COMPILER CONSTRUCTION

**Optimization in a Mini Compiler**

Optimization in a mini compiler involves improving the intermediate representation or code to make the final output more efficient. This can include reducing execution time, memory usage, or other resources. A common example is **constant folding** and **dead code elimination**.

### 1. ****Constant Folding****

**Purpose**: Constant folding simplifies expressions with constant values during compilation. Instead of performing calculations at runtime, the compiler evaluates them at compile time.

**Input Source Code:**

int x = 5 \* 3;

int y = x + 2;

#### **Optimized Intermediate Code:**

int x = 15;

int y = 17;

**Implementation in Mini Compiler**:

public class Optimizer

{

public static Node Optimize(Node node)

{

if (node is BinaryExpressionNode binaryNode)

{

var left = Optimize(binaryNode.Left);

var right = Optimize(binaryNode.Right);

if (left is NumberNode leftNum && right is NumberNode rightNum)

{

int result = binaryNode.Operator.Value switch

{

"+" => leftNum.Value + rightNum.Value,

"-" => leftNum.Value - rightNum.Value,

"\*" => leftNum.Value \* rightNum.Value,

"/" => leftNum.Value / rightNum.Value,

\_ => throw new Exception("Unsupported operator")

};

return new NumberNode(new Token("Number", result.ToString()));

}

return new BinaryExpressionNode(left, binaryNode.Operator, right);

}

return node;

}

}

// Example usage

class Program

{

static void Main(string[] args)

{

// AST for `5 \* 3 + 2`

var expression = new BinaryExpressionNode(

new BinaryExpressionNode(

new NumberNode(new Token("Number", "5")),

new Token("Operator", "\*"),

new NumberNode(new Token("Number", "3"))

),

new Token("Operator", "+"),

new NumberNode(new Token("Number", "2"))

);

Node optimized = Optimizer.Optimize(expression);

Console.WriteLine("Optimization completed.");

}

}

**// Example usage**

class Program

{

static void Main(string[] args)

{

// AST for `5 \* 3 + 2`

var expression = new BinaryExpressionNode(

new BinaryExpressionNode(

new NumberNode(new Token("Number", "5")),

new Token("Operator", "\*"),

new NumberNode(new Token("Number", "3"))

),

new Token("Operator", "+"),

new NumberNode(new Token("Number", "2"))

);

Node optimized = Optimizer.Optimize(expression);

Console.WriteLine("Optimization completed.");

}

}

**Explanation**:

* The optimizer checks if conditions are constant (e.g., true or false).
* Based on the condition's value, it eliminates the unused branch.

# Impact of Optimization

* **Constant Folding** reduces runtime computations, making the program faster.
* **Dead Code Elimination** reduces code size and avoids unnecessary processing.

These optimizations, though basic, showcase how a compiler can enhance the performance of the compiled program.