University of Agder Norway

Turing Project Report

Group Turing
Fredrik Kartevoll, Trygve Solberg & Ludvig Alvir
https://github.com/uiano/Turing-MPS

IKT445
19.12-2021

Abstract

The Turing Machine is a mathematical model of computation that defines an abstract machine that is capable of computing anything any computer on the market can compute [1]. The Turing Machine's models concepts and their relations define the Turing language. This report will present the Turing programming language, which has been built in Jetbrains' language workbench: Meta Programming System (MPS). This programming language is a domain-specific language designed to be a simplistic programming language for domain experts that does not require the same programming knowledge as a general-purpose language. This project was started in 2017 by a different group. The group did not finish the project, and other groups have continued the work in the following years. In this report, we present the previous groups' work, as well as our group's contribution to the project. We have implemented a new run button, changes to the editor, and improved the language's documentation. The Turing language is a functioning programming language that can be run in MPS. The language can be an excellent tool for users who want a simplistic programming language for creating Turing Machines.

Contents

List of Figures

List of Tables

1 Introduction

Alan Turing first described the Automatic machine in 1936, later dubbed the Turing Machine [2]. The Turing Machine is a mathematical model of computation that defines an abstract machine [1]. The Turing Machine consists of a limitless input-tape, which is divided into cells, where each cell can be blank or contain a single symbol. The machine has a head that can read the symbols from the tape one cell at a time. The head has different operations it can execute. The first operation is to erase the symbol in a cell or print a symbol to the cell. The second operation is to move the head right or left one cell at a time to process a new cell. In addition to these operations, the machine has states it can transition between. These states contains rules or instructions of what operations the machine should execute based on the read input symbol. The Turing machine is capable of computing anything a computer on the market can compute due to the limitless memory tape and the operations just described.

The Turing Machine's concepts and their relations can be represented through the Turing language. In this project a Turing programming language has been built using Jetbrains' language workbench: Meta Programming System (MPS). The Turing language is a domain specific language (DSL) designed to be a simplistic language for Turing Machine developers. This DSL allows domain experts to create Turing machines without requiring the same level of programming expertise as a general purpose language (GPL).

1.1 Background

The Turing programming language project was started in 2017 by a group of students at UiA. The project was not finished in 2017, and have have been further developed by new groups in 2018 and 2019. Our group will continue the development of the Turing language at where the 2019 group ended their work. The Turing language is in a state where you are able to create functioning Turing Machines. We would like to improve the project in the following ways:

- **Run button** - Add a new run button that allows users to run the Turing machines without entering the machines' editor
- **Editor** - Improve readability of combination machine editor, and create an editor aspect for the table machine.
- **Documentation** - Add more informational documentation to make it easier for new developers and users of the language.

1.2 Previous Work

The previous Turing groups have created a programming language where you are able to build functioning Turing machines. The work of each group will be described in the this section.

Turing Group 2017
The project was first started in 2017 and the initial Turing language was built in MPS. The group implemented a text-based table machine editor with all necessary concepts and relations to build a working Turing machine. The table machine editor was also able to call already built machines. The group were not able to implement a runtime environment, and thus was not able to run the built machines.

Turing Group 2018
The group abandoned the previous years work and started the project over. The group created a table machine and a combination machine, as well as a runtime solution for the Turing machines, which made it possible to execute the machines through intentions using alt+enter. When running a machine a popup is displayed, and the user to write the symbols for the input tape. After they operations of the machine is complete, the resulting tape string is shown with a popup. They also created a new textual node based editor for the combination machine.

Turing Group 2019
The group implemented a tape input field and a run button in the editor. This allows the user to type the tape content in the top of the editor. The new run button provides a new way to run a machine. The group also changed the combination machine editor presentation to a textual editor instead of the textual node based one from 2018. The runtime solution from 2018, was removed by this group and replaced with a class under behavior. The group started implementing changes to the table machine, but it was not finished and an editor aspect is missing.

1.3 Report Structure

The coming sections of the report will cover:

2. **Language** - Describes the Turing machine's key components and how the Turing machine functions. The description of the Turing machine will then be used to define the Turing language.
3. **Solution** - Describes the concepts of the Turing language with their relationships and constraints. Further on, the semantics and execution of the language are described.
4. **Discussion** - Our thoughts on how well the Turing language works and what could be improved, as well as what could have been done differently.
5. **Plan and Reflection** - How the project management went in regards to time management and goals.

2 Language

In order to understand the Turing language, knowledge of the key concepts of a Turing machine is required. This section of the report will explain the components and functions of a Turing machine. The components of a Turing Machine can be seen in figure fig. 1 on page 6. The Turing language consists of the concepts read, write and move and how they operate according to instructions defined in states given an input symbol.

2.1 Tape

The Turing Machine (TM) has a limitless memory-tape[1]. The tape is divided into cells, where each cell can be blank or contain a single symbol. The Turing machine can solve a predetermined task by manipulates symbols on the tape according to a set of rules/instructions. When the tape has been processed by the TM, the answer to the predetermined task can be read from the altered tape.

2.2 Alphabet

The TM must have a finite alphabet of symbols it can read, the symbols in each cell will determine what rules to follow and the symbols must be known beforehand. In our Turing language the alphabet consists of either 0 or 1. Since the tape is infinitely extensible there is a special blank symbol used to fill all remaining cells on the tape.

2.3 Head

The Turing machine has a head which has three main functions: Read, write and move. The head will always be positioned at one cell of the tape at a given time.

2.3.1 Move

The head can move one cell at the time to either right or left in order to select which cell on the tape to read from or write to.

2.3.2 Read

The Turing machine head can read what symbol is in the cell where the head is located, this is called an input symbol. The read symbol will determine the action of the TM according to its rule set.

2.3.3 Write

The Turing machine head can overwrite the symbol in the cell where the head is located. The TM will use the rules set to determine if the cell should be overwritten, and what symbol that should be written.

2.4 State

The Turing machine will always be in a state, and this state is called the current state. Each state will have a given rule which determines what should be written to the cell, which direction the head should move and what will be the subsequent state. according to the input symbol. See fig. 2 on page 6.

2.4.1 Initial state

When the Turing machine is initialised, it will always start in a given state called the initial state.

2.4.2 Final state / Halting state / Accepting state

One of the Turing machines states has to be defined as the final state, this state may also be referred to as the halting state or accepting state. The Turing machine can enter the final state from a defined state with a defined input symbol. When the TM is in the final state the operation(s) are complete and the result is written on the tape.

2.5 Rules

The Turing Machine needs a set of rules that tells the program what to do in any given state with any given input symbol. If the current state is x with read symbol y, do action z and enter state x2. The action could be to write a new symbol to the current position and move Left or Right, followed by entering a new state. For the next position and state, the rule set will define what the next action will be.

2.6 Transitions

The operation of a Turing machine consists of several transitions. Each transitions consists of the following actions in this order: Read input symbol, write to cell on tape, move the head one cell to either right or left and then change the state. fig. 2 on page 6

2.7 Combination machine

One Turing machine can run other already built Turing machines, this is a called a combination machine. The called Turing machine will replace the original Turing machine at the same position of the tape, and then the original Turing machine will continue where the called machine halted.

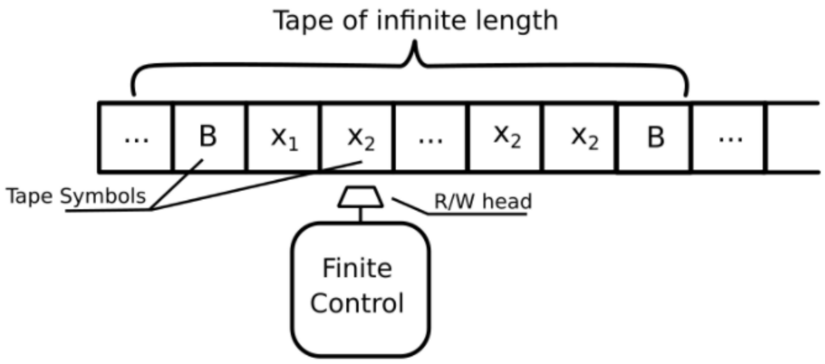


Figure 1: Turing Machine - R/W to tape

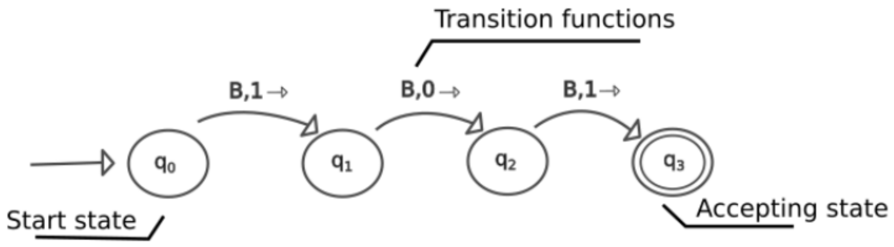


Figure 2: State Transitions

3 Solution

When our group started working on the project, most of the components needed to create Turing machines were working. This section will describe these concepts, their constraints and their relationships. We will also specify what implementations our group have implemented.

3.1 Structure

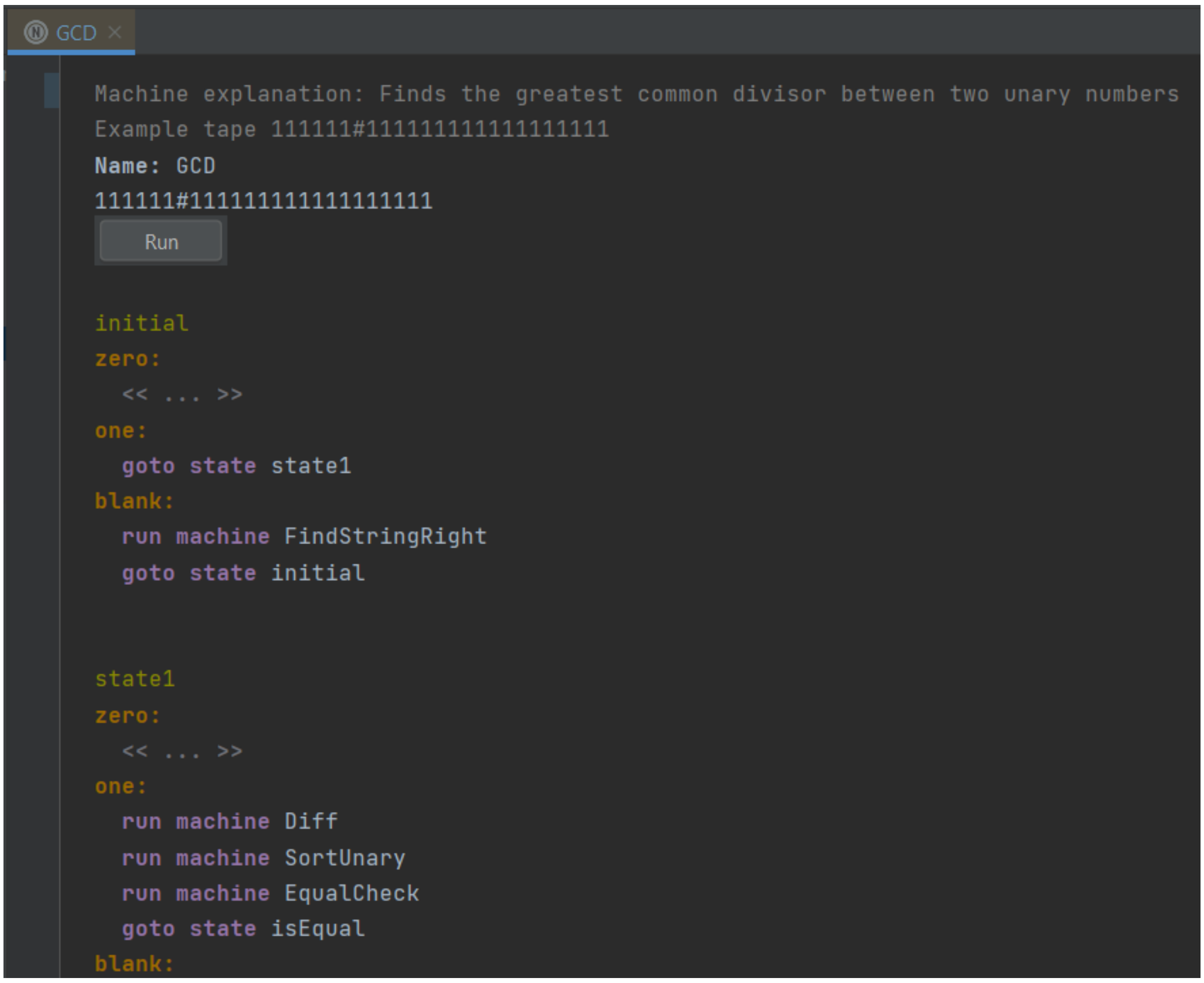
The language consists of several concepts. Each concept represent a component of the Turing machine (see section 2) and how the component is related to other components. For example the machine CombinationMachine concept is a root concept, which means it can be the initial concept of a Combination machine. The CombinationMachine concept has the child concept CombinationState[1..n], which means the CombinationMachine consists of several CombinationStates. The CombinationState concepts has the child concept CombinationOperation[1..n], which means the CombinationOperation has several operations (read, write, move). The hierarchy of the children and parent concepts form the structure of the Turing language.

The concepts of the language was already built when we started to work on the project, so it was not necessary for us to implemented any new concepts. The concepts can have certain limitations in the editor, like name limitations or the scope of variables they can access. These limitations are called constraints and can be created or altered in the constraints aspect. We have implemented one new constraint that requires the user to use unique naming for states in the combination machine editor.

3.2 Syntax

At the current state of the project, there are two different machines that can be built in the language's editor. These two machines are the table machine and the combination machine. Both machines have a textual editor represented in a structural way. It would be preferable to have a tabular view for the table machine for a more intuitive representation. We have changed the editor of the combination machine to make it easier to use and more intuitive for domain experts, who may lack programming experience. We have removed the brackets and replaced the if command with a read command to make it easier to understand for a domain expert. We have also color coded the different components of the editor to increase readability. The new editor can be seen in figure 3 on page 8.

When we started working on the project, there were no descriptions of what the different machines did, or what the tape input should look like. To give new users information about the machines and the tape input, we have added text fields to each machine's editor, where short descriptions of the machines can be written. There is also a text field for an example tape so new users can easier understand how to create their own.



```
Machine explanation: Finds the greatest common divisor between two unary numbers
Example tape 111111#111111111111111111
Name: GCD
111111#111111111111111111

initial
zero:
  << ... >>
one:
  goto state state1
blank:
  run machine FindStringRight
  goto state initial


state1
zero:
  << ... >>
one:
  run machine Diff
  run machine SortUnary
  run machine EqualCheck
  goto state isEqual
blank:
```

Figure 3: Combination machine editor 2021

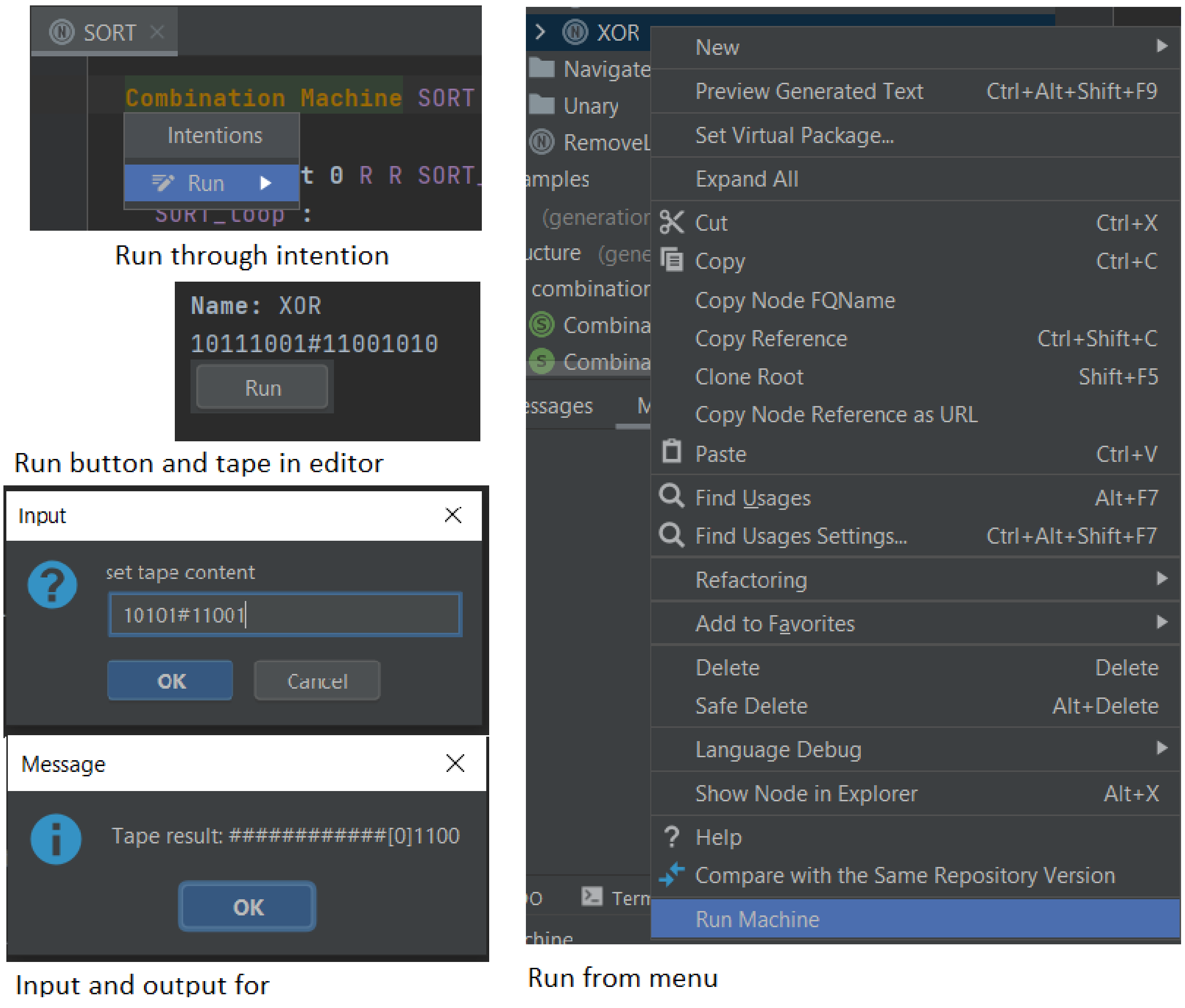
3.3 Execution

The Turing language is executed and run in MPS. The language concepts like move or write have behavior aspects. These behavior aspects contain Baselanguage code that executes the Turing language operations as baseLanguage. One behaviour will be the initial behavior that runs a machine. The initial behavior then runs the next behavior of the Turing machine, until all the instructions of the machine is complete. A string created in BaseLanguage represents the tape. This string is manipulated by the operations of the Turing machine, and after the machine is complete, the result is the manipulated string.

Running the initial behavior of a Turing machine is triggered in three different ways. The language from the 2019 group had two different ways to trigger the run: using an intention in the editor or using the run button in the editor. Our group wanted to separate the run button from the editor and created a new way to trigger the initial behavior. We added a run button that can be accessed by right-clicking a Turing machine within a model. The different ways to run a Turing machine can be seen in figure 4 on page 9.

The runtime element of the language are located in a class under the behavior aspect.

The runtime elements was added to this class by the 2019 group. The 2019 group had a separate runtime solution which contained the runtime elements.



```
Combination Machine SORT
t 0 R R SORT_
SORT_Loop :
```

```
Name: XOR
10111001#11001010
```

```
set tape content
10101#11001
```

```
Tape result: ############[0]1100
```

Figure 4: Run buttons and tape inputs/output of the language

3.4 Example Machines

When we started working on the project, there where two example machines implemented; increment and decrement. The group from 2018 had earlier tried to make a GCD (Greatest common divisor) machine, but ran into some trouble. For validation of the language we have created new functioning machines that complete more complicated tasks. For unary operations we created SORT, DIFF and GCD only using the symbols 1 and blank. Machines doing bit-wise operations like AND, OR, XOR, were created for binary numbers.

3.5 Documentation

When we started working on the project, the documentation explaining how the language works were lacking. We have created a manual.md in the MPS project meant for users of the language. The manual explains how to use the language's editor to create Turing machines. We have also created a language.md meant for developer of the Turing language. The language.md explains what the different language components in MPS are, and how they are connected. The 2019 group had removed the README.md file from the 2018 group. We have reintroduced the readme file and updated it for the current language version.

4 Discussion

The current version of the Turing language have a working combination machine editor in which we have created several functioning example machines, such as the GCD machine which computes the greatest common divisor for two unary numbers. We have created a new run button that can be accessed outside the machine editor. The readability of the editor for the combination machine has also been improved. We found that the documentation for the project was lacking, and have tried to improve with new documentation.

When we started working on the project we read through the documentation to try to get an understanding of the project. We found that the changes introduced to the programming language by the different groups over the years was not properly documented, and we struggled to understand how the programming language worked and what needed to be done. We have created informational documents to help new the next group that works on this project get a better understanding of the language and how to implement improvements.

Runtime
The 2018 group created a runtime solution which was added to the Turing language's model properties. This runtime solution was removed by the 2019 group, and it is not described in their project report why they did this. The 2019 group created a class under the behavior aspect, which now contains functions that works as the language runtime. The 2019's runtime works, but it is preferable to separate the runtime elements in a runtime solution. We advice the next Turing group to take a look at the 2018 version of the language, and try to implement this runtime solution in the latest language version.

Final state / Accepting state
As of now, the combination machine stop when it reaches a state that has an empty read input. One improvement to the machine could be to create a final state which will be the only legal state for the machine to stop at. If the machines stop at a different state with an empty read input, and error should be raised.

Table Machine
We tried to make a tabular view for the table machine. We used the MPS sample project multipleProjections (found in C:/User/user/MPSSamples 2021.2) were you can swap between a structural editor and a tabular editor using an intention as a reference. We tried to implement the multipleProjections in the Turing language, but was not successful. Late in the project we noticed that a normal editor aspect for the table machine was lacking from the 2019 group. Looking back we should have created a normal editor aspect for the 2019 table machine to make it runable. Now there is no editor aspect for the table machine, and this needs to be implemented.

combination machine representation
The textual node editor created by the 2018 group should be revisited. This editor is an orderly representation of the combination machine, but the usability could be improved. The combination machine editor in our version of the language is also a good representa-

tion of the combination machine, but the two different representation should be compared in order to decide which one is best.

Remaining improvements
The remaining improvements that need to be done are listed in the projects TODO.txt document, which can be found in the MPS project. The most important improvements can be summarized as:

- Creating a functioning editor aspect for the tabular machine, preferably an option for swapping between structural and tabular editor as in the multipleProjections MPS sample project.
- Create a separate runtime solution. Use 2018 version as a reference.
- Review what combination machine implementation is the best: 2018 vs 2021.

5 Plan and reflection

5.1 Version Control - GIT

We have used Git to work in separate branches when working on different software features. Separating the features into branches has allowed us to work in parallel on the same code and then merge the changes from the branches when the features are complete. Git has also been a great tool for tracking changes.

5.2 Time Management

The group had several meetings during the semester where we met in person to work on the project. The reflection assignments we have delivered throughout the semester have provided insight in the different components of the Turing language. The previous Turing groups work was not sufficiently documented, which caused us to spend a lot of time trying to understand how the language had been built in MPS and what needed to be done. Better documentation could have provided us with the information needed to easier develop the programming language. We hope that the documentation we have created can help the next Turing group.

References

[1] B. Jack Copeland Alan M. Turing. *The essential Turing: seminal writings in computing, logic, philosophy, artificial intelligence, and artificial life, plus the secrets of Enigma.* Clarendon Press; Oxford University Press, 2004. ISBN: 9780198250791,9780198250807,978142

[2] Alan Mathison Turing et al. 'On computable numbers, with an application to the Entscheidungsproblem'. In: *J. of Math* 58.345-363 (1936), p. 5.