Part 1: Foundations

Kyle Bush

# Source Code

## Main Program

using System;

using System.Reflection.Emit;

namespace KyleBushCompiler

{

    class Program

    {

        static void Main(string[] args)

        {

            Interpreter interpreter = new Interpreter();

            // Build assets for factorial algorithm and run it without Trace

            QuadTable factQuadTable = BuildQuadsForFactorial();

            SymbolTable factSymbolTable = BuildSymbolTableForFactorial();

            interpreter.InterpretQuads(factQuadTable, factSymbolTable, false);

QuadTable factQuadTable = BuildQuadsForFactorial();

            SymbolTable factSymbolTable = BuildSymbolTableForFactorial();

            interpreter.InterpretQuads(factQuadTable, factSymbolTable, true);

            // Build assets for summation algorithm and run it with interpreter

            QuadTable sumQuadTable = BuildQuadsForSummation();

            SymbolTable sumSymbolTable = BuildSymbolTableForSummation();

            interpreter.InterpretQuads(sumQuadTable, sumSymbolTable, false);

            interpreter.InterpretQuads(sumQuadTable, sumSymbolTable, true);

        }

        /// <summary>

        /// Builds and prints the hard coded Reserve Table and Quad Table to run the factorial algorithm.

        /// </summary>

        /// <returns>The Quad Table</returns>

        static QuadTable BuildQuadsForFactorial()

        {

            ReserveTable reserveTable = new ReserveTable();

            reserveTable.PrintReserveTable();

            QuadTable quadTable = new QuadTable(reserveTable);

            quadTable.AddQuad(5, 4, 0, 0);

            quadTable.AddQuad(5, 5, 0, 1);

            quadTable.AddQuad(5, 5, 0, 2);

            quadTable.AddQuad(3, 0, 2, 6);

            quadTable.AddQuad(13, 6, 0, 8);

            quadTable.AddQuad(2, 1, 2, 1);

            quadTable.AddQuad(4, 2, 5, 2);

            quadTable.AddQuad(14, 0, 0, 3);

            quadTable.AddQuad(5, 1, 0, 3);

            quadTable.AddQuad(16, 3, 0, 0);

            quadTable.AddQuad(0, 0, 0, 0);

            quadTable.PrintQuadTable();

            return quadTable;

        }

        /// <summary>

        /// Builds and prints the hard coded Reserve Table and Quad Table to run the summation algorithm.

        /// </summary>

        /// <returns>The Quad Table</returns>

        static QuadTable BuildQuadsForSummation()

        {

            ReserveTable reserveTable = new ReserveTable();

            reserveTable.PrintReserveTable();

            QuadTable quadTable = new QuadTable(reserveTable);

            quadTable.AddQuad(5, 4, 0, 0);

            quadTable.AddQuad(5, 5, 0, 1);

            quadTable.AddQuad(5, 5, 0, 2);

            quadTable.AddQuad(3, 0, 2, 6);

            quadTable.AddQuad(13, 6, 0, 8);

            quadTable.AddQuad(4, 1, 2, 1);

            quadTable.AddQuad(4, 2, 5, 2);

            quadTable.AddQuad(14, 0, 0, 3);

            quadTable.AddQuad(5, 1, 0, 3);

            quadTable.AddQuad(16, 3, 0, 0);

            quadTable.AddQuad(0, 0, 0, 0);

            quadTable.PrintQuadTable();

            return quadTable;

        }

        /// <summary>

        /// Builds and prints the hard coded Symbol Table for the factorial algorithm.

        /// </summary>

        /// <returns>The Symbol Table</returns>

        static SymbolTable BuildSymbolTableForFactorial()

        {

            SymbolTable symbolTable = new SymbolTable();

            symbolTable.AddSymbol("n", SymbolKind.Variable, 0);

            symbolTable.AddSymbol("prod", SymbolKind.Variable, 0);

            symbolTable.AddSymbol("count", SymbolKind.Variable, 0);

            symbolTable.AddSymbol("fact", SymbolKind.Variable, 0);

            symbolTable.AddSymbol("5", SymbolKind.Constant, 5);

            symbolTable.AddSymbol("1", SymbolKind.Constant, 1);

            symbolTable.AddSymbol("temp", SymbolKind.Variable, 0);

            symbolTable.PrintSymbolTable();

            return symbolTable;

        }

        /// <summary>

        /// Builds and prints the hard coded Symbol Table for the summation algorithm.

        /// </summary>

        /// <returns>The Symbol Table</returns>

        static SymbolTable BuildSymbolTableForSummation()

        {

            SymbolTable symbolTable = new SymbolTable();

            symbolTable.AddSymbol("n", SymbolKind.Variable, 0);

            symbolTable.AddSymbol("sum", SymbolKind.Variable, 0);

            symbolTable.AddSymbol("count", SymbolKind.Variable, 0);

            symbolTable.AddSymbol("summation", SymbolKind.Variable, 0);

            symbolTable.AddSymbol("5", SymbolKind.Constant, 5);

            symbolTable.AddSymbol("1", SymbolKind.Constant, 1);

            symbolTable.AddSymbol("temp", SymbolKind.Variable, 0);

            symbolTable.PrintSymbolTable();

            return symbolTable;

        }

    }

}

## Quad Table

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

namespace KyleBushCompiler

{

    /// <summary>

    /// Contains all the Quad Codes for a given application.

    /// </summary>

    public class QuadTable

    {

        private List<Quad> QuadTableData { get; set; }

        public ReserveTable ReserveTable { get; set; }

        /// <summary>

        /// Constructor which takes a Reserve Table object and initializes the Quad Table

        /// </summary>

        /// <param name="reserveTable"></param>

        public QuadTable(ReserveTable reserveTable)

        {

            ReserveTable = reserveTable;

            Initialize();

        }

        /// <summary>

        /// Create a new, empty QuadTable ready for data to be added.

        /// </summary>

        public void Initialize() //size and other parameters as needed

        {

            QuadTableData = new List<Quad>();

        }

        /// <summary>

        /// Returns the int index of the next open slot in the QuadTable.

        /// </summary>

        /// <returns>int index of the next open slot in the QuadTable</returns>

        public int NextQuad()

        {

            return QuadTableData.Count;

        }

        /// <summary>

        /// Expands the active length of the quad table by adding a new row at the NextQuad slot,

        /// with the parameters sent as the new contents, and increments the NextQuad  /// counter to the next available(empty) index.

        /// </summary>

        /// <param name="opcode"></param>

        /// <param name="op1"></param>

        /// <param name="op2"></param>

        /// <param name="op3"></param>

        public void AddQuad(int opcode, int op1, int op2, int op3)

        {

            QuadTableData.Add(new Quad(opcode, op1, op2, op3));

        }

        /// <summary>

        /// Returns the data for the opcode and three operands located at index

        /// </summary>

        /// <param name="index"></param>

        /// <returns></returns>

        public Quad GetQuad(int index)

        {

            return QuadTableData[index];

        }

        /// <summary>

        /// Changes the contents of the existing quad at index. Used only when backfilling jump addresses later, during code generation, and very important

        /// </summary>

        /// <param name="index"></param>

        /// <param name="opcode"></param>

        /// <param name="op1"></param>

        /// <param name="op2"></param>

        /// <param name="op3"></param>

        public void SetQuad(int index, int opcode, int op1, int op2, int op3)

        {

            QuadTableData[index] = new Quad(opcode, op1, op2, op3);

        }

        /// <summary>

        /// Returns the mnemonic string (‘ADD’, ‘PRINT’, etc.) associated with the ///opcode parameter.

        /// Used during interpreter ‘TRACE’ mode to print out the stored opcodes in /// readable format.

        /// Use the ReserveTable ADT to implement this.

        /// </summary>

        /// <param name="opcode"></param>

        /// <returns></returns>

        public string GetMnemonic(int opcode)

        {

            return ReserveTable.LookupCode(opcode);

        }

        /// <summary>

        /// Prints the currently used contents of the Quad table in neat tabular /// format

        /// </summary>

        public void PrintQuadTable()

        {

            Console.WriteLine("QUAD TABLE");

            Console.WriteLine("---------------------------");

            Console.WriteLine($"|{ "Opcode", 7 }|{ "Op1",5 }|{ "Op2",5 }|{ "Op3",5 }|");

            Console.WriteLine("---------------------------");

            foreach (var quad in QuadTableData)

            {

                Console.WriteLine($"|{ GetMnemonic(quad.OpCode),-7 } |{ quad.Op1,5 }|{ quad.Op2,5 }|{ quad.Op3,5 }|");

            }

            Console.WriteLine("---------------------------\n");

        }

    }

}

## Quad

using System;

using System.Collections.Generic;

using System.Text;

namespace KyleBushCompiler

{

    /// <summary>

    /// Contains the data for a single Quad Code

    /// </summary>

    public class Quad

    {

        public int OpCode { get; set; }

        public int Op1 { get; set; }

        public int Op2 { get; set; }

        public int Op3 { get; set; }

        /// <summary>

        /// Constructor which creates a new Quad Code

        /// </summary>

        /// <param name="opcode"></param>

        /// <param name="op1"></param>

        /// <param name="op2"></param>

        /// <param name="op3"></param>

        public Quad(int opcode, int op1, int op2, int op3)

        {

            OpCode = opcode;

            Op1 = op1;

            Op2 = op2;

            Op3 = op3;

        }

    }

}

## Symbol Table

using System;

using System.Collections.Generic;

using System.Linq;

using System.Security.Cryptography.X509Certificates;

using System.Text;

namespace KyleBushCompiler

{

    /// <summary>

    /// Contains all the symbols for a given application.

    /// </summary>

    public class SymbolTable

    {

        private List<Symbol> SymbolTableData { get; set; }

        /// <summary>

        /// Creates a new, empty Symbol Table.

        /// </summary>

        public SymbolTable()

        {

            SymbolTableData = new List<Symbol>();

        }

        /// <summary>

        /// Adds symbol with given kind and value to the symbol table,

/// automatically setting the correct data\_type,

        /// and returns the index where the symbol was located. If the symbol is

/// already in the table, no change or verification is made,

        /// and this just returns the index where the symbol was found.

        /// </summary>

        /// <param name="symbol">The symbol to add to the symbol table</param>

        /// <param name="kind">The kind of symbol</param>

        /// <param name="value">The value associated with the given symbol</param>

        /// <returns>The index of the added symbol in the symbol table as an integer

/// </returns>

        public int AddSymbol(string symbol, SymbolKind kind, int value)

        {

            SymbolTableData.Add(new Symbol(symbol, kind, DataType.Integer, value));

            return SymbolTableData.Count - 1;

        }

        /// <summary>

        /// Adds symbol with given kind and value to the symbol table,

/// automatically setting the correct data\_type,

        /// and returns the index where the symbol was located.

/// If the symbol is already in the table,

        /// no change or verification is made,

/// and this just returns the index where the symbol was found.

        /// </summary>

        /// <param name="symbol">The symbol to add to the symbol table</param>

        /// <param name="kind">The kind of symbol</param>

        /// <param name="value">The value associated with the given symbol</param>

        /// <returns>The index of the added symbol in the symbol table as an integer

/// </returns>

        public int AddSymbol(string symbol, SymbolKind kind, double value)

        {

            SymbolTableData.Add(new Symbol(symbol, kind, DataType.Double, value));

            return SymbolTableData.Count - 1;

        }

        /// <summary>

        /// Adds symbol with given kind and value to the symbol table,

/// automatically setting the correct data\_type,

        /// and returns the index where the symbol was located.

/// If the symbol is already in the table,

        /// no change or verification is made,

/// and this just returns the index where the symbol was found.

        /// </summary>

        /// <param name="symbol">The symbol to add to the symbol table</param>

        /// <param name="kind">The kind of symbol</param>

        /// <param name="value">The value associated with the given symbol</param>

        /// <returns>The index of the added symbol in the symbol table as an integer

/// </returns>

        public int AddSymbol(string symbol, SymbolKind kind, string value)

        {

            SymbolTableData.Add(new Symbol(symbol, kind, DataType.String, value));

            return SymbolTableData.Count - 1;

        }

        /// <summary>

        /// Returns the index where symbol is found, or -1 if not in the table

        /// </summary>

        /// <param name="symbol">The symbol to look for in the table.</param>

        /// <returns>The index of the symbol or -1 if not found</returns>

        public int LookupSymbol(string symbol)

        {

            return SymbolTableData.FindIndex(s => s.Name == symbol);

        }

        /// <summary>

        /// Return kind, data type, and value fields stored at index

        /// </summary>

        /// <param name="index">The index of the symbol to return</param>

        /// <returns></returns>

        public Symbol GetSymbol(int index)

        {

            return SymbolTableData[index];

        }

        /// <summary>

        /// Set appropriate fields at slot indicated by index

        /// </summary>

        /// <param name="index">The index of the symbol to update</param>

        /// <param name="kind">The kind of symbol</param>

        /// <param name="value">The value of the symbol</param>

        public void UpdateSymbol(int index, SymbolKind kind, int value)

        {

            SymbolTableData[index].Kind = kind;

            SymbolTableData[index].SetValue(value);

        }

        /// <summary>

        /// Set appropriate fields at slot indicated by index

        /// </summary>

        /// <param name="index">The index of the symbol to update</param>

        /// <param name="kind">The kind of symbol</param>

        /// <param name="value">The value of the symbol</param>

        public void UpdateSymbol(int index, SymbolKind kind, double value)

        {

            SymbolTableData[index].Kind = kind;

            SymbolTableData[index].SetValue(value);

        }

        /// <summary>

        /// Set appropriate fields at slot indicated by index

        /// </summary>

        /// <param name="index">The index of the symbol to update</param>

        /// <param name="kind">The kind of symbol</param>

        /// <param name="value">The value of the symbol</param>

        public void UpdateSymbol(int index, SymbolKind kind, string value)

        {

            SymbolTableData[index].Kind = kind;

            SymbolTableData[index].SetValue(value);

        }

        /// <summary>

        /// Prints the utilized rows of the symbol table in neat tabular format,

        /// showing only the value field which is active for that row

        /// </summary>

        public void PrintSymbolTable()

        {

            Console.WriteLine("SYMBOL TABLE");

            Console.WriteLine("---------------------------------------------");

            Console.WriteLine($"|{ "Name",-10 } |{ "Kind",10 }|{ "DataType",10 }|{ "Value",10 }|");

            Console.WriteLine("---------------------------------------------");

            foreach (var symbol in SymbolTableData)

            {

                Console.WriteLine($"|{ symbol.Name,-10 }|{ symbol.Kind,10 }| { symbol.DataType,10 }|{ symbol.GetValue(),10 }|");

            }

            Console.WriteLine("---------------------------------------------\n");

        }

    }

}

## Symbol

using System;

using System.Collections.Generic;

using System.Dynamic;

using System.Text;

namespace KyleBushCompiler

{

    /// <summary>

    /// Used to specify data type of a symbol

    /// </summary>

    public enum DataType

    {

        Integer,

        Double,

        String

    }

    /// <summary>

    /// Used to specify the kind of a symbol

    /// </summary>

    public enum SymbolKind

    {

        Label,

        Variable,

        Constant

    }

    public class Symbol

    {

        public string Name { get; set; }

        public SymbolKind Kind { get; set; }

        public DataType DataType { get; set; }

        private int \_intValue;

        private string \_stringValue;

        private double \_doubleValue;

        /// <summary>

        /// Contructor to initialize a Symbol containing an integer value.

        /// </summary>

        /// <param name="name"></param>

        /// <param name="kind"></param>

        /// <param name="dataType"></param>

        /// <param name="value"></param>

        public Symbol(string name, SymbolKind kind, DataType dataType, int value)

        {

            Name = name;

            Kind = kind;

            DataType = dataType;

            \_intValue = value;

        }

        /// <summary>

        /// Contructor to initialize a Symbol containing a double value.

        /// </summary>

        /// <param name="name"></param>

        /// <param name="kind"></param>

        /// <param name="dataType"></param>

        /// <param name="value"></param>

        public Symbol(string name, SymbolKind kind, DataType dataType, double value)

        {

            Name = name;

            Kind = kind;

            DataType = dataType;

            \_doubleValue = value;

        }

        /// <summary>

        /// Contructor to initialize a Symbol containing a string value.

        /// </summary>

        /// <param name="name"></param>

        /// <param name="kind"></param>

        /// <param name="dataType"></param>

        /// <param name="value"></param>

        public Symbol(string name, SymbolKind kind, DataType dataType, string value)

        {

            Name = name;

            Kind = kind;

            DataType = dataType;

            \_stringValue = value;

        }

        /// <summary>

        /// Sets a Symbol with an integer value.

        /// </summary>

        /// <param name="value"></param>

        public void SetValue(int value)

        {

            \_intValue = value;

        }

        /// <summary>

        /// Sets a Symbol with a string value.

        /// </summary>

        /// <param name="value"></param>

        public void SetValue(string value)

        {

            \_stringValue = value;

        }

        /// <summary>

        /// Sets a Symbol with a double value.

        /// </summary>

        /// <param name="value"></param>

        public void SetValue(double value)

        {

            \_doubleValue = value;

        }

        /// <summary>

        /// Checks the DataType of the Symbol and returns the appropriate value.

        /// </summary>

        /// <returns>int, string, or double depending on the DataType property.

/// </returns>

        public dynamic GetValue()

        {

            if (DataType == DataType.Integer)

                return \_intValue;

            else if (DataType == DataType.Double)

                return \_doubleValue;

            else

                return \_stringValue;

        }

    }

}

## Reserve Table

using System;

using System.Collections.Generic;

using System.Linq;

using System.Security.Cryptography.X509Certificates;

using System.Text;

namespace KyleBushCompiler

{

    /// <summary>

    /// Contains all the reserve words for a language.

    /// </summary>

    public class ReserveTable

    {

        public List<OpCode> ReserveTableData { get; set; }

        public ReserveTable()

        {

            Initialize();

        }

        /// <summary>

        /// Initializes the table with all the reserve words for the language.

        /// </summary>

        public void Initialize()

        {

            ReserveTableData = new List<OpCode>();

            Add("STOP", 0);

            Add("DIV", 1);

            Add("MUL", 2);

            Add("SUB", 3);

            Add("ADD", 4);

            Add("MOV", 5);

            Add("STI", 6);

            Add("LDI", 7);

            Add("BNZ", 8);

            Add("BNP", 9);

            Add("BNN", 10);

            Add("BZ", 11);

            Add("BP", 12);

            Add("BN", 13);

            Add("BR", 14);

            Add("BINDR", 15);

            Add("PRINT", 16);

        }

        /// <summary>

        /// Returns the index of the row where the data was place,

/// just adds to end of list.

        /// </summary>

        /// <param name="name"></param>

        /// <param name="code"></param>

        /// <returns>index of the row where the data was placed</returns>

        public int Add(string name, int code)

        {

            OpCode opCode = new OpCode(name, code);

            ReserveTableData.Add(opCode);

            return ReserveTableData.Count - 1;

        }

        /// <summary>

        /// Returns the code associated with name if name is in the table,

/// else returns -1

        /// </summary>

        /// <param name="name"></param>

        /// <returns></returns>

        public int LookupName(string name)

        {

            OpCode opCode = ReserveTableData.FirstOrDefault(x => x.Name == name);

            if (opCode == null)

            {

                return -1;

            }

            return opCode.Code;

        }

        /// <summary>

        /// Returns the associated name if code is there, else an empty string

        /// </summary>

        /// <param name="code"></param>

        /// <returns></returns>

        public string LookupCode(int code)

        {

            OpCode opCode = ReserveTableData.FirstOrDefault(x => x.Code == code);

            if (opCode == null)

            {

                return "";

            }

            return opCode.Name;

        }

        /// <summary>

        /// Searches the table for the given code to test if it is valid.

        /// </summary>

        /// <param name="code"></param>

        /// <returns>True if the code is valid, False if not.</returns>

        public bool isValidOpCode(int code)

        {

            OpCode opCode = ReserveTableData.FirstOrDefault(x => x.Code == code);

            if (opCode == null)

            {

                Console.WriteLine($"{code} is not a valid Op Code.");

                return false;

            }

            return true;

        }

        /// <summary>

        /// Prints the currently used contents of the Reserve table

/// in neat tabular format

        /// </summary>

        public void PrintReserveTable()

        {

            Console.WriteLine("RESERVE TABLE");

            Console.WriteLine("---------------");

            Console.WriteLine($"|{ "Name", -7 }|{ "Code", 5 }|");

            Console.WriteLine("---------------");

            foreach (var code in ReserveTableData)

            {

                Console.WriteLine($"|{ code.Name, -7 }|{ code.Code, 5 }|");

            }

            Console.WriteLine("---------------\n");

        }

    }

}

## OpCode

using System;

using System.Collections.Generic;

using System.Text;

namespace KyleBushCompiler

{

    /// <summary>

    /// Contains the string and integer representations of an OpCode.

    /// </summary>

    public class OpCode

    {

        public string Name { get; set; }

        public int Code { get; set; }

        /// <summary>

        /// Creates a new OpCode object.

        /// </summary>

        /// <param name="name"></param>

        /// <param name="code"></param>

        public OpCode(string name, int code)

        {

            Name = name;

            Code = code;

        }

    }

}

## Interpretter

using System;

using System.Collections.Generic;

using System.IO;

using System.Text;

namespace KyleBushCompiler

{

    /// <summary>

    /// Interprets Quad Codes and Symbols to run a program.

    /// </summary>

    public class Interpreter

    {

        private const int STOP = 0;

        private const int DIV = 1;

        private const int MUL = 2;

        private const int SUB = 3;

        private const int ADD = 4;

        private const int MOV = 5;

        private const int STI = 6;

        private const int LDI = 7;

        private const int BNZ = 8;

        private const int BNP = 9;

        private const int BNN = 10;

        private const int BZ = 11;

        private const int BP = 12;

        private const int BN = 13;

        private const int BR = 14;

        private const int BINDR = 15;

        private const int PRINT = 16;

        private int ProgramCounter { get; set; }

        private Quad CurrentQuad { get; set; }

        public QuadTable QuadTable { get; set; }

        /// <summary>

        /// Prints the relevant Quad Code information when

/// the interpreter is run in Trace Mode

        /// </summary>

        /// <param name="opcode"></param>

        /// <param name="op1"></param>

        /// <param name="op2"></param>

        /// <param name="op3"></param>

        public void PrintTrace(int opcode, int op1, int op2, int op3)

        {

            Console.WriteLine($"PC = {ProgramCounter}: {QuadTable.GetMnemonic(opcode)} {op1}, {op2}, {op3}");

        }

        /// <summary>

        /// Prints the relevant Quad Code information when

/// the interpreter is run in Trace Mode

        /// </summary>

        /// <param name="opcode"></param>

        /// <param name="op1"></param>

        /// <param name="op2"></param>

        /// <param name="op3"></param>

        public void PrintTrace(int opcode, int op1, int op2)

        {

            Console.WriteLine($"PC = {ProgramCounter}:

{QuadTable.GetMnemonic(opcode)} {op1}, {op2}");

        }

        /// <summary>

        /// Prints the relevant Quad Code information when

/// the interpreter is run in Trace Mode

        /// </summary>

        /// <param name="opcode"></param>

        /// <param name="op1"></param>

        /// <param name="op2"></param>

        /// <param name="op3"></param>

        public void PrintTrace(int opcode, int op)

        {

            Console.WriteLine($"PC = {ProgramCounter}:

{QuadTable.GetMnemonic(opcode)} {op}");

        }

        /// <summary>

        /// Prints the relevant Quad Code information when

/// the interpreter is run in Trace Mode

        /// </summary>

        /// <param name="opcode"></param>

        /// <param name="op1"></param>

        /// <param name="op2"></param>

        /// <param name="op3"></param>

        public void PrintTrace(int opcode)

        {

            Console.WriteLine($"PC = {ProgramCounter}:

{QuadTable.GetMnemonic(opcode)}");

        }

        /// <summary>

        /// Runs the program using the data from the given

/// Quad Table and Symbol Table.

        /// Trace mode will print each quad code that the interpretter executes.

        /// </summary>

        /// <param name="quadTable">Quad Table containing all the

/// necessary Quad Codes</param>

        /// <param name="symbolTable">Symbol Table containing all the

/// necessary Symbols</param>

        /// <param name="TraceOn">Toggles Trace Mode on and off</param>

        public void InterpretQuads(QuadTable quadTable,

SymbolTable symbolTable, bool TraceOn = false)

        {

            QuadTable = quadTable;

            ProgramCounter = 0;

            while (ProgramCounter < QuadTable.NextQuad())

            {

                CurrentQuad = QuadTable.GetQuad(ProgramCounter);

                if (QuadTable.ReserveTable.isValidOpCode(CurrentQuad.OpCode))

                {

                    try

                    {

                        switch (CurrentQuad.OpCode)

                        {

                            // STOP

                            // Terminate program

                            case STOP:

                                if (TraceOn)

                                {

                                    PrintTrace(CurrentQuad.OpCode);

                                }

                                ProgramCounter = QuadTable.NextQuad();

                                break;

                            // DIV

                            // Compute op1 / op2, place result into op3

                            case DIV:

                                if (TraceOn)

                                {

                                    PrintTrace(CurrentQuad.OpCode,

CurrentQuad.Op1, CurrentQuad.Op2,

CurrentQuad.Op3);

                                }

                                symbolTable.UpdateSymbol(CurrentQuad.Op3, SymbolKind.Variable,

                                    (symbolTable.GetSymbol(CurrentQuad.Op1).GetValue() / symbolTable.GetSymbol(CurrentQuad.Op2).GetValue()));

                                ProgramCounter++;

                                break;

                            // MUL

                            // Compute op1 \* op2, place result into op3

                            case MUL:

                                if (TraceOn)

                                {

                                    PrintTrace(CurrentQuad.OpCode, CurrentQuad.Op1, CurrentQuad.Op2, CurrentQuad.Op3);

                                }

                                symbolTable.UpdateSymbol(CurrentQuad.Op3, SymbolKind.Variable,

                                    (symbolTable.GetSymbol(CurrentQuad.Op1).GetValue() \* symbolTable.GetSymbol(CurrentQuad.Op2).GetValue()));

                                ProgramCounter++;

                                break;

                            // SUB

                            // Compute op1 - op2, place result into op3

                            case SUB:

                                if (TraceOn)

                                {

                                    PrintTrace(CurrentQuad.OpCode, CurrentQuad.Op1, CurrentQuad.Op2, CurrentQuad.Op3);

                                }

                                symbolTable.UpdateSymbol(CurrentQuad.Op3, SymbolKind.Variable,

                                    (symbolTable.GetSymbol(CurrentQuad.Op1).GetValue() - symbolTable.GetSymbol(CurrentQuad.Op2).GetValue()));

                                ProgramCounter++;

                                break;

                            // ADD

                            // Compute op1 + op2, place result into op3

                            case ADD:

                                if (TraceOn)

                                {

                                    PrintTrace(CurrentQuad.OpCode, CurrentQuad.Op1, CurrentQuad.Op2, CurrentQuad.Op3);

                                }

                                symbolTable.UpdateSymbol(CurrentQuad.Op3, SymbolKind.Variable,

                                    (symbolTable.GetSymbol(CurrentQuad.Op1).GetValue() + symbolTable.GetSymbol(CurrentQuad.Op2).GetValue()));

                                ProgramCounter++;

                                break;

                            // MOV

                            // Assign the value in op1 into op3 (op2 is ignored here)

                            case MOV:

                                if (TraceOn)

                                {

                                    PrintTrace(CurrentQuad.OpCode, CurrentQuad.Op1, CurrentQuad.Op3);

                                }

                                symbolTable.UpdateSymbol(CurrentQuad.Op3, SymbolKind.Variable, symbolTable.GetSymbol(CurrentQuad.Op1).GetValue());

                                ProgramCounter++;

                                break;

                            // STI

                            // Store indexed - Assign the value in op1 into op2 + offset op3

                            case STI:

                                if (TraceOn)

                                {

                                    PrintTrace(CurrentQuad.OpCode, CurrentQuad.Op1, CurrentQuad.Op2, CurrentQuad.Op3);

                                }

                                symbolTable.UpdateSymbol((CurrentQuad.Op2 + CurrentQuad.Op3), SymbolKind.Variable, symbolTable.GetSymbol(CurrentQuad.Op1).GetValue());

                                ProgramCounter++;

                                break;

                            // LDI

                            // Load indexed- Assign the value in op1 + offset op2, into op3

                            case LDI:

                                if (TraceOn)

                                {

                                    PrintTrace(CurrentQuad.OpCode, CurrentQuad.Op1, CurrentQuad.Op2, CurrentQuad.Op3);

                                }

                                symbolTable.UpdateSymbol(CurrentQuad.Op3, SymbolKind.Variable, symbolTable.GetSymbol(CurrentQuad.Op1 + CurrentQuad.Op2).GetValue());

                                ProgramCounter++;

                                break;

                            // BNZ

                            // Branch Not Zero; if op1 value <> 0, set program counter to op3

                            case BNZ:

                                if (TraceOn)

                                {

                                    PrintTrace(CurrentQuad.OpCode, CurrentQuad.Op3);

                                }

                                if (symbolTable.GetSymbol(CurrentQuad.Op1).GetValue() != 0)

                                {

                                    ProgramCounter = CurrentQuad.Op3;

                                }

                                else

                                {

                                    ProgramCounter++;

                                }

                                break;

                            // BNP

                            // Branch Not Positive; if op1 value <= 0, set program counter to op3

                            case BNP:

                                if (TraceOn)

                                {

                                    PrintTrace(CurrentQuad.OpCode, CurrentQuad.Op3);

                                }

                                if (symbolTable.GetSymbol(CurrentQuad.Op1).GetValue() <= 0)

                                {

                                    ProgramCounter = CurrentQuad.Op3;

                                }

                                else

                                {

                                    ProgramCounter++;

                                }

                                break;

                            // BNN

                            // Branch Not Negative; if op1 value >= 0, set program counter to op3

                            case BNN:

                                if (TraceOn)

                                {

                                    PrintTrace(CurrentQuad.OpCode, CurrentQuad.Op3);

                                }

                                if (symbolTable.GetSymbol(CurrentQuad.Op1).GetValue() >= 0)

                                {

                                    ProgramCounter = CurrentQuad.Op3;

                                }

                                else

                                {

                                    ProgramCounter++;

                                }

                                break;

                            // BZ

                            // Branch Zero; if op1 value = 0, set program counter to op3

                            case BZ:

                                if (TraceOn)

                                {

                                    PrintTrace(CurrentQuad.OpCode, CurrentQuad.Op3);

                                }

                                if (symbolTable.GetSymbol(CurrentQuad.Op1).GetValue() == 0)

                                {

                                    ProgramCounter = CurrentQuad.Op3;

                                }

                                else

                                {

                                    ProgramCounter++;

                                }

                                break;

                            // BP

                            // Branch Positive; if op1 value > 0, set program counter to op3

                            case BP:

                                if (TraceOn)

                                {

                                    PrintTrace(CurrentQuad.OpCode, CurrentQuad.Op3);

                                }

                                if (symbolTable.GetSymbol(CurrentQuad.Op1).GetValue() > 0)

                                {

                                    ProgramCounter = CurrentQuad.Op3;

                                }

                                else

                                {

                                    ProgramCounter++;

                                }

                                break;

                            // BN

                            // Branch Negative; if op1 value < 0, set program counter to op3

                            case BN:

                                if (TraceOn)

                                {

                                    PrintTrace(CurrentQuad.OpCode, CurrentQuad.Op3);

                                }

                                if (symbolTable.GetSymbol(CurrentQuad.Op1).GetValue() < 0)

                                {

                                    ProgramCounter = CurrentQuad.Op3;

                                }

                                else

                                {

                                    ProgramCounter++;

                                }

                                break;

                            // BR

                            // Branch (unconditional); set program counter to op3

                            case BR:

                                if (TraceOn)

                                {

                                    PrintTrace(CurrentQuad.OpCode, CurrentQuad.Op3);

                                }

                                ProgramCounter = CurrentQuad.Op3;

                                break;

                            // BINDR

                            // Branch (unconditional); set program counter to op3 value contents (indirect)

                            case BINDR:

                                if (TraceOn)

                                {

                                    PrintTrace(CurrentQuad.OpCode, symbolTable.GetSymbol(CurrentQuad.Op3).GetValue());

                                }

                                ProgramCounter = symbolTable.GetSymbol(CurrentQuad.Op3).GetValue();

                                break;

                            // PRINT

                            // Write symbol table name and value of op 1

                            case PRINT:

                                if (TraceOn)

                                {

                                    PrintTrace(CurrentQuad.OpCode, CurrentQuad.Op1);

                                }

                                Console.WriteLine($"{ symbolTable.GetSymbol(CurrentQuad.Op1).Name} = {symbolTable.GetSymbol(CurrentQuad.Op1).GetValue()}");

                                ProgramCounter++;

                                break;

                            default:

                                Console.WriteLine($"Invalid Opcode {CurrentQuad.OpCode}");

                                break;

                        }

                    }

                    // Catches any exception, prints the appropriate error message, and stops running the current program.

                    catch (Exception e)

                    {

                        Console.WriteLine("FATAL ERROR: " + e.Message + "\n");

                        ProgramCounter = QuadTable.NextQuad();

                    }

                }

            }

        }

    }

}

# Program Output

## Summation Without Trace

RESERVE TABLE

---------------

|Name | Code|

---------------

|STOP | 0|

|DIV | 1|

|MUL | 2|

|SUB | 3|

|ADD | 4|

|MOV | 5|

|STI | 6|

|LDI | 7|

|BNZ | 8|

|BNP | 9|

|BNN | 10|

|BZ | 11|

|BP | 12|

|BN | 13|

|BR | 14|

|BINDR | 15|

|PRINT | 16|

---------------

QUAD TABLE

---------------------------

|Opcode | Op1| Op2| Op3|

---------------------------

|MOV | 4| 0| 0|

|MOV | 5| 0| 1|

|MOV | 5| 0| 2|

|SUB | 0| 2| 6|

|BN | 6| 0| 8|

|ADD | 1| 2| 1|

|ADD | 2| 5| 2|

|BR | 0| 0| 3|

|MOV | 1| 0| 3|

|PRINT | 3| 0| 0|

|STOP | 0| 0| 0|

---------------------------

SYMBOL TABLE

---------------------------------------------

|Name | Kind| DataType| Value|

---------------------------------------------

|n | Variable| Integer| 0|

|sum | Variable| Integer| 0|

|count | Variable| Integer| 0|

|summation | Variable| Integer| 0|

|5 | Constant| Integer| 5|

|1 | Constant| Integer| 1|

|temp | Variable| Integer| 0|

---------------------------------------------

summation = 16

## Factorial Without Trace

RESERVE TABLE

---------------

|Name | Code|

---------------

|STOP | 0|

|DIV | 1|

|MUL | 2|

|SUB | 3|

|ADD | 4|

|MOV | 5|

|STI | 6|

|LDI | 7|

|BNZ | 8|

|BNP | 9|

|BNN | 10|

|BZ | 11|

|BP | 12|

|BN | 13|

|BR | 14|

|BINDR | 15|

|PRINT | 16|

---------------

QUAD TABLE

---------------------------

|Opcode | Op1| Op2| Op3|

---------------------------

|MOV | 4| 0| 0|

|MOV | 5| 0| 1|

|MOV | 5| 0| 2|

|SUB | 0| 2| 6|

|BN | 6| 0| 8|

|MUL | 1| 2| 1|

|ADD | 2| 5| 2|

|BR | 0| 0| 3|

|MOV | 1| 0| 3|

|PRINT | 3| 0| 0|

|STOP | 0| 0| 0|

---------------------------

SYMBOL TABLE

---------------------------------------------

|Name | Kind| DataType| Value|

---------------------------------------------

|n | Variable| Integer| 0|

|prod | Variable| Integer| 0|

|count | Variable| Integer| 0|

|fact | Variable| Integer| 0|

|5 | Constant| Integer| 5|

|1 | Constant| Integer| 1|

|temp | Variable| Integer| 0|

---------------------------------------------

fact = 120

## Summation With Trace

RESERVE TABLE

---------------

|Name | Code|

---------------

|STOP | 0|

|DIV | 1|

|MUL | 2|

|SUB | 3|

|ADD | 4|

|MOV | 5|

|STI | 6|

|LDI | 7|

|BNZ | 8|

|BNP | 9|

|BNN | 10|

|BZ | 11|

|BP | 12|

|BN | 13|

|BR | 14|

|BINDR | 15|

|PRINT | 16|

---------------

QUAD TABLE

---------------------------

|Opcode | Op1| Op2| Op3|

---------------------------

|MOV | 4| 0| 0|

|MOV | 5| 0| 1|

|MOV | 5| 0| 2|

|SUB | 0| 2| 6|

|BN | 6| 0| 8|

|ADD | 1| 2| 1|

|ADD | 2| 5| 2|

|BR | 0| 0| 3|

|MOV | 1| 0| 3|

|PRINT | 3| 0| 0|

|STOP | 0| 0| 0|

---------------------------

SYMBOL TABLE

---------------------------------------------

|Name | Kind| DataType| Value|

---------------------------------------------

|n | Variable| Integer| 0|

|sum | Variable| Integer| 0|

|count | Variable| Integer| 0|

|summation | Variable| Integer| 0|

|5 | Constant| Integer| 5|

|1 | Constant| Integer| 1|

|temp | Variable| Integer| 0|

---------------------------------------------

PC = 0: MOV 4, 0

PC = 1: MOV 5, 1

PC = 2: MOV 5, 2

PC = 3: SUB 0, 2, 6

PC = 4: BN 8

PC = 5: ADD 1, 2, 1

PC = 6: ADD 2, 5, 2

PC = 7: BR 3

PC = 3: SUB 0, 2, 6

PC = 4: BN 8

PC = 5: ADD 1, 2, 1

PC = 6: ADD 2, 5, 2

PC = 7: BR 3

PC = 3: SUB 0, 2, 6

PC = 4: BN 8

PC = 5: ADD 1, 2, 1

PC = 6: ADD 2, 5, 2

PC = 7: BR 3

PC = 3: SUB 0, 2, 6

PC = 4: BN 8

PC = 5: ADD 1, 2, 1

PC = 6: ADD 2, 5, 2

PC = 7: BR 3

PC = 3: SUB 0, 2, 6

PC = 4: BN 8

PC = 5: ADD 1, 2, 1

PC = 6: ADD 2, 5, 2

PC = 7: BR 3

PC = 3: SUB 0, 2, 6

PC = 4: BN 8

PC = 8: MOV 1, 3

PC = 9: PRINT 3

summation = 16

PC = 10: STOP

## Factorial With Trace

RESERVE TABLE

---------------

|Name | Code|

---------------

|STOP | 0|

|DIV | 1|

|MUL | 2|

|SUB | 3|

|ADD | 4|

|MOV | 5|

|STI | 6|

|LDI | 7|

|BNZ | 8|

|BNP | 9|

|BNN | 10|

|BZ | 11|

|BP | 12|

|BN | 13|

|BR | 14|

|BINDR | 15|

|PRINT | 16|

---------------

QUAD TABLE

---------------------------

|Opcode | Op1| Op2| Op3|

---------------------------

|MOV | 4| 0| 0|

|MOV | 5| 0| 1|

|MOV | 5| 0| 2|

|SUB | 0| 2| 6|

|BN | 6| 0| 8|

|MUL | 1| 2| 1|

|ADD | 2| 5| 2|

|BR | 0| 0| 3|

|MOV | 1| 0| 3|

|PRINT | 3| 0| 0|

|STOP | 0| 0| 0|

---------------------------

SYMBOL TABLE

---------------------------------------------

|Name | Kind| DataType| Value|

---------------------------------------------

|n | Variable| Integer| 0|

|prod | Variable| Integer| 0|

|count | Variable| Integer| 0|

|fact | Variable| Integer| 0|

|5 | Constant| Integer| 5|

|1 | Constant| Integer| 1|

|temp | Variable| Integer| 0|

---------------------------------------------

PC = 0: MOV 4, 0

PC = 1: MOV 5, 1

PC = 2: MOV 5, 2

PC = 3: SUB 0, 2, 6

PC = 4: BN 8

PC = 5: MUL 1, 2, 1

PC = 6: ADD 2, 5, 2

PC = 7: BR 3

PC = 3: SUB 0, 2, 6

PC = 4: BN 8

PC = 5: MUL 1, 2, 1

PC = 6: ADD 2, 5, 2

PC = 7: BR 3

PC = 3: SUB 0, 2, 6

PC = 4: BN 8

PC = 5: MUL 1, 2, 1

PC = 6: ADD 2, 5, 2

PC = 7: BR 3

PC = 3: SUB 0, 2, 6

PC = 4: BN 8

PC = 5: MUL 1, 2, 1

PC = 6: ADD 2, 5, 2

PC = 7: BR 3

PC = 3: SUB 0, 2, 6

PC = 4: BN 8

PC = 5: MUL 1, 2, 1

PC = 6: ADD 2, 5, 2

PC = 7: BR 3

PC = 3: SUB 0, 2, 6

PC = 4: BN 8

PC = 8: MOV 1, 3

PC = 9: PRINT 3

fact = 120

PC = 10: STOP