UNIVERSITETET I AGDER

**GROUP: NEWERA**

---

# TURING MACHINES

---

**IKT445: GENERATIVE PROGRAMMING**


**WRITTEN BY:**
ELLEN SYNNØVE SAMDAL
RUNAR ISAKSEN
ALINE IYAGIZENEZA

**SUPERVISOR:**
ANDREAS PRINZ

**FACULTY OF ENGINEERING AND SCIENCE**
**UNIVERSITY OF AGDER**

GRIMSTAD


05.12.2017

STATUS: FINAL

# Abbreviations

| Abbreviation | Explanation |
|---|---|
| TM | Turing Machine |
| MPS | JetBrains Meta Programming System |

# Abstract

Turing Machine is the chosen language that we have implemented in MPS, and it is done in connection with a project in IKT445, fall 2017. Turing Machine covers aspects of structure, syntax, and semantics.

# Contents

# List of Figures

# Chapter 1 Introduction

The task for this project was to implement a chosen language in MPS, where it should also be described the aspects of the chosen language, such as structure, syntax, and semantics. This report will give an overview of the implementation of Turing Machines. In addition, it will describe the aspects in the implementation of the language. It will be discussed the disadvantages and advantages of our solution and why we choose Turing Machines. The report will also give the empirical evidence of the correctness by giving some examples.

The final project can be found in our Git repository, the latest version on the branch *"Delivery"*.

# Chapter 2 Language

The language is defined by different aspects for example structure, syntax and dynamic semantics. Structure is meta-modeling which is the analysis, construction and development of the frames, rules, constraints, models and theories applicable and useful for modeling a predefined class of problems. The syntax of the language is the set of rules that defines the combinations of symbols that are considered to be a correctly structured document or fragment in that language. Semantics describes the process of transformation and execution of a specific language.

A Turing Machine (TM) is a mathematical model which consists of a tape divided into cells on which input is given. The tape should preferably be of infinite length, but in practical terms it is "as long as it has to be". It consists of a head which reads the input tape. A state register stores the state of the TM. After reading an input symbol, it is replaced with another symbol, its internal state is changed, and it moves from one cell to the one on the right or left of it. If the TM reaches the final state, the input string is accepted, otherwise rejected. Through these simple actions, this language can do everything other languages can do, although at a very low level.

TMs have long be seen as a standard for compute ability in language discussions, for this reason we elected to create a language that would be able to create and run a TM inspired language.

# Chapter 3 Solution

## 3.1 Structure

Structure is where the basics of the project are detailed. It describes what concepts the language has, which serves similarly to classes in other languages. It describes what kind of relations concepts have to other concepts, as well as variables the concept holds.

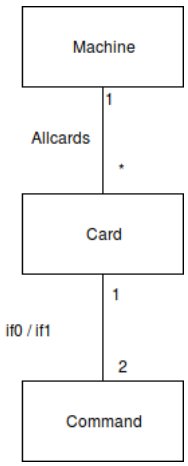


Figure 3.1: The structure of the Turing Machine

### Machine

Machine is the root concept of this language. By implementing INamedConcept it serves as the name of the sandbox files (these are the files that users of the language will edit). The list *"allCards"* is a reference to all Cards that is created by the user. There must be at least one card; this is because it would not make sense having a program without any content. The last feature of Machine is that it implements ScopeProvider, which is used to restrict *"nextCard"* in Command.

### Card

An Alternative name for Card is State. The name Card was chosen because of a comparison that had been made to the concept and to physical card where you could write the commands for the Turing machine to operate after. State is also a more generic word that is often used in programming and mathematics, so differentiating from these concepts would be preferred.

Card implements INamedConcept to have some predefined naming rules, as well as making it easier to work with in other parts of the language as compared to string. The Card's tasks is to choose the correct action based on the input given, which is either a 0 or a 1. The input decides what it will be overwritten with, which bit to move to next, as well as which Card is to be activated next. The Card really only handles the "if" statement where it has two children of type Command, called if0 and if1, which refers to whether the input is a 0 or a 1.

## Command

Command is a concept for storing three variables which will be specified by a user of the language. These are *"Write"*, *"Move"*, and *"nextCard"*. Write is an integer that tells the program what to write in the current bit. Move determines which direction to go to find the next bit. NextCard is a reference to a card which tells the program what card is next. The reference can be 0 or 1 where no reference will signal the end of the program, otherwise it should run the nextCard specified.

Move is restricted to only be able to write the direction either as the whole words "right" or "left", or alternatively only write the first letter of the word as a short form. The move restrictions is case insensitive, which means that any combination of lower and uppercase letters will work. The setup was chosen to force the user to be descriptive so it is easier for the program to recognize and easier to read, while still leave some options to the user. Write can only be a 0 or 1, as the main TM concept only allows for two symbols.

NextCard has a limited scope so that it only has access to variables in the same machine. This was done using the Scopeprovider in Machine to limit the scope to variables it has access to. The reason for limiting the scope is to reduce the number of possible references, as the auto-complete will show all Cards from all Machines otherwise. it can also help to reduce unintended bugs if the user is not careful and has several cards with same or similar names.

Bellow is an example for when an constraint is not followed.

```
machine navnetPaGreia {

  all cards :
    card1 :
      if0:
        Write: 2
        Move: L
        Next: A
      if1:
        Write: 1
        Move: Wrong
        Next: <no nextCard>

    card2 :
      if0:
        Write: 1
        Move: L
        Next: card1
      if1:
        Write: 0
        Move: L
        Next: card2
```

As we can see, the Write in "card1" is giving an error because it is not a 0 or 1, the Next is not referencing a Card that exists in this Machine, and the Move is neither left or right.

Figure 3.2: Constraints giving errors

## 3.2 Syntax

Syntax is how a language is presented to the user. Syntactically editing syntax is different from editing structure, here one create rules on how the language should be presented and edited by the user. It is specified how the variables from structure is edited by the user. Below you can see a basic, one card TM presented as the user will see it.

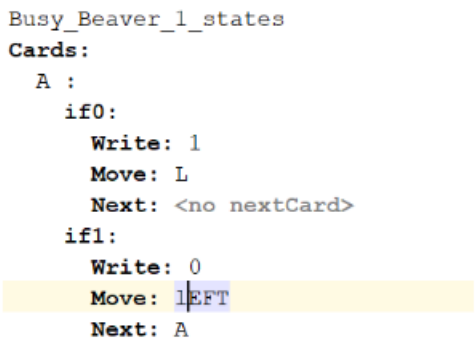


Figure 3.3: Text Editor

### Machine

A Machine's representation contains a name field where the user can write in a name for the program/Machine, and it then has a constant text "Cards" specifying that after that it is the Cards that are presented. All cards are indented to create a better visual differentiation.

### Card

Cards are presented by having a user first create the name, followed by an indent on a new Line. The indent signifies that it is technically the Commands that are specified here, however the Card does separate the statements with a constant text if0 and if1 that tells the user when the two commands will run. The Cards have a blank line at the end to create a visual separation between cards. This helps with readability.

### Command

Each Command will show the three variables, with a single word explaining which variable it is. The words are "Write", "Move", and "Next". The ":" was chosen as it felt more appropriate compared to "=", due to it not being an assignment that can be changed at a later point.

## 3.3 Semantics

The execution of this language is not complete. There was an attempt at using Java-code to help run the different Machines, but there was not enough time and too many issues. One example of an issue was that MPS could not find the code when trying to add it to the language, even though it was added in the project. The sections below will be discussing the best solution we found without the Java-code.

### Machine

Machine is the root of the program, so this is also where the main function of the program will occur. This is done by having a root mapping rule in *generation/main*. The rule points to a class which we have called *"map_Machine"*.

map_Machine has two jobs. First it places all the sources of all the cards out so that the functions for all the different cards are available. The second job is to create the structure of the program that does not occur in the MPS structure. This includes how the program should run through the different cards.

map_Machine also handles the code for the Card concept; this is just because it was easier to reference the different cards when it was created in the same MPS file.

### Command

Test_Command is fairly small and only handles the variables inside of the command concept: how it should write a bit and which direction to move.

## 3.4 Perfections and Imperfections

The part that worked out very nicely in this project was defining the structure of the TM and the text editor.

The biggest problem that is sill existing in the solution is the lack of run-time.

# Chapter 4 Discussion

As described in chapter 3, the implementation of TM in MPS is covering (almost) all aspects of the language. Some examples have been made as sandbox solutions, for testing that our editors are correct and that our constraints are working. one example of the latter is in the class card one can only have two children of type "Command", which should give an error if there are more or less. Another example will be how one can not write anything but "l", "left", "r" or "right" in the Command's Move. It was decided that these four ways of describing direction were enough for directions, as you can not really have any short term of the words larger than one letter and the two words should be sufficient. With that said, there might have been other ways to solve the task that might have been better. If one were to use a separate concept for Move one could perhaps create it so that one could auto-complete it or have it be any word while still giving the program a possibility to understand the action. The reason we did not do it that way is because it is more difficult to do without adding a lot to functionality.

TM was chosen based on information on this language being easily available on the internet compared to other languages. It was also quite easy to understand.

We chose to use run time classes that were not part of the concepts in our implementation since it was easier to implement in MPS than implementing all the logic in the very restrictive generation part of MPS.

There were many alternatives when making the editor. Some of them were as a diagram, tabular, or as text. Text was used for this project, simply because it was the easiest to implement in MPS. In the future, a table or a diagram would be preferred as it is easier to read.

The solution implemented in this project has its advantages and disadvantages. The advantage was that it is clear to understand how the language is built in MPS, and how the parts work together as it is fairly simple and with few concepts. The minus with this language is that it is not simple to implement a working run-time implementation, as it requires a lot of interconnectivity between the Cards.

# Chapter 5 Plan and Reflection

## 5.1 Planning and Time Spent

The group met and worked together several times during the project period, and most of the work was done together during these meetings. There was always at least a part of the group present for all guidance meetings with the supervisor, where we got feedback on what we have done and help with the next step in the project.

The group tried to work one or two days per week for at least four hours, and for the most part succeeded.

## 5.2 Reflection and Conclusion

At the start of the project period the group chose the language Text. After struggling with the language for some time, the language was changed to TM. The delay caused by first working with Text is the reason why the group did not have time to implement other syntaxes. Despite the delay on starting the main project, the language is implemented with all aspects of a language, except execution. This is partly due to MPS not having a run-time environment (RTE), making it hard to give the language a chance to run. If not for the delay at the start, the group is confident that a solution would be found, and the programs in the language would be able to run.

The group is satisfied with the results of the project, with the exception of the execution.