**Name:Mohammad Asim Khan  
Reg no:FA21-BCS-037  
Compiler Construction Lab Terminal**

**Question3: Optimization used in code**

**Answer:**

### ****Optimizations in the Symbol Table****

**Singleton Pattern:**

* 1. The SymbolTable is implemented as a singleton, ensuring only one instance of the symbol table exists during compilation. This saves memory and ensures consistent access to variables across the compiler stages.

**Advantage:** Reduces overhead by avoiding multiple symbol tables and ensures centralized access to variable declarations and values.

**Dictionary Lookup:**

* 1. Variable types and values are stored in dictionaries (\_variables and \_values), which allow for efficient O(1) average-time complexity for lookups, insertions, and updates.

**Advantage:** Makes symbol resolution (e.g., GetVariableType, SetVariableValue) faster compared to linear searches in lists.

**Default Value Handling:**

* 1. Default values for variables are pre-initialized using value.GetDefaultValue() when declaring variables.

**Advantage:** Simplifies variable initialization logic and reduces runtime checks for uninitialized variables.

**Symbol table code:**

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

using MiniCompiler.Interpretar.Values;

using MiniCompiler.Semantic.Types;

using Type = MiniCompiler.Semantic.Types.Type;

namespace MiniCompiler.Semantic

{

public class SymbolTable

{

public static SymbolTable Instance {

get

{

if(\_instance==null)

\_instance=new SymbolTable();

return \_instance;

} }

private static SymbolTable \_instance = null;

private Dictionary<string, Type> \_variables;

private Dictionary<string, InterpreteValue> \_values;

private SymbolTable()

{

\_variables=new Dictionary<string, Type>();

\_values=new Dictionary<string, InterpreteValue>();

}

public Type GetVariableType(string name)

{

if (!\_variables.ContainsKey(name))

throw new SemanticException("Variable "+name+" doesn't exist");

return \_variables[name];

}

public void DeclareVariable(string name,Type value)

{

if(\_variables.ContainsKey(name))

throw new SemanticException("Variable " + name + " already exist");

\_variables[name] = value;

\_values[name] = value.GetDefaultValue();

}

public InterpreteValue GetVariableValue(string name)

{

return \_values[name];

}

public void SetVariableValue(string name, InterpreteValue value)

{

\_values[name] = value;

}

}

}

### ****Optimizations in the Parser****

**Recursive Descent Parsing:**

* 1. The parser employs **recursive descent parsing**, a top-down parsing method that is straightforward and efficient for parsing grammars that don't require extensive backtracking.

**Advantage:** It simplifies code readability and allows for modular handling of grammar rules (e.g., Expr, Factor, Term).

**Token Consumption:**

* 1. The ConsumeToken() method is used to advance through tokens systematically, ensuring minimal redundant checks or re-processing of tokens.

**Advantage:** Makes parsing deterministic and avoids unnecessary computation.

**Separation of Concerns:**

* 1. Different parsing tasks are modularized into methods like Declaration, StatementList, Expr, etc.

**Advantage:** Reduces complexity and makes optimizations in individual components easier.

**Grammar Simplification via Epsilon Rules:**

* 1. Epsilon (empty) productions are handled by returning empty lists or null values in functions like IfStatementP, StatementList, and IntListP.

**Advantage:** Avoids unnecessary intermediate nodes in the abstract syntax tree (AST), making it lighter and simpler to traverse.

**Reusing Data Structures:**

* 1. The parser uses a shared \_primitivetypes dictionary for token-to-type mapping.

**Advantage:** Reduces redundant object creation and ensures consistent type handling

public enum TokenType

{

INT, FLOAT, ID, ASSIGN, PLUS, EOF

}

public class Token

{

public TokenType Type { get; set; }

public string Value { get; set; }

public Token(TokenType type, string value)

{

Type = type;

Value = value;

}

}

public class Lexer

{

private readonly string \_input;

private int \_pos = 0;

public Lexer(string input)

{

\_input = input;

}

public Token GetNextToken()

{

if (\_pos >= \_input.Length)

return new Token(TokenType.EOF, "");

char current = \_input[\_pos];

if (char.IsDigit(current))

{

\_pos++;

return new Token(TokenType.INT, current.ToString());

}

if (char.IsLetter(current))

{

\_pos++;

return new Token(TokenType.ID, current.ToString());

}

if (current == '+')

{

\_pos++;

return new Token(TokenType.PLUS, "+");

}

if (current == '=')

{

\_pos++;

return new Token(TokenType.ASSIGN, "=");

}

throw new Exception("Unknown character: " + current);

}

}

public class Parser

{

private readonly List<Token> \_tokens;

private int \_currentIndex = 0;

// Shared primitive types dictionary

private readonly Dictionary<string, string> \_primitiveTypes = new Dictionary<string, string>

{

{ "int", "INT" },

{ "float", "FLOAT" }

};

public Parser(List<Token> tokens)

{

\_tokens = tokens;

}

private Token CurrentToken => \_tokens[\_currentIndex];

private void ConsumeToken()

{

\_currentIndex++;

}

// Grammar Simplification via Epsilon Rules

private void EpsilonRule()

{

// Handle empty cases (epsilon)

Console.WriteLine("Epsilon rule triggered.");

}

// Recursive Descent Parsing for Expression

public void Expr()

{

Term();

while (CurrentToken.Type == TokenType.PLUS)

{

ConsumeToken(); // Token consumption

Term();

}

}

private void Term()

{

if (CurrentToken.Type == TokenType.INT || CurrentToken.Type == TokenType.ID)

{

Console.WriteLine($"Term: {CurrentToken.Value}");

ConsumeToken();

}

else

{

throw new Exception("Invalid term");

}

}

// Example of modular separation for grammar rules

public void Declaration()

{

if (\_primitiveTypes.ContainsKey(CurrentToken.Value))

{

Console.WriteLine($"Declaration: {\_primitiveTypes[CurrentToken.Value]}");

ConsumeToken();

}

else

{

throw new Exception("Unknown type");

}

}

}