# Turing Machine Simulation in C#

**Hideous Humpback Freak**
21 Dec 2015 CPOL

Turing Machine Simulation in C#

## Introduction

Alan Turing's idea of building a universal computing machine was truly revolutionary and it changed the face of the world. One extraordinarily simple machine capable of carrying out any computation seems impossible. This blog post is a laconic look at the history and operation of the Universal Turing Machine. The goal is to whet the appetite of the uninitiated and inspire them to seek out a deeper understanding of computation.

## History

In 1936, Alan Turing published his famous "On Computable Numbers" paper. It was not his intention to invent a computer. In fact, the machine outlined in the paper was not initially meant to be implemented mechanically. The Automatic Computing Machine (now known as Universal Turing Machine), as he termed it, was nothing more than a mathematic concept. Turing did all of his computation with a fountain pen and paper.

Turing's true ambition was to solve the famous "Entscheidungsproblem". In a nutshell, he used the concept of an automatic computing machine to prove that there are some sequences of numbers that are impossible to compute. His proof is closely tied to the notorious Halting Problem.

In Turing's time, the word computer had a very different meaning. Computer was a job description. Mathematicians painstakingly wrote out precise directions for computers. This way, a person with little background in arithmetic could perform computation without really understanding the overall implications of what they were doing. These instructions are what inspired Turing and they are reflected in the implementation of his automatic computing machine.

What's most amazing about Turing's contemplation of the automatic computing machine is that more than eighty years later, we've yet to come up with a more powerful computational model. We have more efficient means of computation, but none of them can solve any problems that the original machine could not. Extraordinary to say the least!

## Universal Turing Machine

The Universal Turing Machine is elementary. If you've ever built a state machine, this will all start to look very familiar. The machine only has 4 components:

1. Paper Tape
2. Printer
3. Scanner
4. Transition Table

The paper tape is a theoretically infinite piece of paper sectioned off into squares. Each square is meant to hold a single character. The printer and scanner are mounted together on a head that move as a unit. The printer can print a single character to the tape and the scanner can read a single character from the tape. The head can only move one square of tape at a time. The real magic

happens in the transition table, or "table of states of mind" as Turing called them (remember Turing's idea of a computer was a human). The table is a list of instructions in the form of: *If the machine is in state A and the scanner reads character B, print character C, move the head in D direction and place the machine in state E.* ABCDE are all variables.

In order to perform a computation, information goes on the tape, the machine gets an initial state, the transition table is loaded, and it's "off to the races" following the instructions in the transition table. Once the machine reaches its' final state, the tape will hold the computed answer. That is all that is required to perform any computation. Such exquisite simplicity!

# Programmability

Turing's was not the first computation machine. In fact, many had come before him. However, his machine had something truly exceptional: programmability. A physical machine could conceivably contain a single physical transition table. Turing created a physical transition table that could read any number of logical transition tables from the tape. This was truly the "Hello World" of computer programming.

Turing used something he coined "Standard Description" to create an initial transition table capable of reading any other transition table from the tape. Many consider this to be the world's first programming language. Concepts extracted from his work such as symbol tables and m-functions are still in use today.

The notion of a machine that can emulate any other machine is known as Turing Completeness. The ramifications of this are seemingly endless. The whole of the computer industry is built on this concept. The sheer power and simplicity are mind boggling.

# Simulation

Now comes the fun part! Let's build a Turing machine simulation in C#. Please take note that this is only a thought experiment to demonstrate the basic structure of a Turing machine. Some of the more convoluted details such as e-tape/f-tape, encoding, and reading instructions from the tape are left out for the sake of brevity. That being said, with an infinite amount of time it's entirely possible to compute any possible computation using this simulation and a proper initial transition table.

I'm only giving a short synopsis of the code below. Everything is written in a somewhat functional style with no mutation. I'm sure this could be up for debate, but if you choose to debate it you're missing the point of the post... If you have any questions or comments, feel free to comment or contact me directly. I'd love to speak with you about it. The full code is available on github: https://github.com/dalealleshouse/TuringMachine

First, we have a Head that consists of the tape, printer, and scanner. The tape is just an `IEnumerable` of characters that automatically grows as needed (fun fact: Turing was also instrumental in the idea of recursive enumeration). Each character in the `IEnumerable` represents a single square on the paper tape. The class has methods that read from the tape (scanner), write to the tape (printer), and move the head position. One special thing to note is the override of the `ToString` method which will return the data on the tape in a human readable form.

```
using System;
using System.Collections.Generic;
using System.Linq;

namespace TuringMachine
{
    public class Head
    {
        public const char Blank = '_';

        public Head(IEnumerable<char> tape, int headPosition)
        {
            if (tape == null) throw new ArgumentNullException(nameof(tape));

            var safeData = tape as char[] ?? tape.ToArray();
            if (headPosition > safeData.Count() - 1 || headPosition < 0)
                throw new IndexOutOfRangeException("Invalid head position");

            Tape = safeData;
            HeadPosition = headPosition;
        }

        public IEnumerable<char> Tape { get; }
```

```
        public int HeadPosition { get; }

        public Head Write(char head) =>
        new Head(new List<char>(Tape) { [HeadPosition] = head }, HeadPosition);

        public Head MoveLeft() => HeadPosition == 0
            ? new Head(new[] { Blank }.Concat(Tape), 0)
            : new Head(Tape, HeadPosition - 1);

        public Head MoveRight() => HeadPosition == Tape.Count() - 1
            ? new Head(Tape.Concat(new[] { Blank }), HeadPosition + 1)
            : new Head(Tape, HeadPosition + 1);

        public Head Move(HeadDirection direction)
        {
            switch (direction)
            {
                case HeadDirection.Left:
                    return MoveLeft();
                case HeadDirection.NoMove:
                    return this;
                case HeadDirection.Right:
                    return MoveRight();
                default:
                    throw new ArgumentOutOfRangeException(nameof(direction), direction, null);
            }
        }

        public char Read() => Tape.ElementAt(HeadPosition);

        public override string ToString() => $@"Tape:
        {Tape.Select(GetChar).Aggregate((agg, next) => agg + next)}";

        private string GetChar(char c, int index) =>
        index == HeadPosition ? $"({c})" : c.ToString();
    }
}
```

Next, let's turn our attention to the transition table. Below is a POCO to hold transition data. A transition table is an `IEnumerable` of transition objects. `State` is an integer value and `HeadPostion` is an `Enum` composed of Left, No Move, and Right.

```
namespace TuringMachine
{
    public class Transition
    {
        public Transition(int initialState, char read, char write, HeadDirection headDirection,
int nextState)
        {
            InitialState = initialState;
            Read = read;
            Write = write;
            HeadDirection = headDirection;
            NextState = nextState;
        }

        public int InitialState { get; }

        public char Read { get; }

        public char Write { get; }

        public HeadDirection HeadDirection { get; }

        public int NextState { get; }
    }
}
```

The last piece of the puzzle is the machine itself. A machine is initialized with a state, a Head object, and a transition table. The interesting part of this class is the step method. For our purposes, I've defined any state less than 0 to denote a special stop state (error or halt). If the machine is in a stop state, it will no longer process transitions. If the machine is unable to find a suitable transition, it goes into an error state (fun fact: the fact that this machine can go into an error state makes it an undeterministic machine as opposed to a deterministic machine). If it finds a suitable transition, it applies the defined data and returns a reconfigured machine.

```
using System;
using System.Collections.Generic;
using System.Linq;

namespace TuringMachine
{
    public class Machine
    {
        public Machine(int state, Head head, IEnumerable<transition> transitionTable)
        {
            if (head == null) throw new ArgumentNullException(nameof(head));
            if (transitionTable == null) throw new
ArgumentNullException(nameof(transitionTable));

            State = state;
            Head = head;
            TransitionTable = transitionTable;
        }

        public int State { get; }

        public Head Head { get; }

        public IEnumerable<transition> TransitionTable { get; }

        public Machine Step()
        {
            if (State < 0) return this;

            return
                TransitionTable
                    .Where(t => t.InitialState == State && t.Read == Head.Read())
                    .DefaultIfEmpty(new Transition(0, Head.Blank, Head.Read(),
HeadDirection.NoMove,
                        TuringMachine.State.Error))
                    .Select(
                        t => new Machine(t.NextState,
Head.Write(t.Write).Move(t.HeadDirection), TransitionTable))
                    .First();
        }

        public Machine Run()
        {
            var m = this;

            while (m.State >= 0)
                m = m.Step();

            return m;
        }
    }
}
```

That's all we need! Let's start off easy by declaring a transition table that will add two positive numbers. We'll represent the numbers on the tape by a series of "1" characters separated by a blank. Our transition table is below.

```
using System.Collections.Generic;
using System.Resources;

namespace TuringMachine
```

```
{
    public static class TransitionTableGenerator
    {
        public static IEnumerable<transition> Addition() => new[]
        {
            new Transition(0, Tape.Blank, Tape.Blank, HeadDirection.Right, 0),
            new Transition(0, '1', '1', HeadDirection.Right, 1),
            new Transition(1, Tape.Blank, '1', HeadDirection.Right, 2),
            new Transition(1, '1', '1', HeadDirection.Right, 1),
            new Transition(2, Tape.Blank, Tape.Blank, HeadDirection.Left, 3),
            new Transition(2, '1', '1', HeadDirection.Right, 2),
            new Transition(3, Tape.Blank, Tape.Blank, HeadDirection.Left, 3),
            new Transition(3, '1', Tape.Blank, HeadDirection.Left, 4),
            new Transition(4, Tape.Blank, Tape.Blank, HeadDirection.NoMove, State.Halt),
            new Transition(4, '1', '1', HeadDirection.Left, 4)
        };
    }
}
```

The test below shows the machine in action adding 3 and 2. I highly recommend stepping through the code in order to really grok all of this. The answer is returned on the tape as a series of 5 "1" characters.

```
[TestMethod]
public void AddTwoNumbers()
{
    const string expected = "Head: (_)11111__";
    var sut = new TuringMachine.Machine(
        0,
        new TuringMachine.Head(new[] { '1', '1', '1', TuringMachine.Head.Blank, '1', '1' }, 0),
        TransitionTableGenerator.Addition());

    var result = sut.Run();
    Assert.AreEqual(expected, result.Head.ToString());
}
```

OK, that's not exactly an impressive result. Let's see if we can do something slightly more complicated. Using the same input, let's multiply the two numbers. All we need is a different transition table.

```
using System.Collections.Generic;
using System.Resources;

namespace TuringMachine
{
    public static class TransitionTableGenerator
    {
        public static IEnumerable<transition> Multiplication() => new[]
        {
            new Transition(0, Tape.Blank, Tape.Blank, HeadDirection.Right, 1),
            new Transition(0, '1', Tape.Blank, HeadDirection.Right, 2),
            new Transition(1, Tape.Blank, Tape.Blank, HeadDirection.Right, 14),
            new Transition(1, '1', Tape.Blank, HeadDirection.Right, 2),
            new Transition(2, Tape.Blank, Tape.Blank, HeadDirection.Right, 3),
            new Transition(2, '1', '1', HeadDirection.Right, 2),
            new Transition(3, Tape.Blank, Tape.Blank, HeadDirection.Left, 15),
            new Transition(3, '1', Tape.Blank, HeadDirection.Right, 4),
            new Transition(4, Tape.Blank, Tape.Blank, HeadDirection.Right, 5),
            new Transition(4, '1', '1', HeadDirection.Right, 4),
            new Transition(5, Tape.Blank, '1', HeadDirection.Left, 6),
            new Transition(5, '1', '1', HeadDirection.Right, 5),
            new Transition(6, Tape.Blank, Tape.Blank, HeadDirection.Left, 7),
            new Transition(6, '1', '1', HeadDirection.Left, 6),
            new Transition(7, Tape.Blank, '1', HeadDirection.Left, 9),
            new Transition(7, '1', '1', HeadDirection.Left, 8),
            new Transition(8, Tape.Blank, '1', HeadDirection.Right, 3),
            new Transition(8, '1', '1', HeadDirection.Left, 8),
            new Transition(9, Tape.Blank, Tape.Blank, HeadDirection.Left, 10),
            new Transition(9, '1', '1', HeadDirection.Left, 9),
```

```
            new Transition(10, Tape.Blank, Tape.Blank, HeadDirection.Right, 12),
            new Transition(10, '1', '1', HeadDirection.Left, 11),
            new Transition(11, Tape.Blank, Tape.Blank, HeadDirection.Right, 0),
            new Transition(11, '1', '1', HeadDirection.Left, 11),
            new Transition(12, Tape.Blank, Tape.Blank, HeadDirection.Right, 12),
            new Transition(12, '1', Tape.Blank, HeadDirection.Right, 13),
            new Transition(13, Tape.Blank, Tape.Blank, HeadDirection.NoMove, State.Halt),
            new Transition(13, '1', Tape.Blank, HeadDirection.Right, 13),
            new Transition(14, Tape.Blank, Tape.Blank, HeadDirection.NoMove, State.Halt),
            new Transition(14, '1', Tape.Blank, HeadDirection.Right, 14),
            new Transition(15, Tape.Blank, Tape.Blank, HeadDirection.Left, 16),
            new Transition(15, '1', Tape.Blank, HeadDirection.Left, 15),
            new Transition(16, Tape.Blank, Tape.Blank, HeadDirection.NoMove, State.Halt),
            new Transition(16, '1', Tape.Blank, HeadDirection.Left, 16)

        };
    }
}
```

Below is the multiplication test. The answer is indicated by a string of 6 "1" characters on the tape.

```
[TestMethod]
public void MultiplyTwoNumbers()
{
    const string expected = "Head: ______(_)111111";
    var sut = new TuringMachine.Machine(
        0,
        new TuringMachine.Head(new[]
        { '1', '1', '1', TuringMachine.Head.Blank, '1', '1' }, 0),
        TransitionTableGenerator.Multiplication());

    var result = sut.Run();
    Assert.AreEqual(expected, result.Head.ToString());
}
```

# Conclusion

I hope you enjoyed this whirlwind tour of Turing machines as much as I enjoyed writing it. There are much deeper concepts just under the surface and I hope I've piqued your interest enough to make you hungry for more. My ambition is that this post sends you on a journey into the exciting world of computability. Don't be satisfied with merely knowing enough to write programs; keep digging until you master the domain. I'll leave you with a couple book recommendations that I'm sure you'll find invigorating.

- Understanding Computation: From Simple Machines to Impossible Programs - Tom Stuart
- Introduction to Automata Theory, Languages, and Computation - John E. Hopcroft , Rajeev Motwani, Jeffrey D. Ullman

If there is enough interest, I may do a follow up to this post on how to do pattern matching and some simple regular expression parsing using the Turing machine simulation.

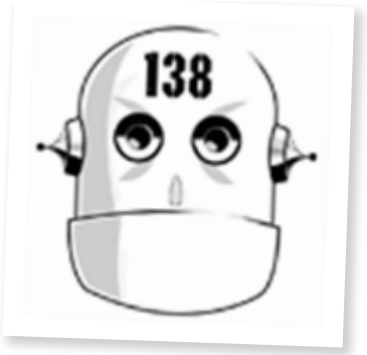
As always, thanks for reading and I would love to hear from you.

CodeProject

# License



# About the Author





## Hideous Humpback Freak

Software Developer (Senior)
United States


I'm a passionate developer with more than 20 years of experience in countless different languages. Currently, I'm a senior software engineer for a research company where I specialize in full stack web development with .NET. I have extensive experience with ASP.NET, MVC, C#, Azure, JavaScript, JQuery, AngluarJS, TypeScript, NoSQL (RavenDB), and SQL Server.

# Comments and Discussions

**3 messages** have been posted for this article Visit **https://www.codeproject.com/Articles/1066233/Turing-Machine-Simulation-in-Csharp** to post and view comments on this article, or click **here** to get a print view with messages.