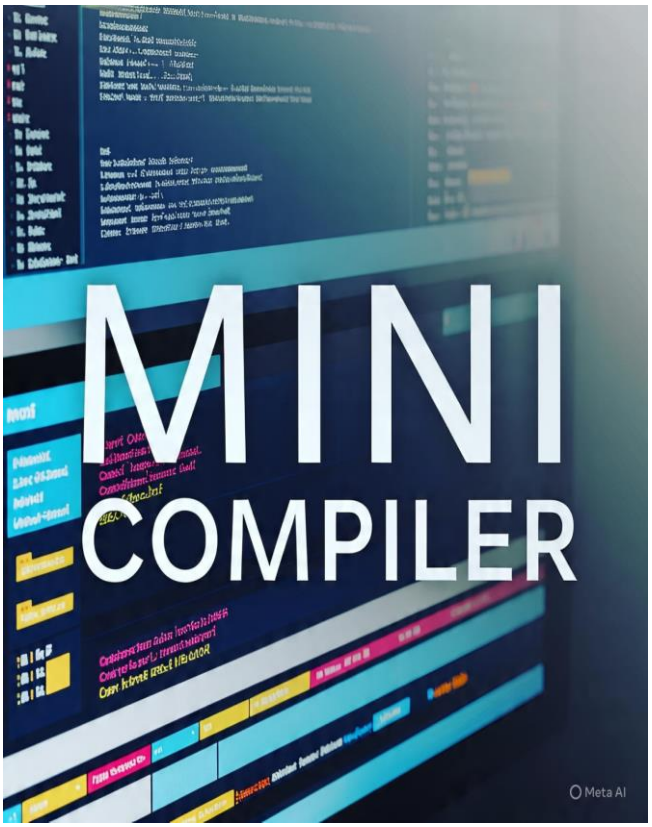


MINI COMPILER

All Phases Implemented

COMSATS UNIVERSITY ISLAMABAD, ATTOCK CAMPUS

Semester Project

Submitted by:

Akasha(sp22-bcs-017)

Areeba(sp22-bcs-047)

Submitted to:

Sir Bilal Haider

Course title:

Compiler Construction

Deadline: May 30, 2025

Table of Contents:

Introduction: 2

Purpose: 2

Phases of Compiler: 2

1. Lexical Analysis: 2

2. Syntax Analysis (Parsing): 2

3. Semantic Analysis: 2

4. Intermediate Code Generation: 2

5. Optimization: 3

6. Target Code Generation: 3

7. Symbol Table Management: 3

8. Error Handling: 3

Diagrams: 3

Sequence Diagram: 3

Class Diagram: 4

Activity Diagram: 4

Code: 5

Output: 24

Mini Compiler

Introduction:

A compiler is a fundamental component in computer science that translates high-level programming language (such as C or Java) into low-level machine code or an intermediate form. A mini compiler is a simplified version of a full-scale compiler that performs essential tasks like lexical, syntax, and semantic analysis, followed by intermediate code generation and optional optimizations.

Purpose:

The purpose of building a mini compiler is to understand the core concepts of compiler design, including parsing, tokenization, semantic checking, and code generation. It also gives hands-on experience with tools like Lex or building components manually using programming languages such as C, Java, or Python.

Phases of Compiler:

1. Lexical Analysis:

This phase reads the source code character by character and groups them into meaningful tokens such as keywords, identifiers, and operators. It also removes unnecessary whitespaces and comments. Tools like Lex or a custom tokenizer can be used here.

2. Syntax Analysis (Parsing):

The syntax analyzer checks whether the sequence of tokens follows the language's grammar rules. It generates a parse tree or abstract syntax tree (AST) that represents the structure of the code. Tools like Yacc or recursive descent parsers are commonly used.

3. Semantic Analysis:

This phase verifies the semantic correctness of the program, such as type checking and ensuring variables are declared before use. It adds type and scope information to the AST and reports semantic errors if found.

4. Intermediate Code Generation:

The AST is converted into an intermediate representation (IR), such as three-address code, which is easier to analyze and optimize. This code acts as a bridge between the high-level source and the low-level machine code.

5. Optimization:

This phase improves the intermediate code by removing redundancies and increasing efficiency. Common techniques include constant folding**,** dead code elimination**,** and loop optimizations. Though optional, it enhances performance.

6. Target Code Generation:

The optimized IR is translated into target machine code or assembly instructions. This output is executable by the hardware or virtual machine. The generated code must be correct and efficient.

7. Symbol Table Management:

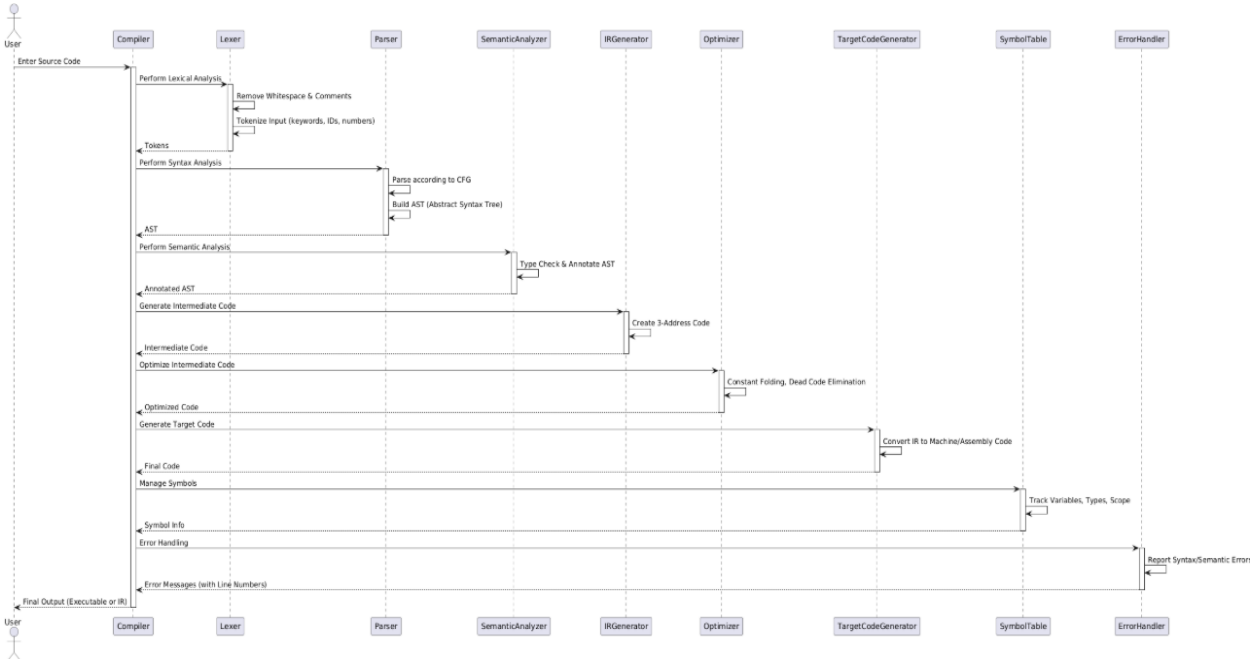
Throughout compilation, a symbol table is maintained to store information about identifiers, such as variable names, data types, scopes, and memory addresses. It supports quick lookups and consistency checks.

8. Error Handling:

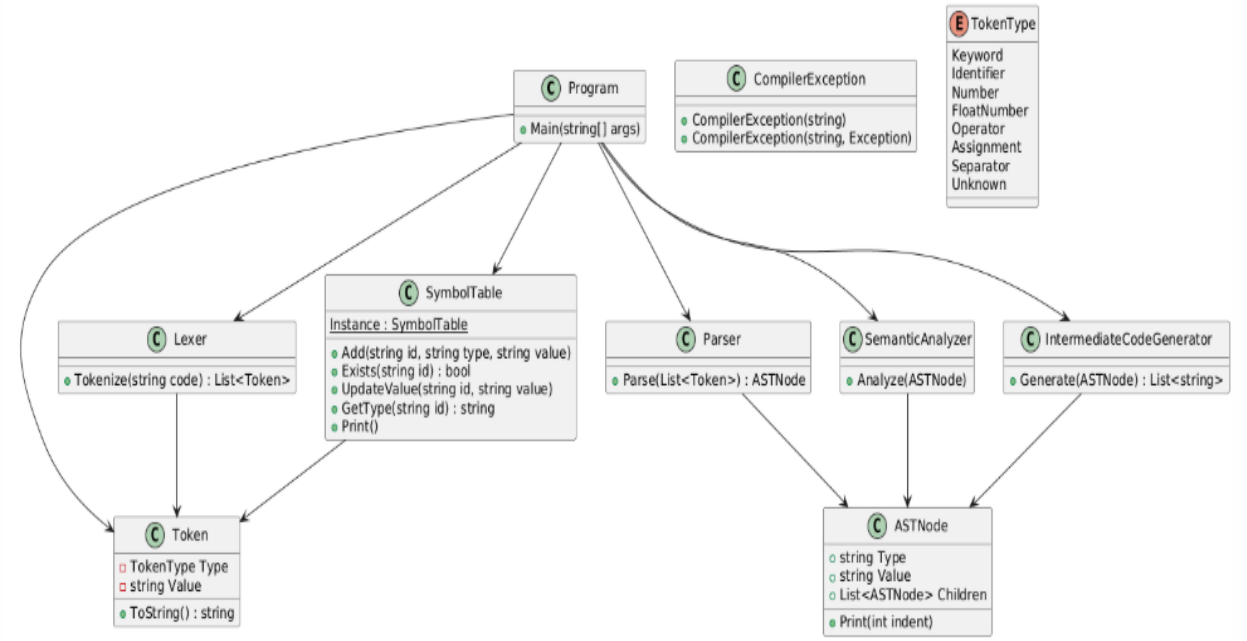
This phase handles both syntax and semantic errors, providing clear and helpful messages with line numbers. It ensures the compiler continues processing as much as possible to detect multiple errors in one pass.

Diagrams:

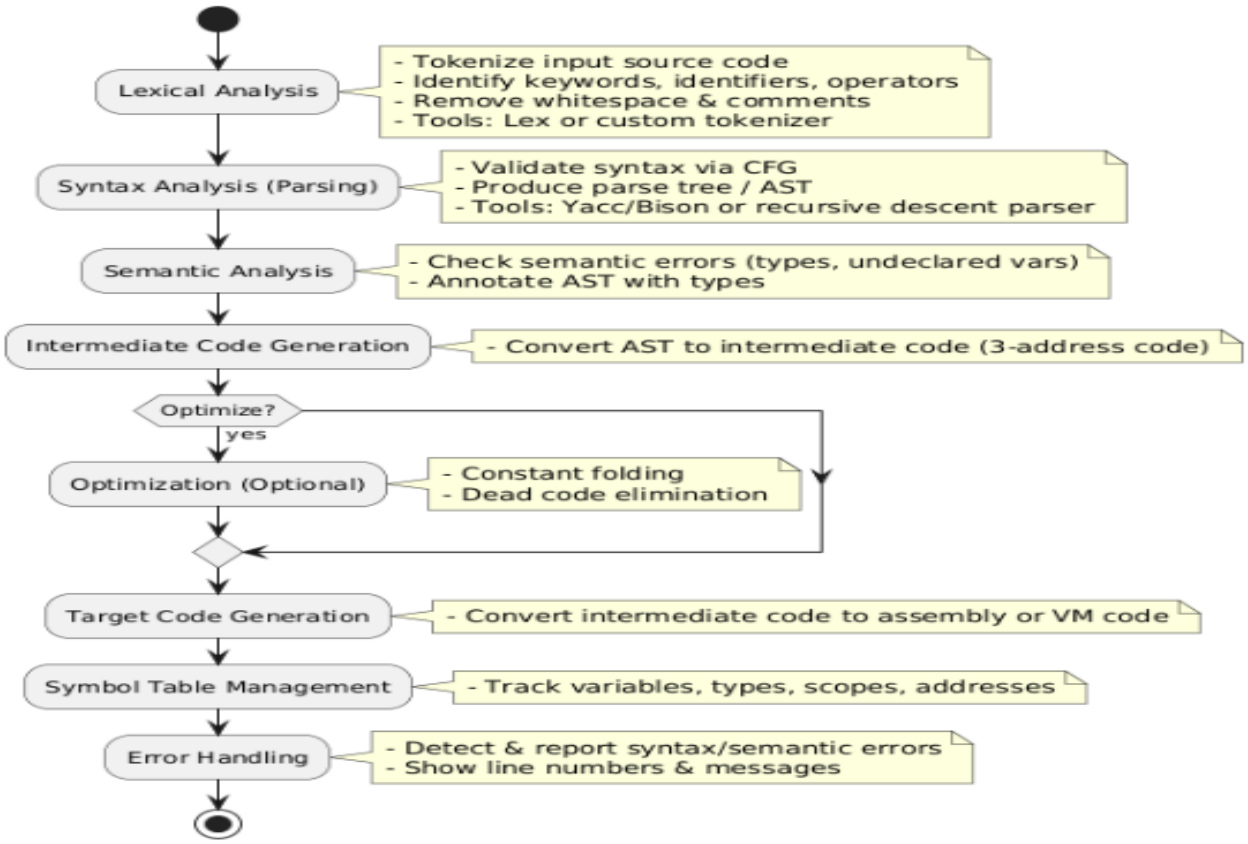
Sequence Diagram:



Class Diagram:



Activity Diagram:



Code:

```
// Updated Mini Compiler with float support and better parsing


using System;

using System.Collections.Generic;

using System.Text.RegularExpressions;


namespace MiniCompiler

{

   class Program

   {

      static void Main(string[] args)

      {

         try

         {

            Console.WriteLine("Enter source code (end with an empty line):");

            string line;

            string inputCode = "";


            while (!string.IsNullOrWhiteSpace(line = Console.ReadLine()))

            {

               inputCode += line + "\n";

            }


            // Lexical Analysis

            Console.WriteLine("\n=== LEXICAL ANALYSIS ===");

            var lexer = new Lexer();

            var tokens = lexer.Tokenize(inputCode);
```

```
    Console.WriteLine("\n--- Tokens ---");
    foreach (var token in tokens)
        Console.WriteLine(token);

    // Syntax Analysis
    Console.WriteLine("\n=== SYNTAX ANALYSIS ===");
    var parser = new Parser();
    var ast = parser.Parse(tokens);

    // Semantic Analysis
    Console.WriteLine("\n=== SEMANTIC ANALYSIS ===");
    var semanticAnalyzer = new SemanticAnalyzer();
    semanticAnalyzer.Analyze(ast);

    // Intermediate Code Generation
    Console.WriteLine("\n=== INTERMEDIATE CODE GENERATION ===");
    var codeGen = new IntermediateCodeGenerator();
    var ir = codeGen.Generate(ast);

    Console.WriteLine("\n--- Intermediate Code ---");
    foreach (var lineCode in ir)
        Console.WriteLine(lineCode);

    // Symbol Table
    Console.WriteLine("\n=== SYMBOL TABLE ===");
    SymbolTable.Instance.Print();
}
catch (CompilerException ex)
```

```
        {
            Console.WriteLine($"\nCOMPILATION FAILED: {ex.Message}");
            if (ex.InnerException != null)
            {
                Console.WriteLine($"Details: {ex.InnerException.Message}");
            }
        }
        catch (Exception ex)
        {
            Console.WriteLine($"\nUNEXPECTED ERROR: {ex.Message}");
        }
    }
}

class CompilerException : Exception
{
    public CompilerException(string message) : base(message) { }
    public CompilerException(string message, Exception inner) : base(message, inner) { }
}

enum TokenType { Keyword, Identifier, Number, FloatNumber, Operator, Assignment, Separator,
Unknown }

class Token
{
    public TokenType Type;
    public string Value;
    public Token(TokenType type, string value)
    {
```

```
        Type = type; Value = value;
    }
    public override string ToString() => $"[{Type}] {Value}";
}

class Lexer
{
    string[] keywords = { "int", "float", "print" };
    public List<Token> Tokenize(string code)
    {
        var tokens = new List<Token>();
        var pattern = @"(?<Keyword>\bint\b|\bfloat\b|\bprint\b)|" +
          @"(?<Identifier>[a-zA-Z_][a-zA-Z0-9_]*)|" +
          @"(?<FloatNumber>\d+\.\d+)|" +
          @"(?<Number>\d+)|" +
          @"(?<Assignment>=)|" +
          @"(?<Operator>[+\-*/])|" +
          @"(?<Separator>[;()])";

        var regex = new Regex(pattern);
        var lines = code.Split('\n');

        for (int lineNum = 0; lineNum < lines.Length; lineNum++)
        {
            var line = lines[lineNum];
            if (string.IsNullOrWhiteSpace(line)) continue;

            Console.WriteLine($"\nLexing line {lineNum + 1}: {line.Trim()}");
            var matches = regex.Matches(line);
```

```
            foreach (Match match in matches)
            {
                foreach (var groupName in regex.GetGroupNames())
                {
                    if (groupName == "0") continue;
                    if (match.Groups[groupName].Success)
                    {
                        TokenType type = Enum.TryParse(groupName, out TokenType parsedType) ? parsedType
: TokenType.Unknown;
                        tokens.Add(new Token(type, match.Value));
                        Console.WriteLine($"  Found token: {type} '{match.Value}'");
                        break;
                    }
                }
            }
        }
    }

    class ASTNode
    {
        public string Type;
        public string Value;
        public List<ASTNode> Children = new List<ASTNode>();
        public ASTNode(string type, string value = "")
        {
            Type = type; Value = value;
        }
```

```
    public void Print(int indent = 0)
    {
        Console.WriteLine($"{new string(' ', indent * 2)}{Type}: {Value}");
        foreach (var child in Children)
            child.Print(indent + 1);
    }
}

class Parser
{
    List<Token> tokens;
    int position;

    public ASTNode Parse(List<Token> tokens)
    {
        this.tokens = tokens;
        position = 0;
        var root = new ASTNode("Program");

        try
        {
            while (position < tokens.Count)
            {
                var stmt = ParseStatement();
                if (stmt != null)
                    root.Children.Add(stmt);
            }
        }
```

```
    catch (Exception ex)
    {
        throw new CompilerException($"Syntax error at position {position}: {ex.Message}", ex);
    }

    Console.WriteLine("\n--- Abstract Syntax Tree ---");
    root.Print();
    return root;
}

ASTNode ParseStatement()
{
    if (Match(TokenType.Keyword, "int") || Match(TokenType.Keyword, "float"))
    {
        string type = tokens[position - 1].Value;
        Console.WriteLine($"\n[Syntax] Parsing {type} variable declaration...");
        var decl = new ASTNode("Declaration", type);
        var id = Expect(TokenType.Identifier);
        decl.Children.Add(new ASTNode("Identifier", id.Value));
        Console.WriteLine($"  Found identifier: {id.Value}");

        if (Match(TokenType.Assignment))
        {
            Console.WriteLine("  Found assignment, parsing expression...");
            var expr = ParseExpression();
            decl.Children.Add(expr);
        }

        Expect(TokenType.Separator, ";");
```

```
            Console.WriteLine("  End of declaration statement");
            return decl;
        }
        else if (Peek().Type == TokenType.Identifier && Peek(position + 1)?.Type ==
TokenType.Assignment)
        {
            Console.WriteLine("\n[Syntax] Parsing assignment statement...");
            var id = Expect(TokenType.Identifier);
            Expect(TokenType.Assignment);
            var assign = new ASTNode("Assignment", "=");
            assign.Children.Add(new ASTNode("Identifier", id.Value));
            assign.Children.Add(ParseExpression());
            Expect(TokenType.Separator, ";");
            Console.WriteLine("  End of assignment statement");
            return assign;
        }
        else if (Match(TokenType.Keyword, "print"))
        {
            Console.WriteLine("\n[Syntax] Parsing print statement...");
            var print = new ASTNode("Print", "print");
            var id = Expect(TokenType.Identifier);
            print.Children.Add(new ASTNode("Identifier", id.Value));
            Console.WriteLine($"  Found identifier to print: {id.Value}");
            Expect(TokenType.Separator, ";");
            Console.WriteLine("  End of print statement");
            return print;
        }

        throw new Exception($"Unexpected token: {Peek()?.Value}");
```

```
}

ASTNode ParseExpression()
{
   Console.WriteLine("  Parsing expression...");
   var left = ParseTerm();

   while (Match(TokenType.Operator, "+") || Match(TokenType.Operator, "-"))
   {
      var op = tokens[position - 1];
      Console.WriteLine($"  Found additive operator: {op.Value}");
      var node = new ASTNode("BinaryOp", op.Value);
      node.Children.Add(left);
      node.Children.Add(ParseTerm());
      left = node;
   }
   return left;
}

ASTNode ParseTerm()
{
   var left = ParseFactor();
   while (Match(TokenType.Operator, "*") || Match(TokenType.Operator, "/"))
   {
      var op = tokens[position - 1];
      Console.WriteLine($"  Found multiplicative operator: {op.Value}");
      var node = new ASTNode("BinaryOp", op.Value);
      node.Children.Add(left);
      node.Children.Add(ParseFactor());
```

```
            left = node;
        }
        return left;
    }

    ASTNode ParseFactor()
    {
        if (Peek().Type == TokenType.Number || Peek().Type == TokenType.FloatNumber)
        {
            var num = tokens[position++];
            Console.WriteLine($"  Found number: {num.Value}");
            return new ASTNode("Number", num.Value);
        }
        else if (Peek().Type == TokenType.Identifier)
        {
            var id = Expect(TokenType.Identifier);
            Console.WriteLine($"  Found identifier in expression: {id.Value}");
            return new ASTNode("Identifier", id.Value);
        }
        else
            throw new Exception("Expected identifier or number in expression");
    }

    bool Match(TokenType type, string value = null)
    {
        if (position < tokens.Count && tokens[position].Type == type && (value == null ||
tokens[position].Value == value))
        {
            position++;
```

```
            return true;
        }
        return false;
    }

    Token Expect(TokenType type, string value = null)
    {
        if (position < tokens.Count && tokens[position].Type == type && (value == null ||
tokens[position].Value == value))
            return tokens[position++];
        throw new Exception($"Expected {type} '{value ?? "<any>"}', found: {Peek()?.Value ?? "EOF"}");
    }

    Token Peek(int ahead = 0) => position + ahead < tokens.Count ? tokens[position + ahead] : null;
}

class SemanticAnalyzer
{
    public void Analyze(ASTNode root)
    {
        Console.WriteLine("\nStarting semantic analysis...");

        try
        {
            foreach (var node in root.Children)
            {
                if (node.Type == "Declaration")
                {
                    string type = node.Value;
```

```
    string id = node.Children[0].Value;
    Console.WriteLine($"\nAnalyzing declaration of variable '{id}' as {type}");

    if (SymbolTable.Instance.Exists(id))
    {
        throw new CompilerException($"Variable '{id}' already declared.");
    }

    string value = null;
    if (node.Children.Count > 1)
    {
        Console.WriteLine($"  Checking initialization expression...");
        value = EvaluateExpression(node.Children[1]);
        Console.WriteLine($"  Initial value: {value ?? "null"}");
    }

    SymbolTable.Instance.Add(id, type, value);
    Console.WriteLine($"  Added '{id}' to symbol table as {type}");
}
else if (node.Type == "Assignment")
{
    string id = node.Children[0].Value;
    Console.WriteLine($"\nAnalyzing assignment to variable '{id}'");

    if (!SymbolTable.Instance.Exists(id))
    {
        throw new CompilerException($"Undeclared variable '{id}'");
    }
```

```
            string type = SymbolTable.Instance.GetType(id);
            string value = EvaluateExpression(node.Children[1]);
            Console.WriteLine($"  Assigned value: {value}");

            // Update symbol table with new value
            SymbolTable.Instance.UpdateValue(id, value);
        }
        else if (node.Type == "Print")
        {
            string id = node.Children[0].Value;
            Console.WriteLine($"\nAnalyzing print statement for variable '{id}'");

            if (!SymbolTable.Instance.Exists(id))
            {
                throw new CompilerException($"Undeclared variable '{id}'");
            }

            Console.WriteLine($"  Variable '{id}' is valid for printing");
        }
    }

    Console.WriteLine("Semantic analysis completed successfully");
}
catch (CompilerException)
{
    throw;
}
catch (Exception ex)
{
```

```
        throw new CompilerException("Semantic analysis failed", ex);
    }
}

string EvaluateExpression(ASTNode node)
{
    try
    {
        if (node.Type == "Number")
        {
            Console.WriteLine($"    Found number literal: {node.Value}");
            return node.Value;
        }

        if (node.Type == "Identifier")
        {
            Console.WriteLine($"    Found variable reference: {node.Value}");
            if (!SymbolTable.Instance.Exists(node.Value))
            {
                throw new CompilerException($"Undeclared variable '{node.Value}'");
            }

            string value = SymbolTable.Instance.GetValue(node.Value);
            Console.WriteLine($"    Variable '{node.Value}' has value: {value ?? "null"}");
            return value;
        }

        if (node.Type == "BinaryOp")
        {
```

```
            Console.WriteLine($"    Evaluating binary operation: {node.Value}");
            string left = EvaluateExpression(node.Children[0]);
            string right = EvaluateExpression(node.Children[1]);

            if (left == null || right == null)
            {
                throw new CompilerException($"Cannot perform operation on uninitialized variables");
            }

            Console.WriteLine($"    Operation: {left} {node.Value} {right}");
            return $"{left} {node.Value} {right}";
        }

        return null;
    }
    catch (CompilerException)
    {
        throw;
    }
    catch (Exception ex)
    {
        throw new CompilerException("Expression evaluation failed", ex);
    }
  }
}

class IntermediateCodeGenerator
{
  int tempCount = 0;
```

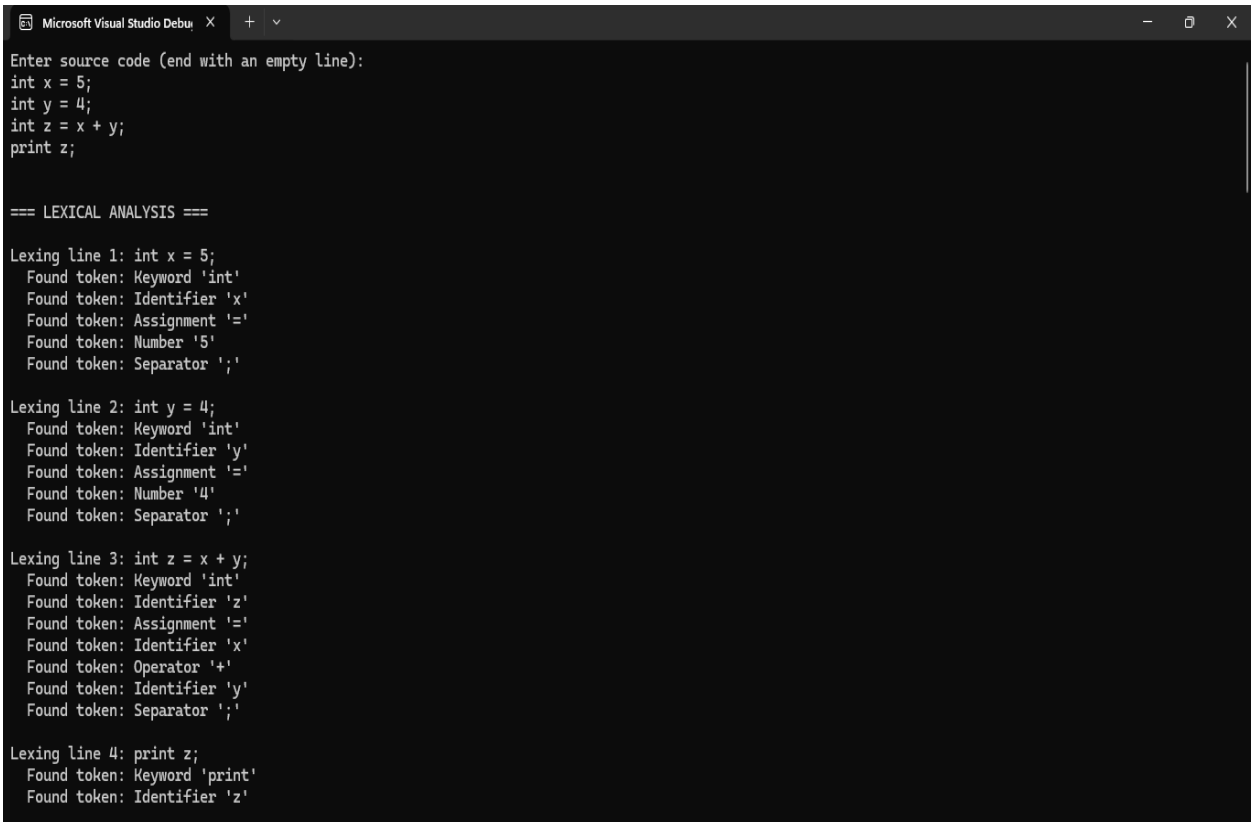
```
public List<string> Generate(ASTNode root)
{
    var code = new List<string>();
    Console.WriteLine("\nGenerating intermediate code...");

    try
    {
        foreach (var node in root.Children)
        {
            if (node.Type == "Declaration")
            {
                string type = node.Value;
                string id = node.Children[0].Value;
                if (node.Children.Count > 1)
                {
                    Console.WriteLine($"  Processing {type} declaration with initialization: {id}");
                    string exprResult = GenerateExpression(node.Children[1], code);
                    code.Add($"{id} = {exprResult}");
                    Console.WriteLine($"    Generated: {id} = {exprResult}");
                }
                else
                {
                    Console.WriteLine($"  Processing {type} declaration without initialization: {id}");
                    code.Add($"DECLARE {id} as {type}");
                    Console.WriteLine($"    Generated: DECLARE {id} as {type}");
                }
            }
            else if (node.Type == "Assignment")
```

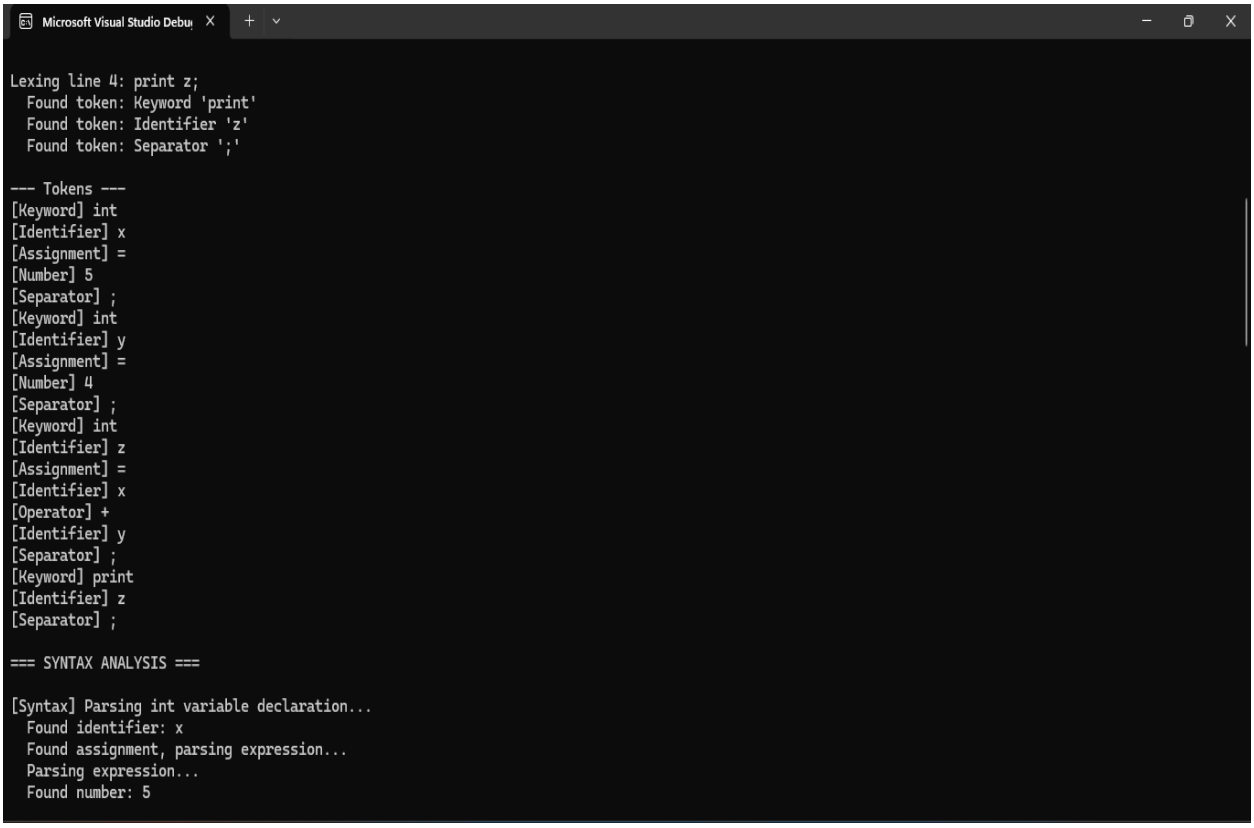
```
            {
                string id = node.Children[0].Value;
                Console.WriteLine($"  Processing assignment to {id}");
                string exprResult = GenerateExpression(node.Children[1], code);
                code.Add($"{id} = {exprResult}");
                Console.WriteLine($"    Generated: {id} = {exprResult}");
            }
            else if (node.Type == "Print")
            {
                string id = node.Children[0].Value;
                code.Add($"PRINT {id}");
                Console.WriteLine($"  Generated print statement for {id}");
            }
        }
    }
    catch (Exception ex)
    {
        throw new CompilerException("Intermediate code generation failed", ex);
    }

    return code;
}

string GenerateExpression(ASTNode node, List<string> code)
{
    if (node.Type == "Number")
    {
        return node.Value;
    }
```

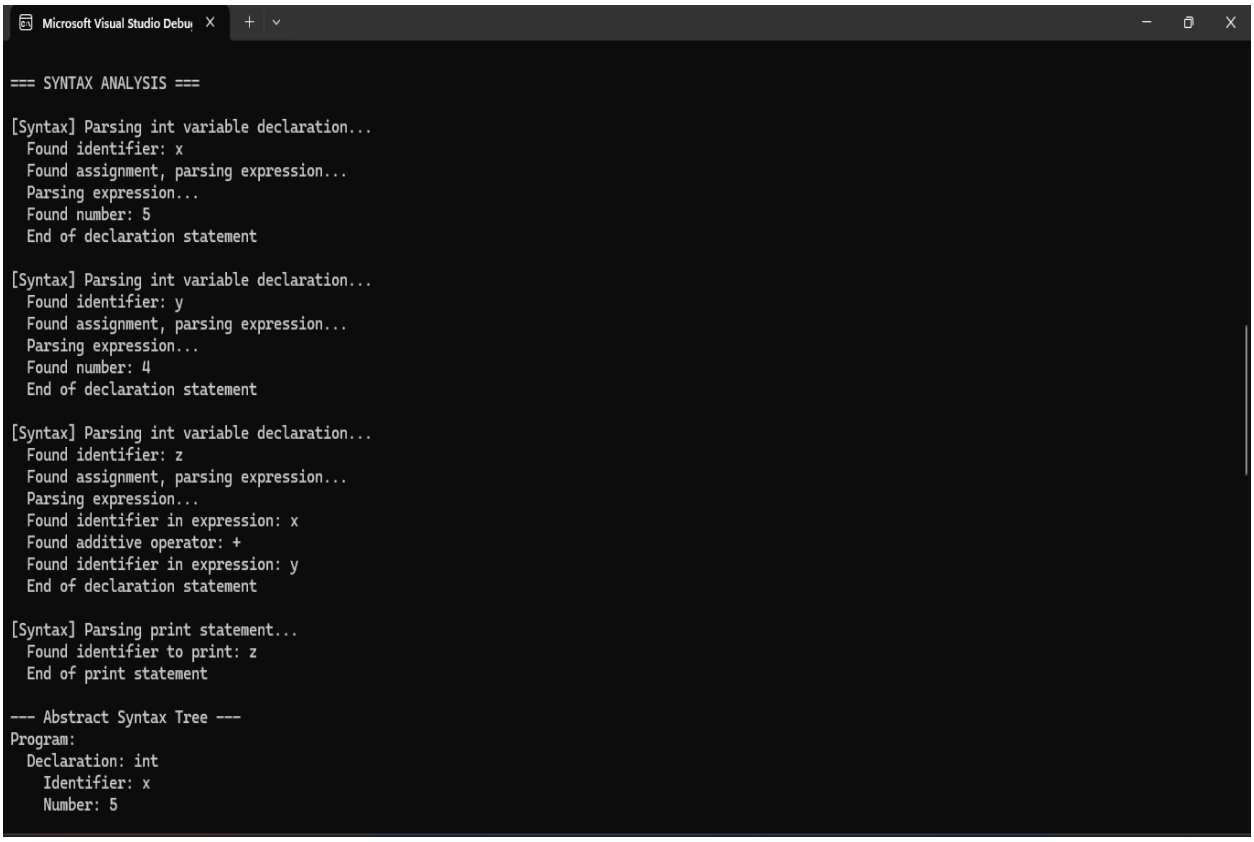
```
        if (node.Type == "Identifier")
        {
            return node.Value;
        }

        if (node.Type == "BinaryOp")
        {
            string left = GenerateExpression(node.Children[0], code);
            string right = GenerateExpression(node.Children[1], code);
            string temp = $"t{tempCount++}";
            code.Add($"{temp} = {left} {node.Value} {right}");
            Console.WriteLine($"    Generated temp operation: {temp} = {left} {node.Value} {right}");
            return temp;
        }

        throw new CompilerException($"Unsupported node type in expression: {node.Type}");
    }
}

class SymbolTable
{
    private Dictionary<string, (string type, string value)> table = new();
    private static SymbolTable _instance;
    public static SymbolTable Instance => _instance ??= new SymbolTable();

    public void Add(string name, string type, string value)
    {
        table[name] = (type, value);
```

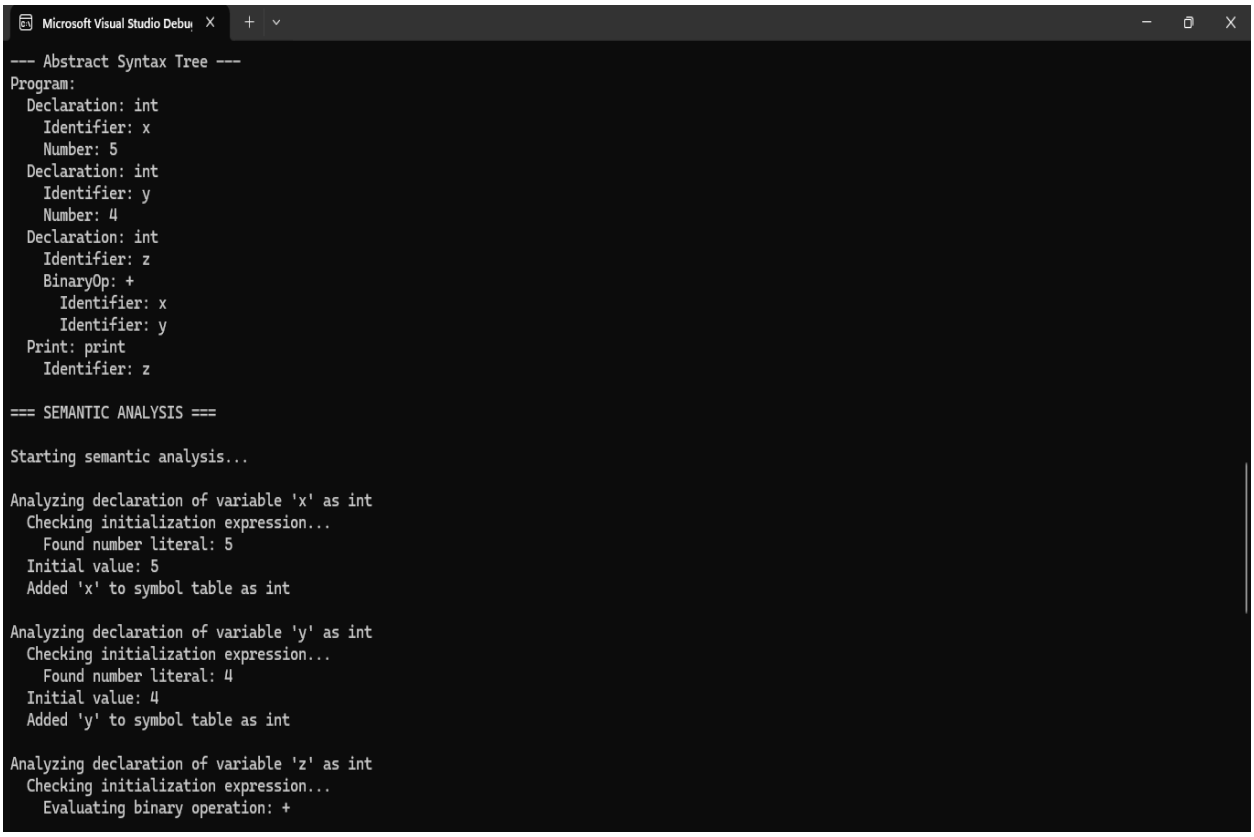
```
    }

    public void UpdateValue(string name, string value)
    {
        if (table.ContainsKey(name))
        {
            table[name] = (table[name].type, value);
        }
    }

    public bool Exists(string name) => table.ContainsKey(name);

    public string GetValue(string name) => table.TryGetValue(name, out var entry) ? entry.value : null;

    public string GetType(string name) => table.TryGetValue(name, out var entry) ? entry.type : null;

    public void Print()
    {
        if (table.Count == 0)
        {
            Console.WriteLine("Symbol table is empty");
            return;
        }

        Console.WriteLine("Name\tType\tValue");
        foreach (var entry in table)
            Console.WriteLine($"{entry.Key}\t{entry.Value.type}\t{entry.Value.value ?? "null"}");
    }
}
```

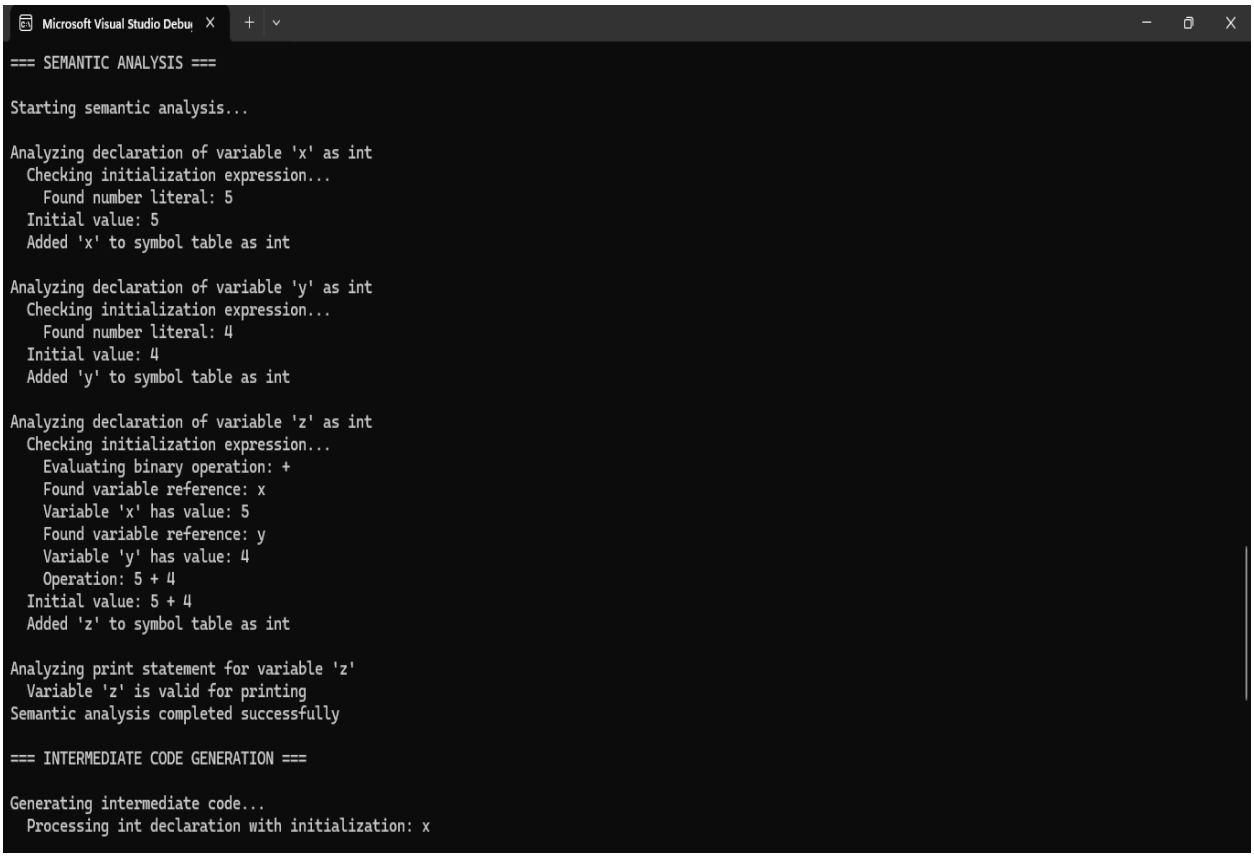
Output:

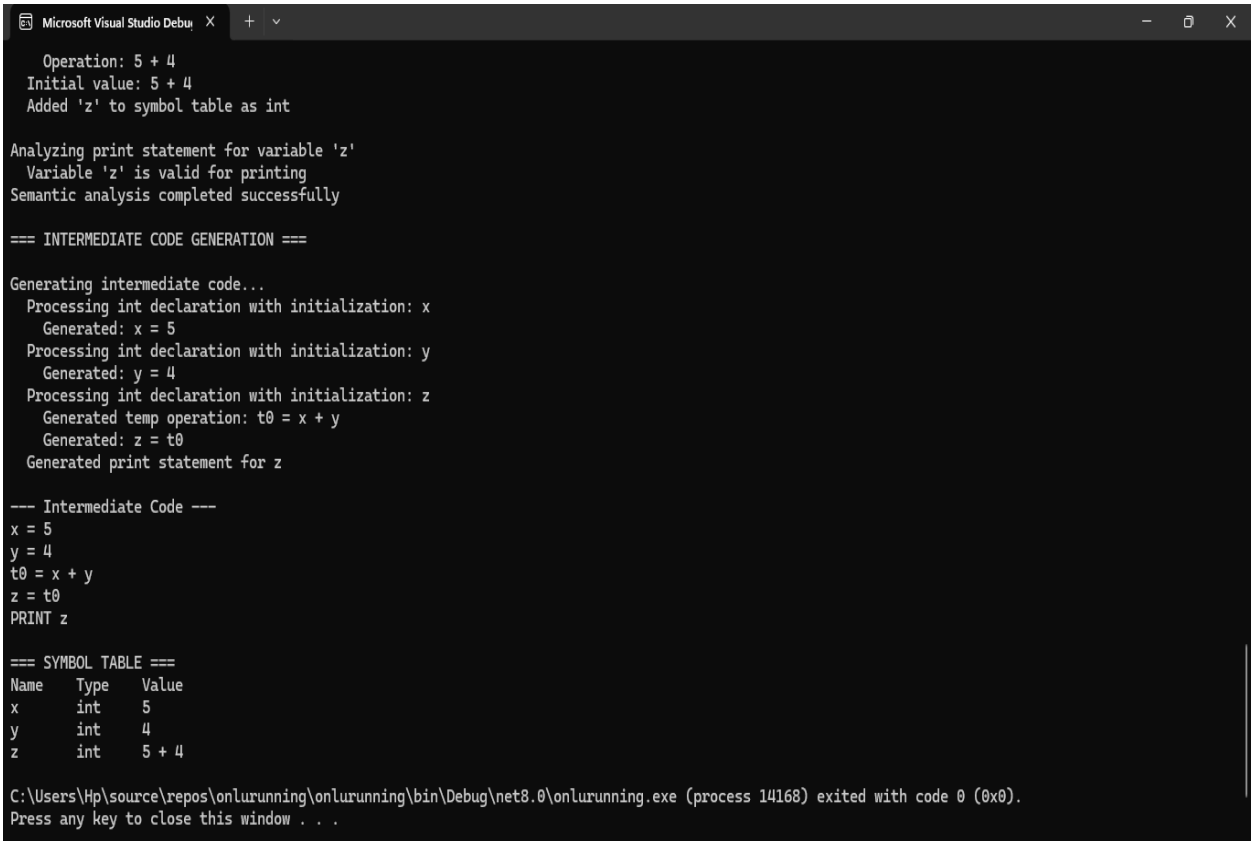

Microsoft Visual Studio Debu
Enter source code (end with an empty line):
int x = 5;
int y = 4;
int z = x + y;
print z;
=== LEXICAL ANALYSIS ===
Lexing line 1: int x = 5;
Found token: Keyword 'int'
Found token: Identifier 'x'
Found token: Assignment '='
Found token: Number '5'
Found token: Separator ';'
Lexing line 2: int y = 4;
Found token: Keyword 'int'
Found token: Identifier 'y'
Found token: Assignment '='
Found token: Number '4'
Found token: Separator ';'
Lexing line 3: int z = x + y;
Found token: Keyword 'int'
Found token: Identifier 'z'
Found token: Assignment '='
Found token: Identifier 'x'
Found token: Operator '+'
Found token: Identifier 'y'
Found token: Separator ';'
Lexing line 4: print z;
Found token: Keyword 'print'
Found token: Identifier 'z'


Microsoft Visual Studio Debu
Lexing line 4: print z;
Found token: Keyword 'print'
Found token: Identifier 'z'
Found token: Separator ';'
--- Tokens ---
[Keyword] int
[Identifier] x
[Assignment] =
[Number] 5
[Separator] ;
[Keyword] int
[Identifier] y
[Assignment] =
[Number] 4
[Separator] ;
[Keyword] int
[Identifier] z
[Assignment] =
[Identifier] x
[Operator] +
[Identifier] y
[Separator] ;
[Keyword] print
[Identifier] z
[Separator] ;
=== SYNTAX ANALYSIS ===
[Syntax] Parsing int variable declaration...
Found identifier: x
Found assignment, parsing expression...
Parsing expression...
Found number: 5


Microsoft Visual Studio Debu
=== SYNTAX ANALYSIS ===
[Syntax] Parsing int variable declaration...
Found identifier: x
Found assignment, parsing expression...
Parsing expression...
Found number: 5
End of declaration statement
[Syntax] Parsing int variable declaration...
Found identifier: y
Found assignment, parsing expression...
Parsing expression...
Found number: 4
End of declaration statement
[Syntax] Parsing int variable declaration...
Found identifier: z
Found assignment, parsing expression...
Parsing expression...
Found identifier in expression: x
Found additive operator: +
Found identifier in expression: y
End of declaration statement
[Syntax] Parsing print statement...
Found identifier to print: z
End of print statement
--- Abstract Syntax Tree ---
Program:
Declaration: int
Identifier: x
Number: 5


Microsoft Visual Studio Debu
--- Abstract Syntax Tree ---
Program:
Declaration: int
Identifier: x
Number: 5
Declaration: int
Identifier: y
Number: 4
Declaration: int
Identifier: z
BinaryOp: +
Identifier: x
Identifier: y
Print: print
Identifier: z
=== SEMANTIC ANALYSIS ===
Starting semantic analysis...
Analyzing declaration of variable 'x' as int
Checking initialization expression...
Found number literal: 5
Initial value: 5
Added 'x' to symbol table as int
Analyzing declaration of variable 'y' as int
Checking initialization expression...
Found number literal: 4
Initial value: 4
Added 'y' to symbol table as int
Analyzing declaration of variable 'z' as int
Checking initialization expression...
Evaluating binary operation: +


Microsoft Visual Studio Debu
=== SEMANTIC ANALYSIS ===
Starting semantic analysis...
Analyzing declaration of variable 'x' as int
Checking initialization expression...
Found number literal: 5
Initial value: 5
Added 'x' to symbol table as int
Analyzing declaration of variable 'y' as int
Checking initialization expression...
Found number literal: 4
Initial value: 4
Added 'y' to symbol table as int
Analyzing declaration of variable 'z' as int
Checking initialization expression...
Evaluating binary operation: +
Found variable reference: x
Variable 'x' has value: 5
Found variable reference: y
Variable 'y' has value: 4
Operation: 5 + 4
Initial value: 5 + 4
Added 'z' to symbol table as int
Analyzing print statement for variable 'z'
Variable 'z' is valid for printing
Semantic analysis completed successfully
=== INTERMEDIATE CODE GENERATION ===
Generating intermediate code...
Processing int declaration with initialization: x


Microsoft Visual Studio Debu
Operation: 5 + 4
Initial value: 5 + 4
Added 'z' to symbol table as int
Analyzing print statement for variable 'z'
Variable 'z' is valid for printing
Semantic analysis completed successfully
=== INTERMEDIATE CODE GENERATION ===
Generating intermediate code...
Processing int declaration with initialization: x
Generated: x = 5
Processing int declaration with initialization: y
Generated: y = 4
Processing int declaration with initialization: z
Generated temp operation: t0 = x + y
Generated: z = t0
Generated print statement for z
--- Intermediate Code ---
x = 5
y = 4
t0 = x + y
z = t0
PRINT z
=== SYMBOL TABLE ===
Name Type Value
x int 5
y int 4
z int 5 + 4
C:\Users\Hp\source\repos\onlurunning\onlurunning\bin\Debug\net8.0\onlurunning.exe (process 14168) exited with code 0 (0x0).
Press any key to close this window . . .