Turing Project Report

Group: Turing

Florian Weingartshofer & Eivind Hagtvedt

Github

IKT445

17. December 2023

Abstract

The Turing language is a language based on the concept of the Turing machine implementing in the Meta Programming System. It consists of two primary components, the table machine which performs operations on the tape, and the combination machine which runs Turing machines. Over the years multiple groups have made additions to the language, which includes creating a structure for each language, a syntax and an execution aspect. In this iteration the combination machine is given a diagrammatic syntax, and a runtime environment is added with debugging functionality for both the Turing machine and the combination machine. A test suite is also added so that tests so that the correctness of combination machines and Turing machines can be verified. Finally, new examples of the combination machine has been added to test the language's capabilities and to prove that it works.

Contents

1 Introduction

The Turing Project started in 2017 with a group of students at the University of Agder. Over the years new student groups have worked on the project and added new features [2]. In this iteration, we are adding a graphical syntax in the form of a diagrammatic view for the combination machine, a runtime environment, a debugger and a test suite.

1.1 Report Structure

- **Language** - Describes the language that is implemented in MPS.
- **Solution** - A summary of the structure, syntax and semantics of the language, with an explanation of the changes made.
- **Discussion** - A short discussion of the choices that were made, the positive elements of the solution, its limitations and future changes that should be made. As well as an empirical proof of the correctness of the solution using examples.
- **Plan & Reflection** - A summary of the work plan and the management of the project.
- **Appendix** - MPS learning questions, the branch that we worked on and the prerequisites. As well as the figures of the combination machines that were implemented.

2 Language

This section describes the language that has been implemented.

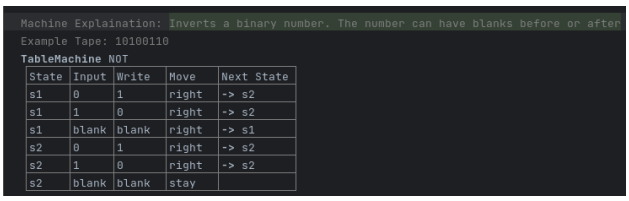
2.1 The Turing Machine

The Turing Machine is a computational machine that uses a hypothetically infinite tape to represent a program. The tape consists of cells that the Turing machine will read from and write values to. The machine accesses the tape through the head, and can move both forward and backwards through the tape, or remain in place. The behavior of the Turing machine is defined by a set of instructions described in a table. In this table there can be multiple states where the instructions will perform different operations on the tape depending on the current value at the head and current state in the table. Figure 1 shows an example of a table machine.[1]

The language has an alphabet consisting of three symbols; "zero", "one" and "blank".

2.2 The Combination Machine

The combination machine is a type of Turing machine that can execute a series of Turing machines a in sequence [2]. This can simplify the creation of Turing machines since it makes the Turing machine more modular. The combination machine uses the same alphabet as the Turing machines. The combination machine consists of activities that defines a sequence of Turing machines, and then navigates between activities based on the state at the head of the tape. The navigation is based on the current symbol of the head of the tape and the condition that is defined between two activities as an edge. The condition is either a specific symbol, like zero, one or blank, or an any match, which will match any symbol. Figure 2 shows an example of a combination machine.



State	Input	Write	Move	Next State
s1	0	1	right	-> s2
s1	1	0	right	-> s2
s1	blank	blank	right	-> s1
s2	0	1	right	-> s2
s2	1	0	right	-> s2
s2	blank	blank	stay	

Figure 1: The NOT Table Machine

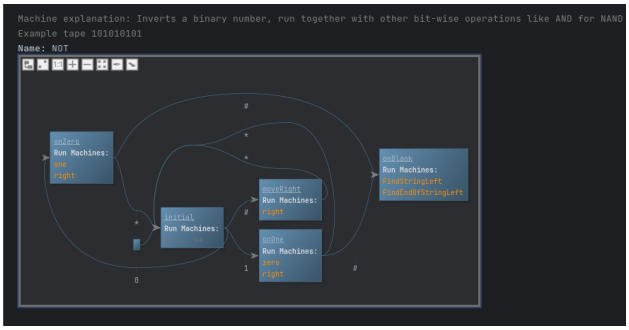


Figure 2: The NOT Combination Machine

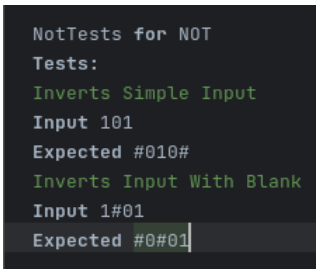


Figure 3: Test Suite for the NOT Combination Machine

2.3 The Test Suite

The test suite allows to define tests for table machine or combination machine. First the machine is defined and then a number of tests can be defined. A test has an input tape and an expected output. If the resulting tape of a machine run matches the output tape the test was successful. Otherwise, it failed. The test suite is a completely new addition to the project, so there are no previous versions of that feature. Figure 3 shows an example of a test suite.

3 Solution

This sections covers the solution for the language and the changes that have been made.

3.1 Structure

This section covers the structure of the language.

3.1.1 Common Elements

There are a few common elements which can be reused across the different machines and the newly added test suite concept.

The machine concept has common properties for both table and combination machine. The common initial tape string was removed and replaced by the machine explanation and

an example tape string. The reason for removing the tape string is that it is part of the runtime environment and not of the model.

The tape concept contains the tape string and has a constraint that checks if the tape string is valid, e.g. it only contains 0, 1, or #.

CellValue and Movement are enums which contain the values that can be present on a tape and in which direction the tape can be moved.

The state concept is a interface, which is used by the Activity concept and the TableState concept, it implements the INamedConcept.

3.1.2 Table Machine

The TableMachine concept inherits from the Machine concept and contains the TableStates. It can have one or more TableStates. So there has to be one state at least otherwise the Turing machine would not perform any actions. The TableMachine implements the Machine interface, which once contained the tape on which the machine operates. This tape was removed from the Machine concept and moved to the runtime environment. Furthermore, the machine explanation and the example tape were moved from the TableMachine to the machine interface, since these are properties, which are shared with the combination machine.

The TableState concept inherits from the state concept and contains the TableOperations, "zero", "one" and "blank". It must have exactly one of each of these operations. What this does is ensure that for any given value read from the tape, there is exactly one set of instructions. The TableState defines an explicit constraint that ensures, that there are no duplicate names.

The TableOperations contains the set of instructions write, move and goto. While the write and move instructions must have exactly one, the goto instruction can also be left out. This enables final states for the Turing machine. The table operation used the Operation interface, but this interface was removed since it did not add additional functionality.

The TableWrite concept is used to write a value onto the current head of Turing tape. This value can be "zero", "one" and "blank".

The TableMove concept is used to move the head of the Turing tape to a new cell. This can be "stay", "left" and "right".

The TableGoto concept references exactly one TableState and is used to navigate to the next state in the Turing table. It has a referent constraint that prevents the suggestions of states in other table machines.

3.1.3 Combination Machine

The structure of the combination machine was focus of major restructuring. The reason for that is to make the combination machine ready for a diagram editor instead of a simple text editor. Some concepts such as the "Combination state" concept was renamed to "Activity" to better describe the concept. The conditional and GoTo concepts were merged into one GoTo concept, where the GoTo concept now has a condition on which it is activated. This means there are no longer conditionals inside of an activity that determines which machines are run and the gotos, but instead all machines inside an activity are run and the conditionals decide only which activity is the next. This is a much cleaner look for a diagrammatic syntax.

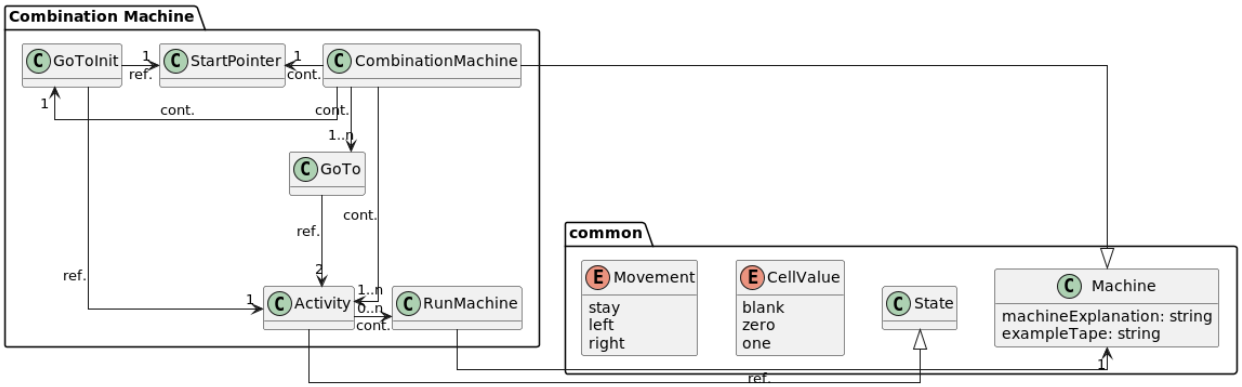


Figure 4: Combination Machine Structure

It was also moved into the combination machine to implement the diagram editor more easily. Since two combination operation were removed out of the three, the third one being the run machine concept, there was no need anymore for an explicit combination operation concept, so it was removed too. A StartPointer and InitialGoto concept has been added, since it is useful for the diagram view as it creates an easily discernible start point. Figure 4 shows the current structure.

The CombinationMachine concept has been changed to contain zero or more GoTo concepts, which makes it easier to keep track of the edges between activities. Furthermore, it contains one or more Activity concepts. It has also been made to contain exactly one StartPointer and exactly one GoToInit that connects the StartPointer with an Activity. This means that there is an implicit constraint that there must be a StartPointer, but only one, and an edge to the first Activity. Without them the combination machine would not perform any operations on the tape or it would have multiple start points which does not make sense.

The CombinationMachine implements the scopeProvider interface to only allow activities that are declared in the combination machine to be used in gotos. This helps with developer experience, since it is not possible anymore to accidentally reference an Activity of another combination machine. This prevents bugs and increases productivity.

There is an explicit constraint in place to ensure that there is at least one Activity without any outgoing edges, these activities are known as final activities.

The Activity concept has a containment relationship with the combinationMachine concept and it inherits from the State concept. An Activity can have zero or more TableMachine references, which means that each Activity can run multiple Turing machines or none at all which may be the case for final states. The Activity concept also had the explicit property constraint that ensured the uniqueness of the Activity name, which was moved to the typesystem, since it is a semantic error.

The StartPointer concept is similar to the Activity concept although it does not contain any machine references since it is not intended to run any machines, it is merely points to the first Activity that we want to run.

The GoTo concept no longer references the combination state, instead it connects exactly two activities via references, one "from" Activity and one "to" Activity, and has a condition property so that the next Activity is matched against the current character at the head of the Turing tape. There can only be one goto connecting two activities and each goto must

have a different condition, this is ensured by an explicit constraint. Furthermore, a goto can only be connected between activities that are present in the same combination machine, also ensured with scopes and explicit constraints. The concept also inherits the scope of its parent combination machine, that way no external activities can be referenced in the "to" and "from" reference.

The GotoInit concept is similar to the goto concept except that the "from" node references the initial pointer. There is also a reference to the next Activity called "to". While it may be possible to remove this concept in favor of having the StartPointer reference the first Activity, such a solution would cause issues with the diagram view as it seems to require that the edge is between two diagram boxes. If the two concepts were joined it would have to be the edge between itself and an Activity. There are also no conditionals for this concept since there can only be one in the combination machine. The GoToInit can only connect to a starting point and Activity which are in the same combination machine. This is done by inheriting the scope from the parent combination machine.

The Condition can be matched against the tape, and allows the combination machine to filter which state should be run next. There are two types of condition an exact match, which is either zero, one or blank on the tape that is equal to the character in the condition, or an any match which is represented with an asterisk(*). The any match will match with any character on the tape.

The RunMachine concept references exactly one machine that it will run.

3.1.4 The Runtime Environment

The runtime environment have corresponding runtime concepts to concepts found in the combination and table machine. Each runtime element references its source concept to access its properties, children and references. The goto runtime element is the only exception as it acts like a program counter for one CombinationMachine runtime element.

The runtime CombinationMachine concept references exactly one combinationMachine and contains exactly one runtime Activity.

The runtime Activity concept references exactly one Activity and contains the run machines in the Activity as well as the current run machine.

The runtime goto concept references zero or one Activity concepts.

The runtime TableMachine concept references exactly one table machine.

The runtime TableState concept references exactly one TableState.

3.1.5 Test Suite

The TestSuite inherits from the INamedConcept. It references exactly one machine under test, a name property and contains zero or more MachineTests, which are run against the previously defined machine.

The MachineTest inherits from the INamedConcept. It defines a test name property, and contains an input and output tape, both are of the tape of the tape concept, which both must follow the tape syntax of consisting only of zero, one or blank characters.

3.2 Syntax

This section covers the syntax of the language and the changes made to the syntax.

3.2.1 Common Elements

The tape is a simple string that allows for 0, 1, or #. The validity of the string is checked using an explicit constraint.

3.2.2 Table Machine

The syntax of the table machine is unchanged from the previous group. The table machine is displayed as a table in textual form. Each row contains a set of instructions that will be executed according to the state and value read from the current head of the tape. This solution makes the creation of a Turing machine fairly intuitive, and shares similarities to common representations of Turing machines. Above the table in the table machine are input fields with a title for each. These are the machine explanation that describes the machine, the example tape which shows an example of the tape that can be input, and a naming field for the table machine.

The TableState is the first column in the table. It accepts a string input and has an explicit property constraint that prevents multiple states from sharing the same name.

The next column is the input which reads the current head of the tape, while this field is editable it has no effect on the execution of the machine.

The TableWrite is the third column, and allows for only "0", "1" and "blank" to be written, this is an implicit constraint in the concept properties.

The TableMove is the next column which allows for three inputs, "left", "right" and "stay", which are also implicit constraints from the corresponding concept's property.

The TableGoto is the final column and accepts an existing state as input.

3.2.3 Combination Machine

Diagram view

The Combination machine is displayed as a diagram to make the creation of larger combination machines easier and to make it easier to understand the connections between activities. Though it can still be read as a textual syntax, it is no longer designed for it. To implement a diagrammatic notation for the combination machine the plugins de.itemis.mps.editor.diagram and com.mbeddr.mpsutil.editor.querylist are used. The plugins require MPS version 2022.3, since they have not been updated to work with the new 2023.x versions of MPS.

The combination machine has a diagram object which contains the relevant concepts such as activities and gotos. To enable the connection between two activities or the StartPointer, from and to node are specified in the connection creators. Additionally, the palette entries is made to show the activities and the StartPointer so that these can be added. While there can only be one StartPointer, and there is always one when a combination machine is first created, it should still be available in the palette in case it is deleted.

The Activity in the diagram is represented as a default rectangle, with the underlined name of the Activity at the top and then a string that indicates that it runs the following machines. The machines are highlighted in orange to contrast them from the rest of the

Activity. To ensure that edges to and from the Activity are also deleted when the Activity is deleted, a function is used to search through the gotos to remove any connected edges as seen in 5.

```
delete ()->void {
  list<node<GoTo>> gotos = thisNode.ancestor<concept = CombinationMachine>.gotos;
  sequence<node<GoTo>> toRemove = gotos.where({~it => it.to == thisNode || it.from == thisNode; });
  gotos.removeAll(toRemove);
  thisNode.detach;
}
```

Figure 5: Activity Deletion along with GoTo removal

The StartPointer is similar to the Activity, except it has no text. It is made to look different from the activities in order to distinguish it from them.

The edges between activities has an arrow shape pointing towards the Activity that the combination machine is transitioning to. The arrow makes it easier to read the diagram and understand the hierarchy between activities and the direction of the graph. It also has an input field where the conditional is written.

3.2.4 Test Suite

The test suite has a simple textual syntax. It is displayed first indicating the name of the test and which machine it is for. Then it has the tests in sequences showing the test name, the input and the expected output.

The test suite has a name and a machine reference in the first line of the editor. The next blocks, all on new lines, contain the machine tests.

The machine test has a name, an input tape on the next line and an expected output tape on the last line.

3.3 Semantics

The language uses execution not transformation. A combination machine can be represented as a directed graph, where each Activity is a node and each goto is an edge with the condition as value. The starting point is a special node, which can only have one outgoing edge with an implicit any value. The starting point has to be able to reach every Activity in one way or another. This means that there needs to be a check in place on how the graph is connected. If a graph is not fully connected and/or if there is a node without any in going edges the user will be warned. The decision on using a warning instead of an error was made to make the developer experience better, since during development it might happen that an Activity is created and not immediately connected. There are also typesystem checks in place that check if the activity's name is a duplicate.

The execution is done using behaviors with runtime concepts to create a runtime environment that the languages steps through when running. All runtime elements have to implement the runtime interface, which defines a few methods that need to be implemented by its descendants as seen in figure 6. This enables the ability for debugging, by stepping through the current runtime state when creating a combination or Turing machine. A runtime concept will always reference exactly one of the corresponding concept in the structure, with one exception being the goto runtime element. During execution the runtime elements are added onto a stack in the behavioral concepts, which are then removed from the stack and run in the run/debugger plugin. The behavioral concepts have functions that return information

```
interface concept behavior RuntimeElement {

  constructor {
    <no statements>
  }

  public virtual abstract void run(MachineState machineState, Stack<node<RuntimeElement>> stack);

  public virtual abstract string runtimeDescription();

  public virtual abstract string getMachineName();

}
```

Figure 6: Runtime Element Methods

about the current state of the system, such as the name of the current Turing machine or TableState in the Turing machine this is used in the debugger to provide information to the user so that they can better analyze the system.

The initial state of the tape is defined in an input message box that comes up when clicking "debug" or "run". There the user will input a sequence of valid characters where the head of the tape is the leftmost character that was input. The current head of the tape is found by reading the value at an index. This index is initialized as 0, and will increment if moving right and decrement if moving left. For both the Turing machine and the combination machine the final state of the tape is shown as a message dialogue when finished, the head of the tape will be at the last index of the machine state. The debugger will similarly display the current state of the tape and machines, and give the option of going to the next "step", to "cancel" or to "continue" which runs until the end of the machine. These functionalities are implemented as action plugins.

The test execution also utilizes the runtime elements described above. The tests itself don't have a runtime element, since the test plugin itself executes the tests. The plugin will load a test suite and create a runtime element of the referenced machine. Then it will run the machine with the input tape and compare it with the output tape. If the tape is equal the test was successful, if not it was a failure.

3.3.1 TableMachine

The TableState behavior when run is called will add the next TableState onto the stack with the operation on the tape and the moving of the head on the tape given the input value read from the tape. It can also return the name of the current TableState.

The TableMachine behavior can return information about the name of the machine. It will put the first TableState as runtime element on the stack, the first state is the one on top of the table view.

3.3.2 Combination Machine

Since a StartPointer concept has been added this will now point to the first Activity that is pushed onto the stack and then run. The final Activity of the combination machine is any Activity that does not have an outgoing edge that matches the current character of the head of the tape.

The runtime combination machine behavior is made to first add the Activity from the InitialGoto "to" attribute to a stack when run. It also finds the edges to the next activities.

The behavior enables the return of information about the current name of the combination machine which is used for the debugging.

The Activity behavior has a few utility functions that help to find neighboring activities, incoming or outgoing edges. Furthermore, it has a method to check if it is the initial Activity, the one that is referenced by the gotoinit concept. This methods are primarly used by the typesystem to check for semantic errors, but also by the goto runtime element to find all outgoing edges.

The runtime Activity behavior adds the edge and all the referenced machines, combination or table, onto the stack, when run. Similarly to the combination machine it allows for the return of information about the current state of the system for debugging purposes.

The runtime goto behavior initializes the runtime Activity and adds the next Activity to the instance of it. Then it adds the Activity to the stack. It also has similar functionality of being able to return information about the current state, such as the source Activity and the next Activity. It does not reference a specific goto concept, but the source Activity, i.e. the Activity that is referenced in the "from" field of the goto. The reason for this is, that the correct goto can only be found after all the machines, which are referenced by the Activity, have been executed. These machines can change the state of the tape, and the right goto can only be found using the tape state. So the goto runtime element references the source Activity and when executed will find the right goto in the machine and then put the Activity in the "to" field on the stack as a runtime element. Therefore, it is in a way a program counter. The runtime goto might not find a next Activity, this means that the machine is finished executing and there is nothing to put on the stack.

4 Discussion

In this section the project is discussed.

4.1 Positives

One major goal was to build a diagram editor, which worked out very well. The diagram editor is very intuitive to use and has some good quality of life features, such as auto formatting the diagram.

The introduction of runtime elements helped with separating the runtime environment from the the model. It also introduces the stack based approach to run the machines, which allows for easier extension of the runtime elements.

4.2 Limitations & Future Work

There are no constraints that prevents the combination or Turing machine to get stuck in an infinite loop. This is an issue since it crashes the program. This can either be solved by using a counter that stops the program after reaching a maximum value, or potentially by writing a property constraint that analyzes the machine and evaluates if it is finite.

The debugging functionality can be improved by having it show more information. For the combination machine this would be to have it show the current activity with all of the table machine references, and the states in the current table machine. The current machine reference, and the current state in the table can be highlighted with a "$*$" to indicate that it is current. Most of this would require changes to the debugger plugin, where it compares the

current machine reference name to the list of machine references in the activity to annotate the current machine. It also requires changes to the behaviors so that one can retrieve information to tell what type of machine is running and the list of table states and machine references.

Currently the alphabet that of Turing machine is defined in the implementation of the language and not by the user. This means that using a unary tape is not restricted to "blank" and "one" characters. This should be changed to make it possible for the user to define the valid characters for the Turing machine. This can be done by extracting the tape and CellValues into its own concepts were the alphabet is defined. The goto condition would need this alphabet to match against the current tape value.

There are a number of example machines, which all can be executed, but they have not been tested yet. An improvement would be to write test suites for each machine using the newly added test suites. This would allow to verify if a machine has been broken, after changing it.

4.3 Alternative solutions

An alternative solution to using stacks was to run the machines immediately rather than putting them on a stack before running them. Using a stack solution gives a greater control of the runtime environment by keeping the machines in memory before running them. It enables the development of the debugging plugin, which is a big advantage when developing complex machines. A disadvantage of this approach is that a completely new interpreter was implemented, and has to be maintained.

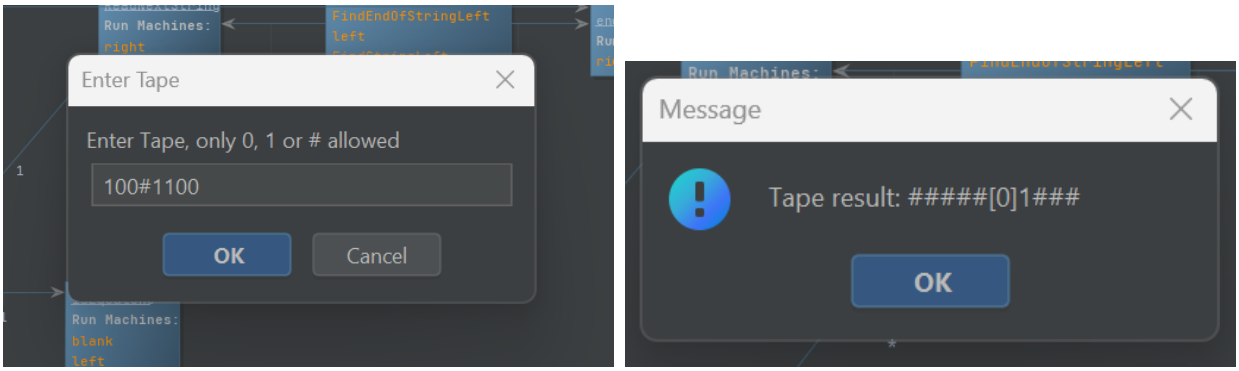
An alternative solution to the activity was to keep the conditionals inside the activity so that it determines which machines are run. Whilst removing the conditionals inside the activity makes each activity less capable of complex behavior, it makes them easier to read and understand. To achieve the same functionality of an activity from the alternative solution, in the current solution one will have to have more complex table machines or, more likely, use multiple activities.

4.4 Examples

The previous examples for the combination machines have been migrated to work for the updated solution. Additionally, unary addition, a palindrome detector and a combination machine that checks if two binary strings are equal has been added to further verify the correctness of the language. The new examples can be seen in the appendix. With the results of two inputs for the combination machine that checks that two strings are equal. The first one seen in figure 7 where the input is 100#1100 which returns ...[0]..., which means that it is not equal and in figure 8 where the input is 110110#110110 and the output is ...[1]..., which means that it is equal. Multiple strings of various lengths and combinations have been tested to ensure that it covers a range of possible inputs.

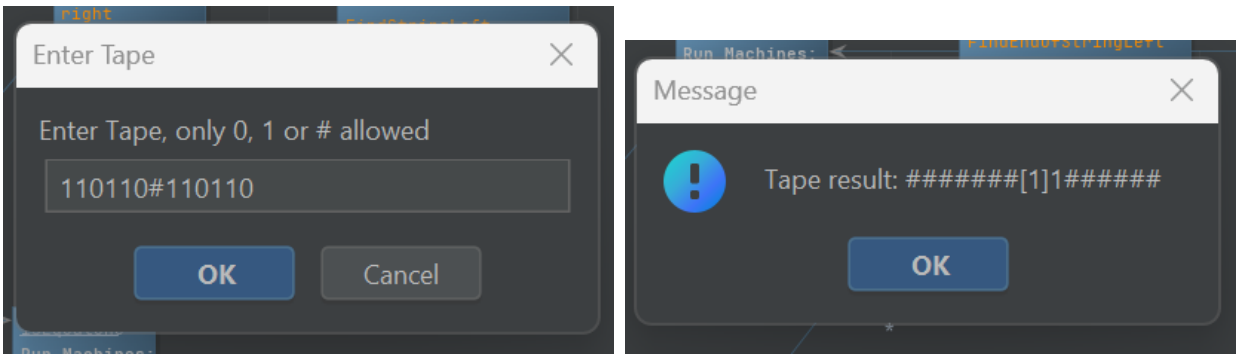
4.5 Meta Programming System

Our experience with using MPS is mixed. It makes the implementation of language fairly clear and simple, but it is also quite difficult to work with. Making changes often requires deleting more than one would want, it does not always recognize a reference link if changes have been made to the module. Furthermore, making drastic changes to the language



(a) Input **(b)** Output

Figure 7: Unequal string input returns false.



(a) Input **(b)** Output

Figure 8: Equal string input returns true.

makes migrating preexisting examples difficult. If a property is deleted from a concept and its editor, the change is not propagated to the examples in the project. The editor will not display the deleted property anymore, but it is still present in the underlying model, which results in an error in the example.

One major hurdle was the limited availability of online resources, since Jetbrains MPS is not as commonly used as other programming languages or tools. Errors often have to be worked on without any online help and the documentation is sometimes very limited on how detailed it is.

5 Plan & Reflection

There were challenges with fitting the project work in between reports and the readiness assurance test, but overall we managed to fit it in. We largely used discord to plan and cooperate dynamically. We used GitHub for version control and collaboration purposes. Since the whole team is proficient in Git this was no problem and we could work very efficiently together, without any conflicts.

GitHub issues were used to indicate on which problems we were working on. In general this is a common practice, but since we were only allowed to merge everything after submitting the report, we could not close any issues until after the merge of one big branch.

The workload also proved to be a lot. Especially after the presentation there were many changes requested, such as the addition of the test suites and the changes in the machine structure. Since this was about a week before the final submission, we had to stress to implement all the changes and writing the report. It would have been better if we were allowed to limit ourselves to the changes that were talked about in the beginning.

List of Figures

References

[1] *Models of Computation Lecture e: Turing Machines [Fa"']*. Tech. rep. 2016. URL: http://jeffe.cs.illinois.edu/teaching/algorithms/.

[2] Group Turing et al. *Turing Project Report*. Tech. rep. 2022. URL: https://github.com/uiano/Turing-MPS.

6 Appendix

6.1 MPS learning quiz Questions

Can MPS provide both a graphical and a textual syntax for a language?

- 1. No, MPS can only provide a textual syntax which is defined in the constraints.
- 2. Yes, by specifying in the structure of the language that it must show a graphical syntax or a textual syntax.
- 3. No, MPS can only provide a diagrammatic syntax.
- 4. Yes, but it requires third party plugins for the graphical editor. **(Correct)**

Where in MPS would you define relationships between concepts?

- 1. In the constraints section.
- 2. In the editor section.
- 3. In the structure section. **(Correct)**
- 4. In the properties section.
- 5. All of the above.

6.2 Github branch

The branch that our project work is on is the diagram-editor-for-combination-machine.

6.3 Prerequisites for using the language

The languages relies on two plugins to be installed in JetBrains MPS. The de.itemis.mps.editor.diagram-plugin, which is the implementation of the diagram view, and the com.mbeddr.mpsutil.editor.querylist-plugin. The plugins require MPS version 2022.3.

6.4 Combination Machines

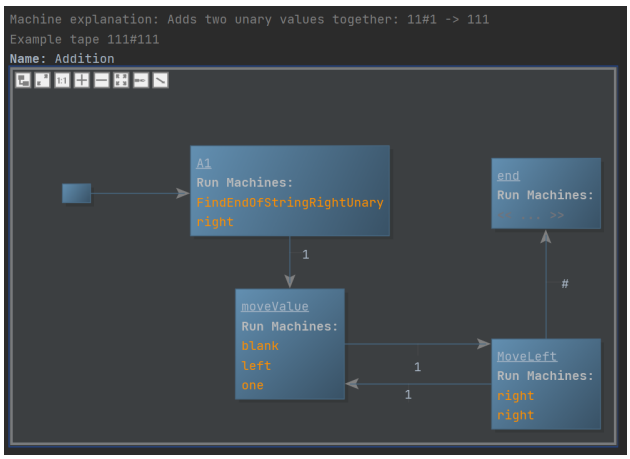


Figure 9: The Unary addition combination machine.

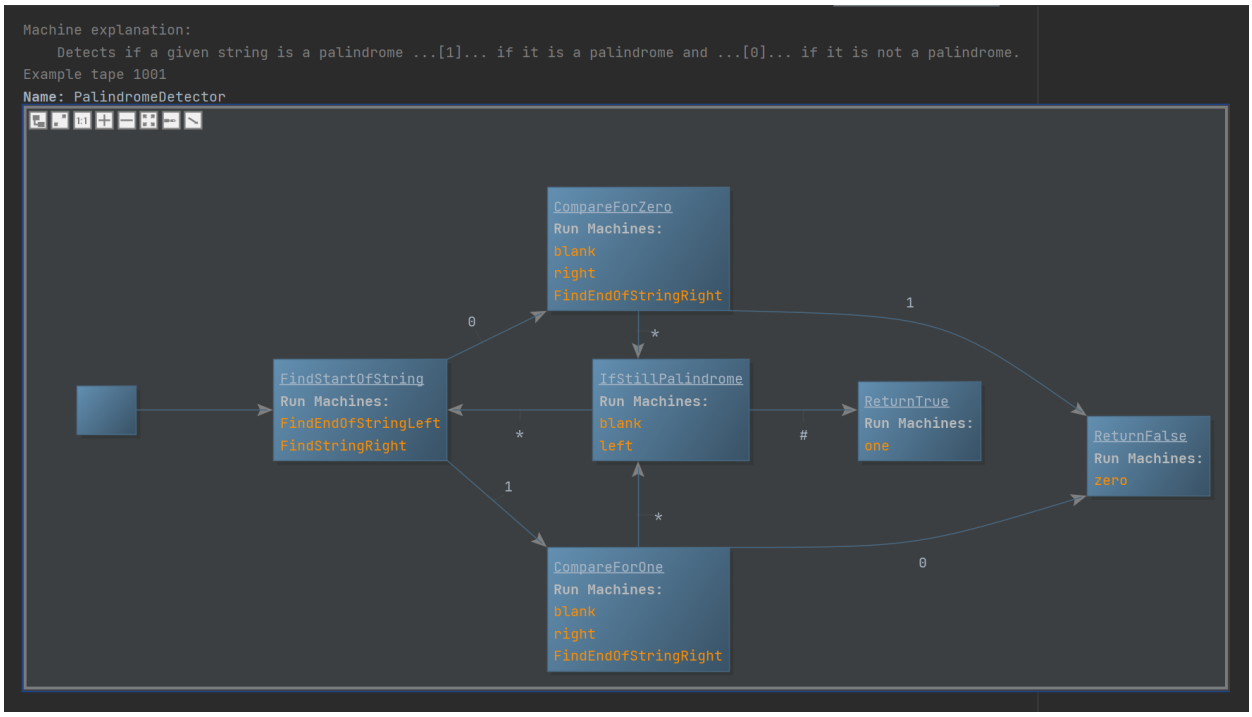


Figure 10: The Combination Machine that detects if a string is a palindrome.

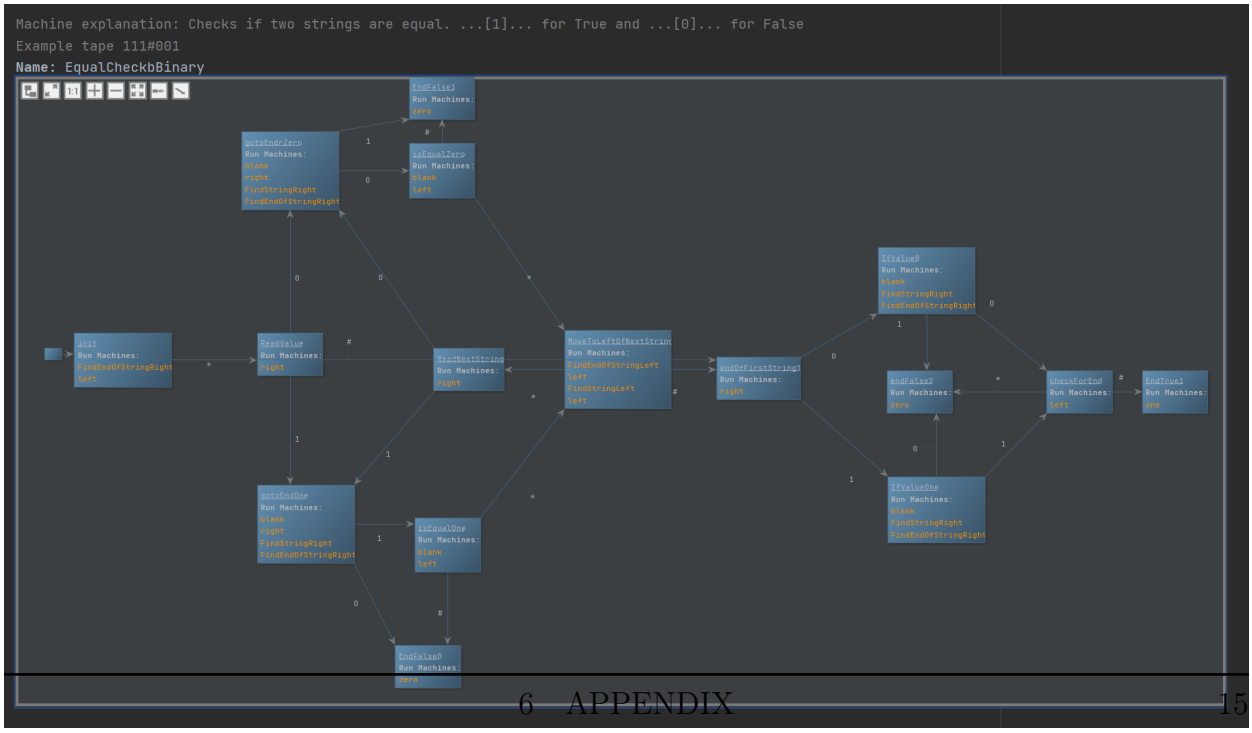


Figure 11: The combination machine that checks if two strings are equal.