need the environment that is active when the function is defined.

• Store this parse tree in the environment with the function name as the key.
Parse Tree for Function Definition
Parse Tree for Function Call

• The parse tree needs the name of the function and a list of the expressions used for arguments
Parse Tree for Function Call
Evaluating a function call

• Look up the function in the environment
• Create a new environment
  o Extend the one stored with the function for static scope
  o Extend the current one for dynamic scope
Evaluating a function call

• For each argument,
  o evaluate it and
  o store it in the new environment as the value of the corresponding parameter

• Evaluate the body of the function in this new environment