Universiteit Leiden
The Netherlands

Opleiding Informatica

Creating a language to describe Turing machines

On creating a Towards Turing machine compiler

David Noteborn

Supervisors:
H.J. Hoogeboom & W.A. Kosters

BACHELOR THESIS

Leiden Institute of Advanced Computer Science (LIACS)
www.liacs.leidenuniv.nl

01/07/2024

Abstract

By means of following the steps of compiler design, a program to convert a more high level description of a Turing machine into simulation/C code can be created. During this process, a new comparison technique and algorithm is developed for Turing Machines, which could be used on directed graphs as well.

Contents

1 Introduction

The question whether all problems are solvable is one that has existed for a long time. Some problems, even those that seemed to require advanced forms of logical thinking, might not be solvable by only using calculations. Eventually Alan Turing created a system to reason about calculations and calculating. Ideas that follow this system are eventually considered 'Turing Machines'. He wasn't the only one in his time that was thinking about computation models. Turing's professor, Alonzo Church, also created a model in which one could calculate. This model is called 'Lambda Calculus'. Also people start thinking about general recursive functions. Now while computers might calculate with these ideas quite easily, humans might find these ideas and concepts difficult to grasp. Turing Machines consists of states, symbols and transitions. These are usually represented by circles, Latin characters and arrows. Humans on the other hand, usually think in terms of experiences and functions. In this thesis I want to answer the questions: How can we design a language capable of describing Turing machines in a comprehensible and human-readable way while remaining machine-interpretable? And how can we enrich this language with capabilities to express complex structures occurring in Turing machines in a concise and readable manner? The answer to these questions are important because they may narrow the gap between humans and the abstract concepts of Turing Machines. The recently released tool ChatGPT has raised popularity by outputting suprisingly humanlike responses to natural language. I will also answer the question whether this tool could be used to achieve this kind of conversion.

1.1 Language

If you have multiple systems, a desire or need to communicate can occur. 'Language' is the answer to the question: how do we communicate? Different systems use different forms of language. Take for example bees. They show others where to find nectar by moving in a way that humans describe as dances. Usually these dances contains moving in a round. This is totally different from the courtship display of a peacock. Both are however forms of language. From this point, we will discuss language only in the form of spoken or written by humans.
Natural language is what most people speak most of the time. English and Dutch are natural languages. Both change over time, and do so without any clear planning. They came into existence without a clear plan either. The only reason was the desire and need to communicate. Some languages are not natural but constructed. These are made by men who had ideas on how the language should be. The most common one is Esperanto. These constructed languages are hardly spoken. Natural language in its completeness is generally considered impossible to parse correctly by a computer program. This applies also to most constructed languages. As a last group of language we have formal languages. This group is defined by the fact that there are rules that can completely describe whether some expression, word or text is part of the language or not. There are no cases that are unclear. Programming languages are all formal languages. It is clear for these languages what expressions are in the language (the syntax) and what these expressions mean (the semantics). It should be noted that while in theory this is the case, a person that knows the syntax does generally not know what the semantics are of the expression. Most of the time a programmer would know a simplified version of the semantics, but rarely do people know the completenes of the meaning of their expression. Programming is not the only application to formal language however. Some languages are specifically formed to assist reasoning about logic. Alongside that certain file

formats, like XML and PDF are also well specified and can be considered a formal language.

1.2 Compiler

Most people have a hard time thinking in the concepts of computers. The computer processor only calculates in terms of registers and memory locations. These concepts are very abstract for the average human mind. Compilers are created to narrow the gap between human thinking and computer concepts by some form of translation. Today many programmers hardly have an idea what kind of instructions are native to a CPU. The first compilers all translated their input code to low-level assembly or machine code. Some later compilers also could translate a program to a simpler form, but still not interpretable by a computer processor. Another program would later execute the translated code. Some compilers even compile to a language that need to be compiled itself. An example is 'Jakt'. This is a new programming language that is planned to be the language most of the code of the Serenity OS project will be (re)written in. The current compiler compiles to C++. Also the compiler 'Natalie' can be used to compile Ruby code to C++. Due to all of these programs being known as a compiler, what a compiler actually is can vary vastly. Still there are seven stages that most compilers have in common. These stages are: lexing, parsing, semantic analysis, intermediate code generation, intermediate code optimalisation, machine code generation and machine code optimalisation. Lexing is splitting the source code into individual parts and annotating these parts with a type. Parsing is generating datastructures that represent the source code (or the part that describes instructions) on the basis of the lexer output. The datastructure that is used for this in 99% of the compilers is the tree structure. Semantic analysis is about gathering information that could not easily be done during parsing. Intermediate code is a simpeler form of code than the source code. In this code, a single aritmetic operation is represented using a single line of code. Intermediate code does not take into account the specific details of a certain target platform or computer architecture. Machine code is the code that can actually be executed on the target platform. This is for example some form of assembly code, or some text representations of byte codes.

1.3 Batreaux

To transform natural language descriptions into working C executables and diagrams I have built a tool. In this thesis I introduce BATREAUX[1]. BATREAUX could be known as more than a singular thing. It is both a language and a compiler-debugger program. If confusion could arise by using the name I shall use the term 'BATREAUX-language' for the language, and 'BATREAUX-compiler' for the compiler. BATREAUX-compiler is able to output parse trees and abstract syntax trees as graphviz descriptions, and is able to execute a described Turing machine itself. As input it can be given a description of the Turing machine, in English language. The description should be understandable by a person in the field of computer science. Also the description should not expect the other person to solve a problem by means of creativity. A description like: 'the machine should

[1]BATREAUX is the name of a character in the video game THE LEGEND OF ZELDA: SKYWARD SWORD. This character one first found to be a demon. One can choose to 'help' this character, which slowly makes it more appearing as a human. Similarly this program (and this language) have slowly become more like natural language through the development of this tool

accept strings that are palindromes' is therefore not allowed. Also the described Turing machine could be displayed using the graphviz technology.

1.4 Descriptions

There are multiple descriptions I have used to test my algorithm. In this report I will mainly discuss three: a simple description that is close to the graph structure of the Turing machine (the simple description, Appendix B), a somewhat more high-level description that is not explicit about every state (Appendix C), and a description that describes on a more free-form way than the other graph structure description (Appendix A). The latter description is known as the Bolhuis description as it is written by the well-informed outsider K. Bolhuis [2].

1.5 Graphviz

Graphviz is used in Batreaux for visualizations of the created Turing machines. Also abstract syntax trees are visualized using this technology. Graphviz uses the DOT-language to process the graphs. The DOT-notation that is created can sometimes be fitted in a URL to be displayed in a webpage. The sites Dreampuf [1], AduH95 [5] and Devtools Daily [4] are all supported in this. To generate DOT-notations more efficiently, objects are used to store rules in DOT. These objects are stored themselves in a list. A separate function then generates text from the list of objects that can be used as the input for Dreampuf.

1.6 Thesis overview

This chapter contains the introduction; Section 2 discusses the idea behind the Turing Machine and the notation I use in this thesis; Section 3 discusses possible sentence structures and properties of Batreaux-language; Section 4 includes how GOLD works; Section 5 discusses how I process the parse trees, which are converted to intermediate code. This intermediate code can be converted to machine code. Both conversions are discussed in Section 6; Section 8 discusses how I have used ChatGPT in the development of Batreaux-compiler; Also it discusses whether GPT could be used as a converter between descriptions and transition tables; To compare the output of GPT and the reference output a comparisons algorithm is made that is discussed in section 7; Section 9 discusses the quality of the compiler and its output; Section 10 concludes. The overview of the steps that the data followed though this thesis is shown in Figure 1.

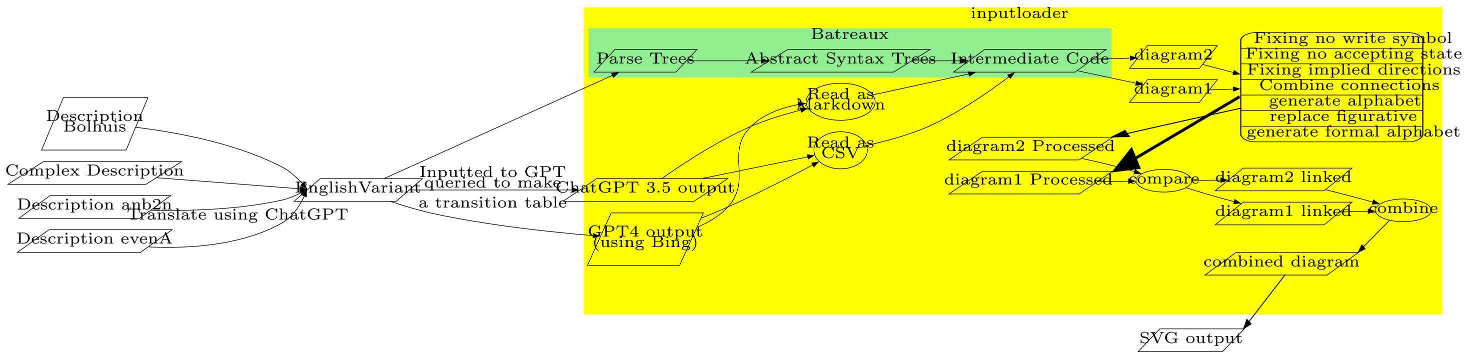


Figure 1: The general flow of the descriptions through this thesis. The green part are the steps that are the most characteristic for Batreaux. The yellow part is the steps followed within the program inputloader. This program uses the functionality of Batreaux.

2 Turing Machines and the used notation

Turing Machines were invented by Alan Turing [7]. He called these machines A-machines himself. A Turing Machine was created to have a very simple model that can still calculate everything that one is able to. The concept was made by Turing to answer whether the 'Entscheidungsproblem' is computable. The 'Entscheidungsproblem' wants an algorithm that, with a statement as the input will output 'yes' or 'no' based on whether or not the statement is valid. A Turing machine is an idealized version of a computer. It has an infinite amount of memory. This memory has the form of a tape that has an infinite amount of cells on both the left and the right side. Every cell on the tape can hold a single symbol which the creator of the machine could choose. The amount of unique symbols must be finitely many. The machine should also be able to run for an undefined amount of time. The only form of input to the machine are the symbols that one puts on the tape before the machine is started. The only form of output are the contents of the tape at the end of the execution. Every tape cell is by default empty. The machine has a single head on the tape. The machine can read, write or delete a symbol at the position of the head. The 'Entscheidungsproblem' would be computable if there is a machine that could calculate for any given machine that it ends up in an infinite loop. It also would be computable if there is a machine that could calculate whether eventually a certain symbol would be printed on the tape. By the simple design of the Turing machine, Alan Turing was able to reason that neither of both machines could exist. Therefore the 'Entscheidungsproblem' is not computable. Another model in the time of Alan Turing, the lambda calculus, led to the same conclusion. This was published by Alonzo Church slightly before Alan Turing had the chance to publish this. However, the Turing Machine model resembles an actual processor better and therefore has a preference by many.

2.1 Implemented notation

A Turing Machine does not only consists of a tape and its head. The components of a given Turing Machine could be defined as a 7-tuple: $M = \langle Q, \Gamma, \Delta, \Sigma, \delta, q_0, F \rangle$. Turing machines also have states, which the symbol Q represents. The symbol Γ represents the symbols to be written on the tape. For practical reasons the empty cell should be displayed with a symbol too. This symbol is included in Γ and indicated by Δ. Σ is the set of all symbols except for the blank symbol Δ. One state is the initial state. This state is the state the machine is in at the start of the execution, which is defined by q_0. The collection of final states is represented by F. If the Turing machine enters one of these states, the machine halts execution. At last, the way to note what actions the Turing machine should take is by means of transitions, which is represented by δ, the transition function. The transition function decides the possible actions a machine can take. It is represented by a 5-tuple: (p, N, W, D, q). The symbol p is the current state from which the transition could be made. The symbol that should be read at the current position of the head is displayed by N. The written value is represented with W represent the symbol that is written to the tape if this transition is made, or to empty the current tape position or to keep the current value. In my notation, a symbol is always present. If one chooses to erase, the symbol will be Δ and if one chooses to keeps the value, the read symbol will be this symbol. The symbol D represents the direction the tape head moves for one cell. The direction can be left (L), right (R) and stand still (S). Some implementations call stand still 'none'. This direction is not in all implementations, but is in my notation. The state the transition ends in is q. Halting states should be called 'Ha' in my notation. It is not possible to have halt-reject states. Rejecting the input occurs implied upon reading a symbol that has no described transitions in from the current state. If the description follows closely the graph, the transition should be labeled in the format 'N/W, D' (with D one of 'L', 'R' or 'S'). The alphabet used is currently always extracted out of the explicit descriptions of the symbols in the transitions. Symbols in the Turing machine model could be anything that could be written repeatedly. My implementation is limited to the Latin alphabet 'a' to 'z', both uppercase and lowercase. A sequence of characters could also be a symbol in BATREAUX. Then numbers are allowed as well, but they cannot be the first character.

Example Machine

The notation can be illustrated by a Turing Machine that checks whether a string of A's contains an even amount of this letter. The symbol Γ is then $\{a, \Delta\}$. Thus Σ is now $\{A\}$. We can make a Machine with four states. First of all: one we are in upon reading an even amount of A's, and one upon having read an uneven amount of A's. Let's call the former state q_1 and the latter q_2. There is one last state where we end up after the last read A if the amount of A's is even. We can call this state H_a. Also there should be one state to read the first Δ. We can call that state q_i. So now we can say $Q = \{q_1, q_2, H_a, q_i\}$. The symbol q_0 should be q_i because one starts upon having to read a Δ. The state we should end in is H_a, so $F = \{H_a\}$. There should only be four explicit transitions. One by reading a Δ at the start. One by reading an A from state q_1, one by reading an A from state q_2 and one by encountering a Δ from within state q_1. This makes $\delta = \{(q_i, \Delta, \Delta, R, q_1), (q_1, A, A, R, q_2), (q_2, A, A, R, q_1), (q_1, \Delta, \Delta, S, H_a)\}$.
We can now set AAAA on the tape. The tape then looks like ΔAAAA followed by an infinite amount of Δ's. IF we execute this Turing Machine now, we start in q_i. We stand on the first position of the tape(Δ). We follow the first transition in δ. We end up in the second position of the tape(A) in state q_1. We can follow the second transition for we read an A and are in q_1. We end up in the third position of the tape(A) in state q_2. We can follow the third transition for we read an A and are in q_2. We end up in the fourth position of the tape(A) in state q_1. We can follow the second transition for we read an A and are in q_1. We end up in the fifth position of the tape(A) in state q_2. We can follow the third transition for we read an A and are in q_2. We end up in the sixth position of the tape(Δ) in state q_1. We can follow the fourth transition for we read an Δ and are in q_1. We end up in the fourth position of the tape(Δ) in state H_a. In this state the machine accepts the input. For this state, accepting the input means that the string of A's is even.

3 Examples of allowed description structures

Lower/graph level descriptions

In this section I will demonstrate sentences and description structures that could occur in graph level descriptions and are accepted in Batreaux-compiler. The sentences used as examples are borrowed from Appendix A (the description Bolhuis). The notation of transitions used is the explicit label with the form 'R/W, D'. With R the read symbol, W the written symbol and D the direction as a single (uppercase) character. The transition is surrounded by round parentheses. Multiple transitions could be combined by a space or the word ‘and’. The empty cell could both be described as the word delta as well as the Unicode symbol (both uppercase and lowercase). On a physical computer, the tape of the simulated machine cannot be infinite, but it can expand to the right until there is no free memory left on the system that is used for simulation.

Explicit transition

The most obvious way to declare a Turing machine is to describe the source, target and the label of it when it is represented as a graph.

```
From q2, there is also an arrow (delta/delta, L) leading to the state Ha
```

Self referential transitions

Some transitions end up in the same state as before:

```
From q13, an arrow (A/A, L B/B, L) to itself.
```

Step-based state references

When a state is not explicitly named, it can help to describe a part of the figure as steps. Now it is later possible to refer to a state as a point after or before a certain step:

```
From q14, an arrow goes back to itself in six steps.
...
Step 3 arrow (A/a, L) to a new state
...
And from this point, the arrow (B/b, L) to the point after step 3 above
```

Note: it is required to describe all steps. Otherwise the program will fail to generate the Turing machine model correctly.

Higher level descriptions

Here, I will show sentences and description structures that could occur in higher level descriptions and are accepted in Batreaux-compiler. I refer to Appendix D for the full grammar. The example sentences are as they occur in Appendix C (the complex description).

Implicit states

On a Turing machine table all states should be named explicitly. In Batreaux-language it is possible to express transitions and states in the following way:

```
We can also transition to a new state by replacing an ’a’ with a ’b’ and moving to the
 right.
From there, we read a delta that we leave on the tape and move right.
```

This will create 2 states. Both have no name, but a transition ‘a/b, R’ is ending in the first state, and a transition ‘Δ/Δ, R’ goes from the first to the second state.

Implicit transition symbol case/diacriticallity

If a description requires the Turing machine to change diacriticallity or case for a set of symbols, it could be expressed like this:

```
We can also transition to a new state by replacing an ’a’ or ’b’ with a capital letter
 and moving to the right.
```

This will make both the transitions ‘a/A, R’ and ‘b/B, R’. This will also work if one specifies that any letter should be replaced with a capital. Batreaux-compiler will calculate the used alphabet out of all explicit symbol descriptions, and then generate all the required transitions.
The diacriticallity equivalence of this sentence is the following:

```
We can also transition to a new state by replacing an ’a’ or ’b’ with a hatted letter
then moving to the right
```

Also the word ‘accented’ could be used to denote a form of diacriticallity. If one wants to specify a specific symbol with a hat one could also write this like:

```
We can also transition to a new state by replacing a ’c’ hat with a ’c’
then moving to the right
```

Implicit state for a read/replacement loop

Sometimes one wants to describe a part of the Turing machine where it reads symbols M, until it encounters symbols E. One can express this with:

```
We then move over all M to the right until encountering E.
```

Batreaux-compiler will generate the implicit intermediate state that transitions to itself with ‘M/M, R’.

Implicit expressing a transition over multiple lines

One might want to express a transition in multiple lines of text like:

```
We can then read a delta.
We then move left.
```

BATREAUX-compiler shall generate a transition in the form 'Δ/Δ, L' towards a newly generated state. The compiler will calculate whether the current transition can be extended with the information in the new line, or that a new transition is needed. This is done using variables that store the previous active transition and state. These are updated every time a sentence is processed. A line does not have to end with a period. A newline character is then still required however.

Detecting sentence structures in complex sentence structures

Upon sentences that are parsed by rules that may or may not result in non-empty terminals the position of certain sentence structures might vary. By means of searching through the children of the nonterminal until the correct rule name is found, problems that arise with this are resolved. For example: The following two sentences both have the same global structure:

```
from q5 we can read all B's moving left
from q5        read all B's then move left
```

However the pronoun is discarded in the last sentence. The nonterminal that would match that part of the sentence will only match empty string. This nonterminal is therefore removed from the parse tree. In the abstract syntax tree the index of the nonterminal that holds the value of the symbol that is to be read (the B's) will be different for both sentences. Using a specific index might therefore result in wrong strings or crashing behavior of the program. By searching over all children until the nth occurrence is found this problem is solved.

Freedom within the syntaxis

The syntaxis is created with some freedom in mind. Everywhere in the examples where 'we' stands, could also be written 'I' or 'one'. Every word that does not describe a symbol could be written in both lower and uppercase. If a symbol of state name does only contain alpha numeric characters starting with a latin character, quotes around the name are not needed. Words can be replaced by their synonyms. For example the word read could be replaced by 'reads', 'traverses', 'observe', 'read', 'notice', 'notices', 'noticed', 'observes' and 'traverse'. Sentence structures that could start with 'beginning there' do need a 'we', 'I', 'one' or 'it', but the words 'can', 'now', 'currently' and 'also' are optional. Using a comma after a symbol is optional.

4 GOLD

A parser could be created by writing it oneself directly in a programming language. Directly implementing a parser is, however a task that takes time. Also if structural changes are made in the language, this will require to rewrite the parser. Writing a parser could be automated by a tool that is known as a **parser generator**. A tool like that can also be called **compiler compiler**. There are multiple parser generators available, and if one wants to implement a language one can also write the parser itself. “GOLD” [3] is a term to describe the language, tools and parsing methods as implemented by Devin Cook. GOLD is the acronym for “Grammar Orientated Language Developer”. I have ended up with GOLD for a variety of reasons. Many parser generators only output to one or a few programming languages. This makes the grammar bound to only a few languages. GOLD works different. It can output to many programming languages. Also, an alternative parser generator that I have used, “Coco/R” reported conflicts in the grammar at a stage early on. GOLD, by being a LALR parser requires more advanced grammar constructs to end up with reported conflicts (see below for these conflicts).

4.1 Tokens

A complex parser usually is preceded by a lexer. This type of algorithm is also known as ‘scanner’ or ‘tokenizer’. The result of the lexer is a sequence of tokens. A token is a part of the source code which represent a distinct element. In many general purpose programming languages, a single number would result in a single token. Every token has both a type and a value. In the case of a number, the type would generally be ‘number’ or something like that. The value would then be a string containing every digit of the number. The value is formally called ‘lexeme’. From the position where the previous token ended, a lexer could always figure out the next token of a source text by reading new characters that belong to the token. These characters are either part of the token, or not part of any token at all. GOLD uses ‘DFA’ to tokenize. DFA stands for Deterministic Finite Automata. By default, GOLD will automatically ignore whitespace characters. Whitespace will only be used to split tokens. This behavior can directly be used for my language. Also the case sensitivity is disabled. In most cases it doesn’t matter whether a user entered ‘READ’ or ‘read’ in Batreaux. While ‘Identifier’ seem to be a programming language term, still this token is created in Batreaux for the purpose of naming states and symbols. Also a token is defined for tokens that describe the nth case or state.

4.2 LALR

Grammars work by a collection of rules. Every rule consists of a nonterminal rule that can be replaced by nonterminals and terminals. Replacing can never end with a nonterminal, it always will end with a terminal. In my grammar every token ends up as a terminal. This parser works by figuring what following token could be read, after a rule ended. This is checked for every rule. Using that information, a table could be created. Tokens that might be read after a rule is ended are called lookahead tokens or symbols. Every parser type can accept a k number of lookahead tokens. A LALR parser that works with k lookahead tokens is considered a $LALR(k)$ parser. Generally, the number of lookahead tokens is reduced to one single for efficiency reasons in terms of calculation time and memory. LR Parser generators that generate parsers with more than 1 lookahead are considered rarely implemented. LALR parsers operate on the table and the input string of tokens using 4 actions. The parser could 'shift'. This is adding a token in the parse stack. Also it could 'reduce'. This is replacing all tokens on the parse stack by the nonterminal of the accepted parse rule. Sometimes, the parser performs the action 'GOTO'. That is changing to another state. At last the parser could 'accept' if the input is completely parsed, or 'reject' if the input does not follow the grammar rules. A LALR parser is a simplified version of the LR parser. A LR parser does not combine states in configuration set. This way LR parsers have more states.

4.3 Conflicts

From a theoretical point of view, an unambiguous grammar submitted to a parser generator would result in a working parser. In practice, this is not always the case. This is the result of the number of lookahead symbols the parser generator supports. If a situation arises where the k lookahead tokens could no longer decide what rule to complete, a conflict will be reported and the parser generator might not produce a parser in the end. two types of conflicts could be the result of GOLD generating a parser. By working on the grammar I have experienced both types of conflicts.

4.3.1 Shift-Reduce

First, there is the shift-reduce conflict. This conflict arises if a rule can be completed, but it could also be the case that a new token is read and added to the parse stack. By using recent versions of GOLD, the 'shift' action is used in this situations. Forms of the dangling, or hanging else problem are always a shift-reduce conflict (as one can see on the grammar below):

$\langle Id \rangle$ ::= Letter AlphaNumeric*

$\langle Statement \rangle$ ::= if $\langle Id \rangle$ then $\langle Statement \rangle$
| if $\langle Id \rangle$ then $\langle Statement \rangle$ else $\langle Statement \rangle$
| $\langle Id \rangle$:= $\langle Id \rangle$

This conflict had occurred in my grammar at the following point:

$\langle arrowgoes \rangle$ ::= $\langle withaOpt \rangle$ $\langle aoranOpt \rangle$ $\langle arrow \rangle$ $\langle goes\ Opt \rangle$ $\langle there\ Opt \rangle$ $\langle to \rangle$ $\langle state\ id \rangle$ $\langle andfromto \rangle$

$\langle arrowstep \rangle$ ::= $\langle withaOpt \rangle$ $\langle aoranOpt \rangle$ $\langle arrow \rangle$ $\langle annotatie \rangle$ $\langle back\ Opt \rangle$ $\langle tostatOpt \rangle$

Both `<arrowgoes>` and `<arrowstep>` could be expanded from the same nonterminal. `<withaOpt>` could be empty. `<aoranOpt>` could be 'a', 'an' or empty. With the token 'a' read, the parser ends up in the Shift-Reduce conflict.

4.3.2 Reduce-Reduce

Second, a situation might arise where two rules might be reduced at the same time. This kind of errors actually reports always ambiguity. Therefore if GOLD reports these conflicts, no parser is generated. This is an example of a grammar that will result in a reduce-reduce conflict:

$\langle Start \rangle$ ::= $\langle a \rangle$ | $\langle b \rangle$

$\langle a \rangle$ ::= ‘`c`’ ‘`d`’

$\langle b \rangle$ ::= ‘`c`’ ‘`d`’

This grammar is by itself unambiguous and should be resolved by re-evaluating what the goal of the grammar is. This type of conflicts occurred during the development of this grammar in significant lesser amounts than the Shift-Reduce variant.

4.4 Syntax

4.4.1 Character sets

Character sets is the method GOLD uses to describe accepted characters at the point they are presented in a token. Character sets are named and surrounded with '{}'-parentheses. A Character set can be declared using first the name, then the ‘=’ sign, then an other character set or character. This can be followed by ‘-’ (other character set or character) or ‘+’ (other character set or character) as many times as needed. The ‘-’ sign means remove these characters from the set. The ‘+’ sign means to add them. Characters itself are surrounded by brackets (‘[ ]’). Some character sets are already pre-defined in GOLD. I created character sets that could be expressed inside single quotes the following way: `{SQ Chars}      = {Printable} + {HT} - ['']`. I used the same way to define characters inside double quotes. Also I defined the whitespace character set as the pre-defined set, with the exception of newline characters.

4.4.2 Tokens

Tokens are defined by a Regex-like syntax. ‘*’, ‘+’, ‘|’ and ‘?’ all have their Regex-related meaning. Single quotes can be used around characters to remove their special meaning. I used tokens to define quoted strings. As an example: `SingleQuoteStr = ['']{SQ Chars}*['']`. Also I defined numbers as: `{Number}{Number}*`.

4.4.3 Productions

To describe the grammar I have in mind so GOLD could parse it correctly, a syntax (or grammar) is needed. The language of this grammar is called the “GOLD Meta-Language”. The syntax is very close to the Backus-Naur Form. It differs only in expressing how one could read no tokens at all. An empty part of the ‘|’(pipe) symbol or ‘<>’ could be written to declare a null-able rule, while in Backus-Naur form two double quotes would be used. My grammar can be defined in four types of non-terminals. First, we have non-terminals that stand for synonyms and optional words. Rules for these I do call ‘synonyms-rules’. Take for example the word ‘begin’. Where that word is used, also the words ‘starts’, ‘start’ and ‘initiate’ could be used. The rule I used for that is: `<begin> ::= 'begin' | 'starts' | 'start' | 'initiate'`. If a non-terminal could end up in the parse tree with no tokens read, the non-terminal will end with ‘Opt’. An example of this is `<with Opt> ::= 'with'|'with' 'the' 'annotation'|'with' 'the' 'inscription'|`. Secondly some non-terminals describe parts of sentences. Rules that describe these do I call ‘pseudo-rules’. Take for example the sentence ‘And then we go to q3’. We can replace ‘q3’ with ‘a new state’, ‘the state after step 4’ and ‘this new state’. These four descriptions are about the same concept. They all describe a state. I expressed this in my grammar as:

$\langle$*state id*$\rangle$ ::= $\langle$*sidprefix*$\rangle$ $\langle$*state*$\rangle$ $\langle$*comma Opt*$\rangle$ Identifier $\langle$*comma Opt*$\rangle$
  | $\langle$*the Opt*$\rangle$ $\langle$*state*$\rangle$ $\langle$*comma Opt*$\rangle$ Identifier $\langle$*comma Opt*$\rangle$
  | $\langle$*the Opt*$\rangle$ $\langle$*thecurloc*$\rangle$ $\langle$*comma Opt*$\rangle$
  | $\langle$*the Opt*$\rangle$ 'next' $\langle$*comma Opt*$\rangle$
  | $\langle$*the Opt*$\rangle$ $\langle$*state*$\rangle$ $\langle$*after at*$\rangle$ 'step' $\langle$*numberID*$\rangle$ $\langle$*abovbefor*$\rangle$
  | $\langle$*sidprefix*$\rangle$ $\langle$*state*$\rangle$ $\langle$*comma Opt*$\rangle$
  | Identifier $\langle$*comma Opt*$\rangle$
  | $\langle$*cur loc*$\rangle$ $\langle$*comma Opt*$\rangle$

Also some sentences end in multiple ways. These are pseudo-rules as well. Also there are non-terminals that describe a complete sentence. These I call 'actual-rules'. One actual rule I use is: `<from read>::= <from> <state id> <personOpt> <there Opt> <can isOpt> <also Opt> <read> <other Opt> <allof1sym> <then etc> <untileOpt>` At last there are some non-terminals to describe how sentences could be combined. While this is in the grammar, these rules have no function in BATREAUX. This is because BATREAUX will split sentences itself, and input every sentence individually in GOLD. The advantage of this approach is that if some sentences are not in BATREAUX-language, still the compiler could do something with the results. These rules, with the exception of `<rule Act>` are:

$\langle$*rule*$\rangle$ ::= $\langle$*rule Act*$\rangle$ $\langle$*end statement*$\rangle$

$\langle$*Program*$\rangle$ ::= $\langle$*rule*$\rangle$ $\langle$*nl Opt*$\rangle$ $\langle$*Program*$\rangle$ |

$\langle$*Start*$\rangle$ ::= $\langle$*nl opt*$\rangle$ $\langle$*Program*$\rangle$

$\langle$*nl Opt*$\rangle$ ::= NewLine $\langle$*nl Opt*$\rangle$

5 Semantic Analysis

After parsing the input of the user, some form of semantic analysis is done. In BATREAUX, the parse tree is modified to be more in an abstract syntax tree format. Some non-terminals could be the start of every rule. I have therefore defined these in the grammar rules in front of the rule non-terminal. In case these universal non-terminals are not empty, I move them to within the rule. An example of this is displayed in Figure 2.
Also many places in the grammar recursive non-terminals can make long edge chains (Figure 3 for example), while the actual structure that I want to parse has the form of a list. These chains are converted to a single node with multiple children. The converted example is shown in Figure 4.
It can also occur that natural language descriptions of lists contain the word 'and', which is here removed. Some non-terminals are optional. In most cases, if these non-terminals are reduced to an empty string, they can be discarded from the parse tree, which will happen in this step.

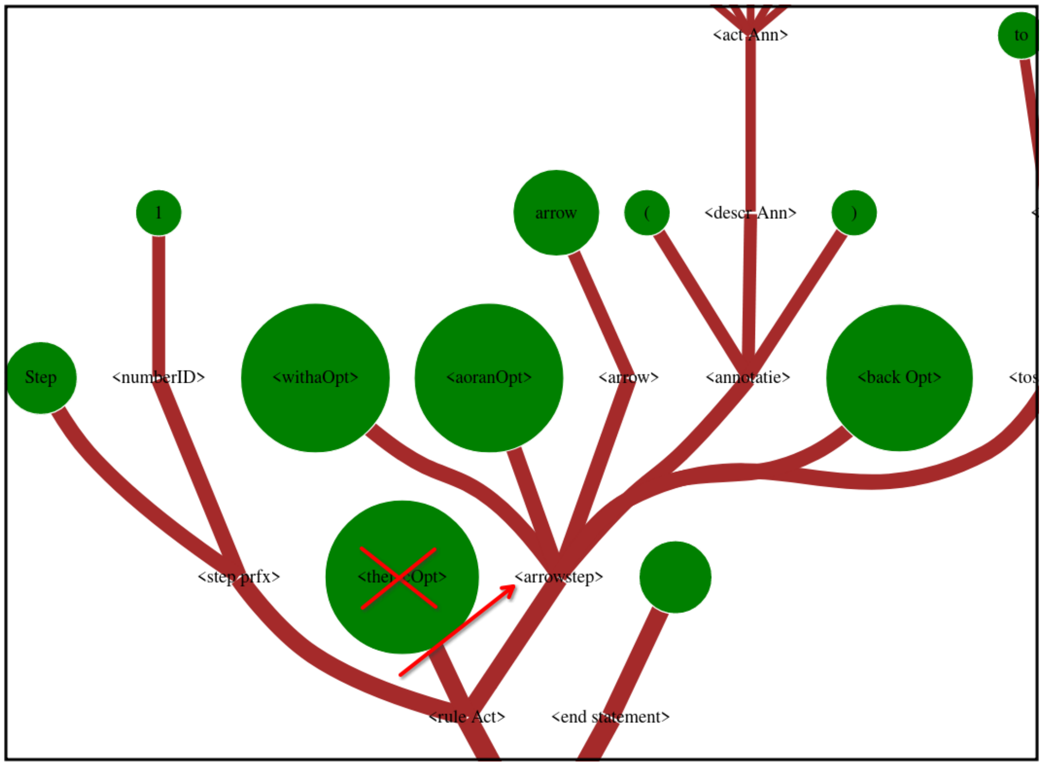


Figure 2: The <step prfx> non-terminal will become a child of the actual rule, for it is not empty. The other shown non-terminal(<then cOpt>) is empty and will be removed from the tree.

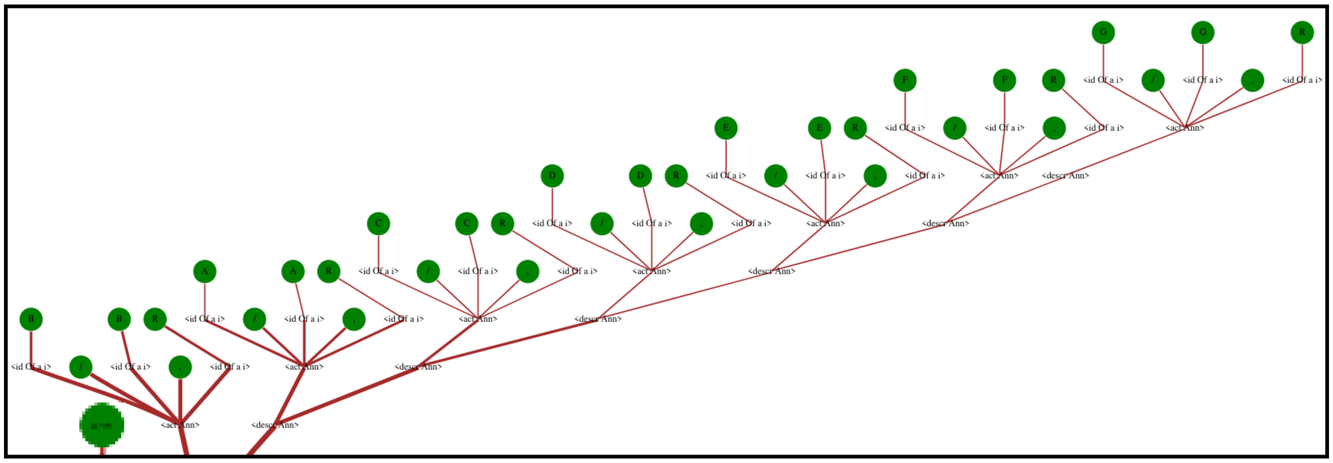


Figure 3: The <descr Ann> non-terminal represents a list. Due to how GOLD and most parsers work, this non-terminal is recursively repeated.

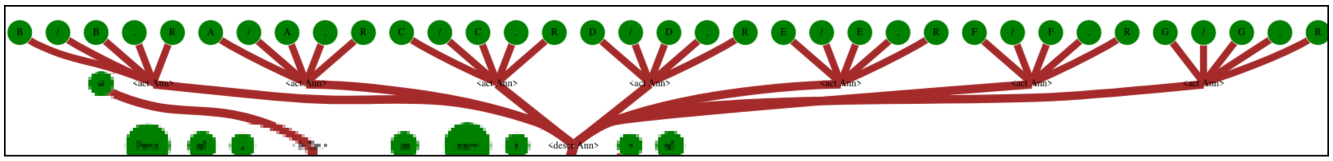


Figure 4: The <descr Ann> non-terminals are converted to a single node with multiple children.

6 Code generation

Intermediate Code

The intermediate code of Batreaux contains two lists. One list contains states, the other contains transitions. The first state in the list is the starting state. The transition list could be considered the most close to intermediate code. Intermediate code in an average compiler could be considered as three-operands code; Every element in the list has (at most) three operands. The transition list in Batreaux consists of transitions with three operands as well: the source state, the target state and a list with annotations. Every annotation is an object that contains the read and write symbol of the transition and the direction. Every symbol has a name which is a string, a list of properties and whether to interpret the symbol figuratively. This structure is chosen, because it allows a simple generation of transitions in the form of "replace all lowercase characters with a capital". At first, the description will be converted into a transition with a single annotation. Later on, the annotations list will be replaced with n number of annotations, one for each lowercase character that occurs in the alphabet of this Turing machine.

Generation

Different sentence structures are defined by a different 'actual-rule'. During the generation step of Batreaux, every sentence in the description is processed on the basis of these nonterminals. Some of them share the same code, but most do not. During the generation of intermediate code, the direction of a transition might not always be explicitly stated. In the rare cases that this happens, an implied direction is stored. This direction is the last direction that was explicitly stated. After a sentence is processed, a following sentence might make use of the transition or state that is created or modified by this first one. This can be done by some variables that store these values. At the beginning of the processing of a sentence, the 'current state' will be stored in the variable for the previous line state. The same applies for the 'current transition'. Sometimes a transition is described in multiple sentences. In these cases it is checked whether the described elements of the transition are not yet filled in in the last intermediate code. Implicit directions are then considered not filled in yet. If all the elements are not yet filled in, the description is applied to the previous transition instead of a newly created transition. Because some parts of the grammar(pseudo-rules) are shared over multiple 'actual-rules', the processing of these pseudo-rules are done in functions. For example the function 'symbolsVanuitSymbolsId' processes the nonterminals 'symbolsid' and 'symbolssi'. These nonterminals expand to a list of symbols in singular form and plural form, respectively. The function returns a list of symbol objects. Also there is the function 'staatVanuitStateId' that processes the description of a state. In case this state is not yet created in the Turing machine object, this new state is created. The function ends with returning a state object.

Machine Code

Before any machine code is generated, the intermediate code is modified. Transitions that contain figuratively-interpreted symbols are converted to transitions that can be interpreted literally. This is done by first calculating the alphabet out of all literal symbols. Then we know all the symbols that a figurative symbol could be replaced for. The machine code that Batreaux generates is in the form of C code files. These files can then be automatically compiled into native Linux executables. The C code is partially a translation of the VB.NET implementation of the Turing Machine interpreter. Two versions of machine code are created currently. The first version creates at the start of the program all the states and transitions in the same format as the intermediate code objects. At the start of the program the states, symbols and transitions are added to simple list structures. This is different from the second version. In the second version of the machine code, data structures are optimized. Now the states, symbols and transitions are saved in a array which is only once allocated. Only annotation information is copied in this version. Also much of the properties that where available in the VB.NET implementation of the intermediate code is removed. Also comparisons are made with the index of the symbol and not with the actual string contents of the symbol, which should result in faster comparisons.

7 Comparison Algorithm

Making comparisons between Turing machines can help to retrieve information. To do this within a shorter time span, an algorithm is made that compares Turing machines in structural similarity. The algorithm will not help in all cases, but can help in some. The algorithm works by comparing states of both machines. For two machines with both more than 10 states and 5 transitions on average per state, the number of comparisons to be made can grow quadratically. The algorithm tries to reduce the number of comparisons by not always giving the best match.

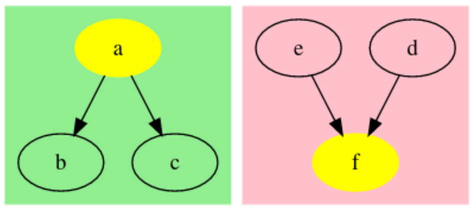
7.1 Other similarity scores

Several algorithms that would calculate similarity over graphs exists. The three most common are: Jaccard similarity, Overlap similarity and Sørensen-Dice similarity. All of these take a relation into account of the amount of matching neighbourhood, and the size of the (shared) neighbourhood. On comparing Turing Machines exact matches of nodes are relevant, and nodes that differ can usually just be considered different. This is especially the case on Turing Machines that are supposed to be the same. Therefore these similarity scores are not taken into account in this algorithm. Still this algorithm gives useful results. It is possible to implement some form of these similarity scores later on in the algorithm in case the results do not satisfy. A minimal value should be chosen carefully to give workable results.

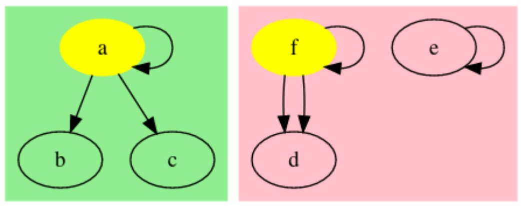
7.2 The algorithm similarity scores

We compare various characteristics of a state in this algorithm (as displayed in Figure 5):

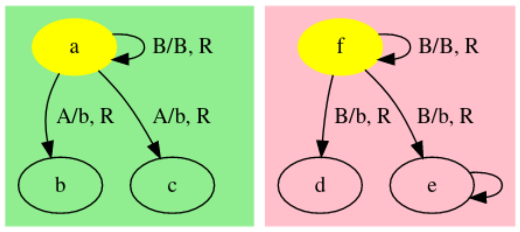
- The number of transitions
- The number of outgoing, ingoing and self-referential transitions
- the number of outgoing and incoming transitions in relation to an other state
- The transitions itself, in relation to other states



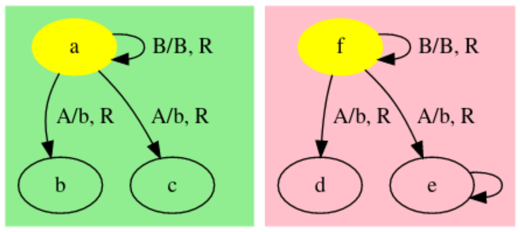
(a) The yellow states have the same number of transitions. The labels are not drawn for a more simple graph. However, the number of in- out- and self referential transitions are not equal.



(b) The number of in- out- and self referential transitions are the same for the yellow states. The labels are not drawn for a more simple graph. However, the number of in- out- and self referential transitions in relation to an other state are not equal.



(c) The number of in- out- and self referential transitions in relation to an other state are equal for the yellow states. However, the transitions itself are not.



(d) The yellow states will be considered each others alter ego by the algorithm. All checked characteristics are equal.

Figure 5: Levels of similarity of the two yellow states ‘a’ and ‘f’

To compare a pair of states in this algorithm first these more global comparisons are made (in the order I have introduced them). This results in making more comparisons than one should have to on comparing two states. However this would filter out incompatible states early on, which should save time on Turing machines that have similarity.
The number of transitions a state has is saved during the algorithm so it can be queried faster. Upon comparing, lower amounts of possible pair combinations are drawn before higher amounts. If this is equal, or within the current accepted error, the algorithm continues with checking the similarity of the pair up to the point that it considers them alter-egos.
At some point in the algorithm I work with the term ‘state profile’. A ‘state profile’ is the information about the amount of transitions grouped by connected state and orientation. The last comparison that I make is on ‘state profiles’ where the information is not the number, but the actual transitions. If the states are found to be a match, then they are removed from the structure that stores the scores of pair combinations. Also direct neighboring states are directly tried to match. This is because it is expected that these have generally a higher chance of having a perfect match. Matched states are also marked by the property ‘alterEgo’(that defines the alter-ego) of the state. This property is now a pointer to the matching state. This is done both ways. The matching states can no longer be proposed for a pair again. After the comparison algorithm cannot find more working

matches within error, the effective allowed error is raised until it would either exceed the actual allowed error, or a new pair could be proposed again.
When no pair can be proposed within the actual allowed error, both diagrams can be combined in one. This new diagram has stored for every state and transition whether that one is one that occurs in both, in the original diagram, or the modified one. Upon displaying this combined diagram will draw modified states and transition green, and original states and transitions blue. Combined ones and texts are always displayed black.

7.3 Other comparison algorithms

There are several methods to compare graphs. There is for example the Weisfeiler Leman graph isomorphism test. This test does, however only work if both Turing Machines are completely isomorph. Also one could implement a color refinement algorithm like Weisfeiler Leman graph isomorphism. This would result in a slower algorithm, for the coloring step would take more time, however it should give better results in some situations where a part of the structure of the Turing Machine is repeated. The Turing Machines presented in this thesis are not big enough for that. It is also possible to create a mapping between nodes using a Graph kernel method. This would result in a similarity matrix. A value higher than the highest value in the matrix can be substracted by the similarity matrix to produce a cost matrix. This cost matrix can then be applied to the Hungarian Algorithm. This would then result in the optimal mapping on basis of the similarity matrix. One could use the Floyd-Warshall algorithm as the kernel to calculate the similarity matrix. The time complexity of this is however $\Theta(|V|^3)$, where V is the number of nodes. This is larger than the complexity of my comparison algorithm. The accuracy would be better on repeating Turing Machines.

8 ChatGPT

Large Language Models(LLMs) are techniques that allow the computer to process natural language. These models are not perfectly able to parse natural language, but can still be a useful tool. GPT is a series of language models created by openAI. GPT stands for Generative Pre-trained Transformer. The recent versions of GPT are generally considered to belong to the best language models that are available. ChatGPT is an online tool that uses GPT3.5 or later to allow conversations in 'Markdown'. GPT3 models are trained on Wikipedia, but most of all archived websites of the internet. The GPT models are able to react with programming code. The latest models of GPT4 have an option to query certain websites to retrieve real time information.

8.1 Personal usage

I have used ChatGPT to assist me in writing code for Batreaux in multiple ways. Some example descriptions on which I have worked were in Dutch. ChatGPT is about the same level at translating (high-resource) natural languages as any other transition programs [6]. I have used ChatGPT to translate these Dutch descriptions. Also ChatGPT is used in writing trivial code for Batreaux. The parts that it contributed to are mainly writing functions that can programmatically run a program. The way in which this is done is something that takes relatively a lot of code in Java and Visual Basic.NET to Python. Also in other functions that help format output I have used ChatGPT. I have not used ChatGPT for actual algorithms that compile code or evaluate Turing machines. ChatGPT is also not able to assist in solving concrete cases of grammar conflicts and problems.

8.2 Solution for Turing machine compilation

In this thesis I focus mainly on a formal language. My reasons for not focussing on machine learning approaches are that these are considered impossible to correct when created. Usually the only way a model is improved on machine learning is to generate a new one with more or better training data. A model created using machine learning will also use more resources than a traditional human-written computer program. ChatGPT is tested on some Turing machine descriptions. In all of these descriptions ChatGPT makes errors. Both the 3.5 model of ChatGPT and the newer, improved GPT4.0 model was used. The latter could be tested using Microsoft Bing. The tested language that is known as language XX $= \{xx | x \in \{a, b\}^*\}$. A reference on how the output should look like is Figure 6. I have compared GPT4 with the simple description, the higher level text and also the Bolhuis description. Alongside language XX, also the languages AnB2n $= \{a^n b^{2n} | n \geq 0\}$ and Aeven $= \{a^n | \texttt{n is even and greater or equal 0}\}$ have been tested. The descriptions of these language could be considered higher level. All of the comparisons are made with an allowed error of 1. This error is theoretical defined as the amount of transitions that do not match between the compared states. In practice my comparison algorithm might be stricter and more prone to result in bigger error values. The GPT output is created by querying the GPT model. For Bing has a character limit lower than some descriptions, a description is split in multiple parts and the individual parts are queried. The output of GPT is then combined in the end. Repetitive rows are removed. My query to ChatGPT is the line: `Create a table of the following description of a Turing machine.` Followed by the description in between triple quotes (`"""`). If the the output contained occurrences

of state names repeated more than they should, I queried again with as addition that GPT should name the new states. If GPT outputs 'a-z' or 'A-Z' in the output table, the program substitutes this with a figuratively symbol with properties uppercase or lowercase. If GPT outputs some extra explanation after a state in parenthesis, this extra explanation is removed. The '-' sign is interpreted as the 'stay' movement in GPT's output.
The comparison between the AEVEN reference graph and the GPT4 output (converted to graph format) displays but a single distinct element(Figure 7). The states q4 and q5 (as called in the reference graph) are merged by GPT. These states are implicit states. They are not named in the description. Their existence is implied. In the comparison of the description of language 'AnB2n' it is shown that the states are correctly generated. However, the direction in 3 positions in different from the reference graph. The directions where implied in the description, and the direction to 'Halt Accept' would not matter. Still the other 2 directions should be left. If these are stay, the algorithm would not work correctly. All of the descriptions of language XX inputted in GPT result in incorrect graphs. Figure 8 for example lacks a transition that ends in a new state in which an uppercase character is read and one goes to left (zoomed in on differences with Figure 9). Sometimes this is because of states that should be separate, but end up merged in the GPT output. The most explicit example of this is GPT3.5 with the Bolhuis description(Figure 10). After describing a certain number of transitions, the algorithm only describes two types of transitions. Those from q14 to itself and from q14 to a state called 'new state'. GPT is in the end a transformer of texts. Upon graph drawing it is important to store a fragment of information at which state one currently is. This storage, possibly in combination with generating state names itself, seemed absent or failing in GPT. This could be explained for GPT being a generative transformer and not a traditional algorithm. Sometimes the errors of GPT could not resemble this pattern and are more random. In conclusion GPT could not generate graphs correctly of a description.

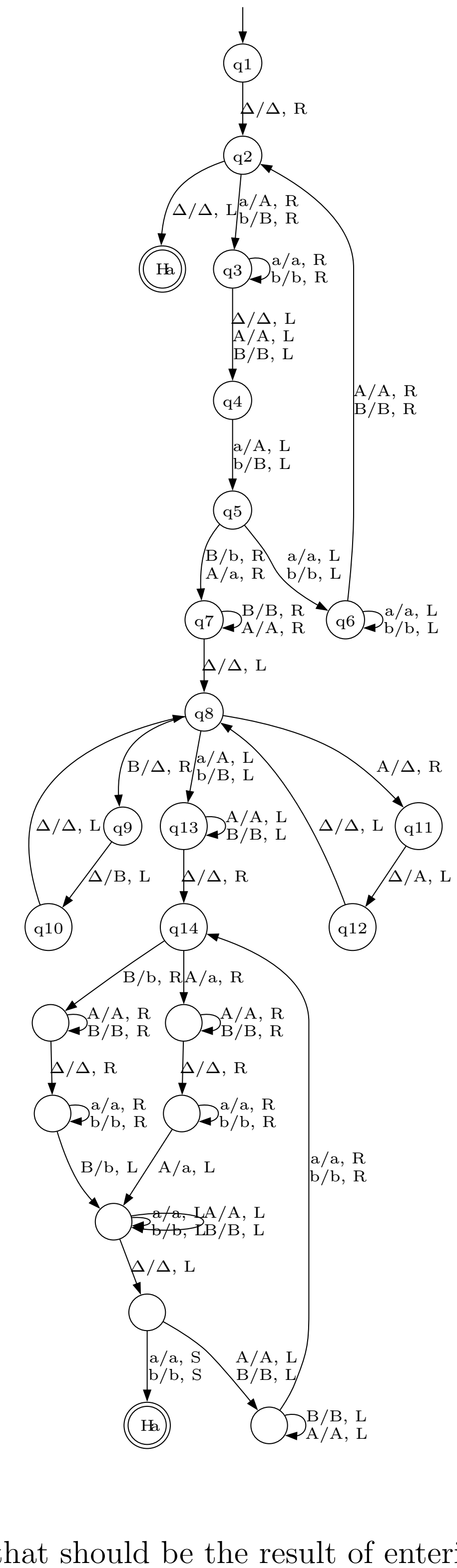


Figure 6: This is the diagram that should be the result of entering the (language XX) language descriptions in (Chat)GPT. In practice the two 'H_a'/Halt-accept states could be merged to one in the output. The same applies for the two self-referential arrows 2 states before 'H_a'. The part of the diagram until encountering q_8 moves the tape head to the middle of the tape, and therefore also checks whether the contents of the tape is even. In this process the contents of the tape is put in uppercase.

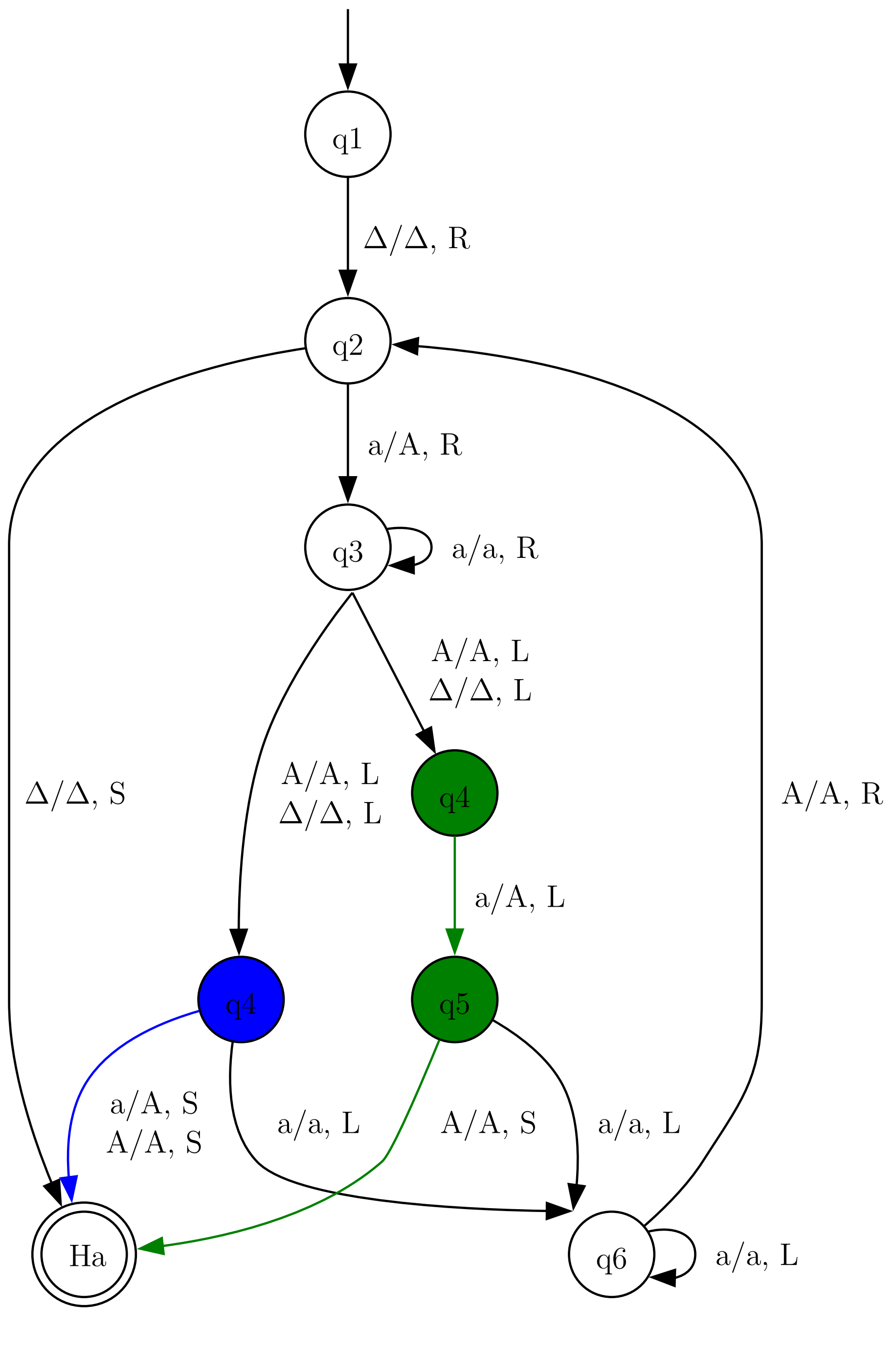


Figure 7: ChatGPT4 output of a description of language Aeven, in comparison to the output of this description by Batreaux. Unmerged reference states and transitions are colored green. Unmerged ChatGPT states are colored blue. Combined states and transitions are colored black.

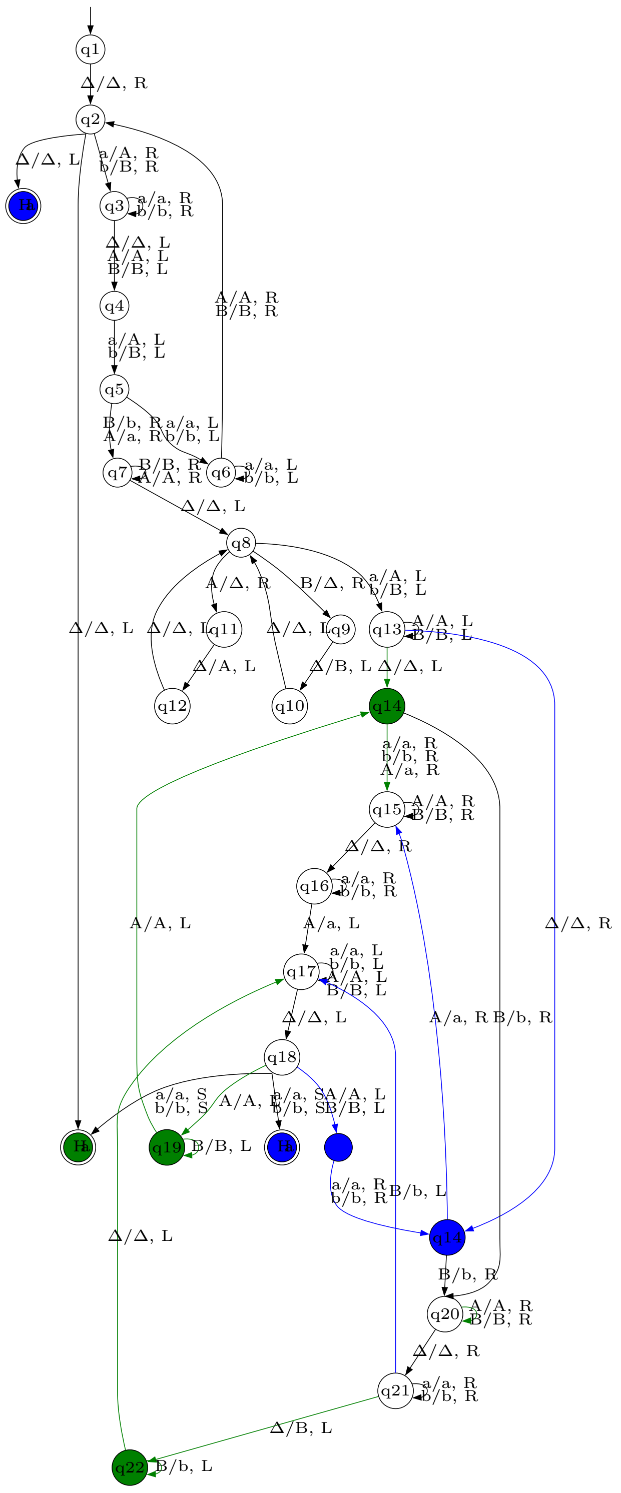


Figure 8: ChatGPT4 output of the Bolhuis description of language XX, in comparison to the output of description Bolhuis by BATREAUX. Unmerged reference states and transitions are colored blue. Unmerged ChatGPT states are colored blue. Combined states and transitions are colored black. The Halt-accept states are not merged due to an error in the comparison algorithm I presented in the previous chapter.

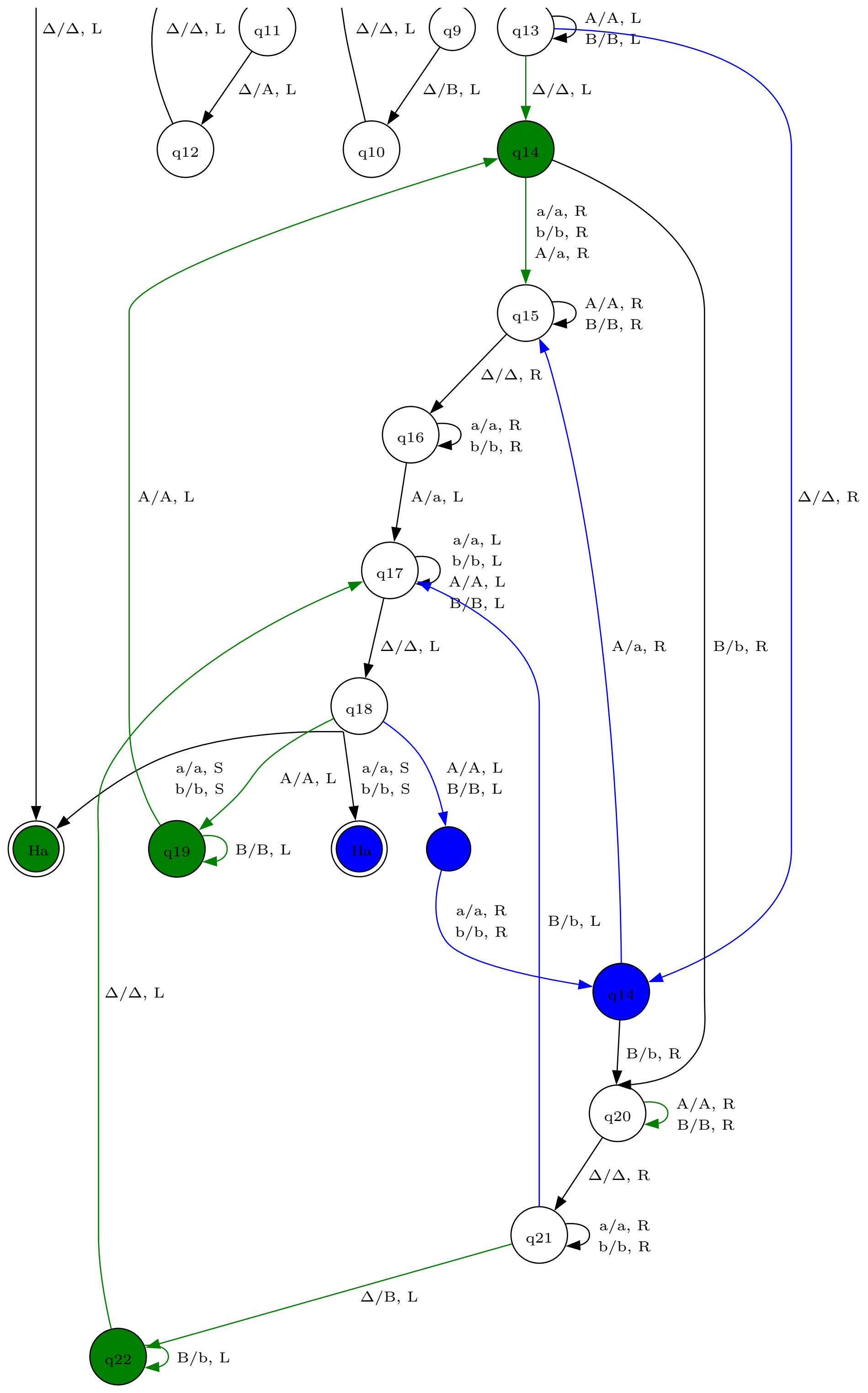


Figure 9: Zoom in of Figure 8. The Upper blue halt accept and the green halt accept are different because of a flaw in the comparison algorithm.

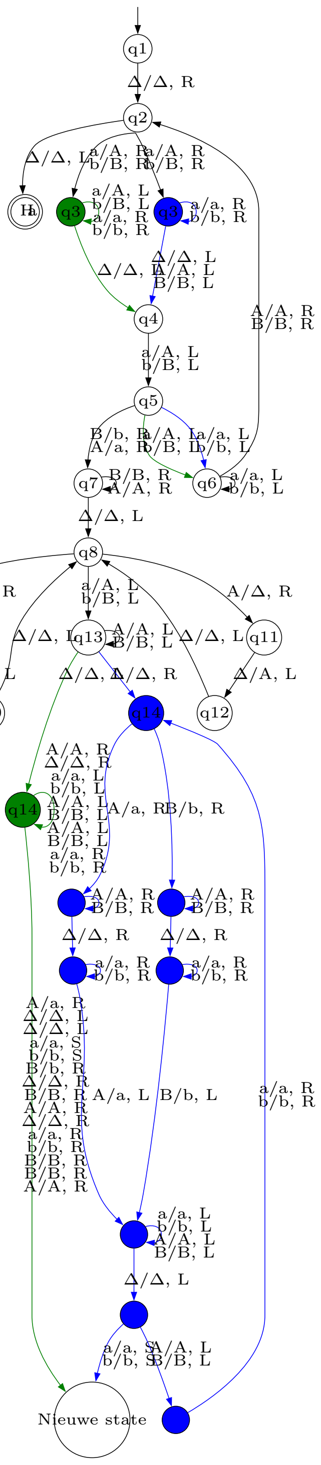


Figure 10: ChatGPT3.5 output of the Bolhuis description of language XX, in comparison to the output of description Bolhuis by BATREAUX. Unmerged reference states and transitions are colored blue. Unmerged ChatGPT states are colored green. Combined states and transitions are colored black.

9 Compiler results

Alongside the (chat)GPT method, also the created compiler is tested upon functionality. Using the description in Appendix C, it is possible to recreate the graph (Figure 11). A single H_a state remains unmerged. This is the result of the Bolhuis description working with two Ha states, and the complex description working with one. The program does not merge Ha-states yet, although doing so would improve the performance of Turing Machines. The complex description gives the Turing Machine 37 transitions and 22 states. All the tests that will follow are executed on a system with the following specifications: AMD A6-9220 RADEON R4 CPU, Memory Caches (sum of all): L1d: 64 KiB (2 instances), L1i: 128 KiB (2 instances), L2: 2 MiB (2 instances) and RAM memory with 2133MHz clock speed and a width of 64 bits. Simulating this Turing Machine for 3249902 steps (by means of an input of 1800 symbols) takes about 7 minutes and 44,53 seconds to run (including interpretation of the Turing Machine). Simulating it for 512 steps (by means of a string of length 20 which repeats itself once) while also making an image per step takes about 1 minute and 43.98 seconds. Simulating this same scenario without making the images takes under a second (0.732) to run.

9.1 Generating code

The compiler BATREAUX compiles the description in Appendix C into 77 lines of C code. Together with the static C code, this can generate an executable of 36760 bytes. This executable is able to execute 3249902 steps within 4 minutes and 24.37 seconds. With printing the tape contents(up to the part that contains only deltas), the execution takes 48 minutes and 27.07 second.
The previous version of the C code generator outputs 71 lines. The binary executable is 48888 bytes. Executing 3249902 steps will take 4 minutes an 25.49 seconds. With printing the tape contents, the execution takes 41 minutes and 51.94 seconds.
The description for the language $a^n b^{2n}$ results in the Figure 12. The generated C program can process $n = 10000$ in 2 minutes and 13.03 seconds.

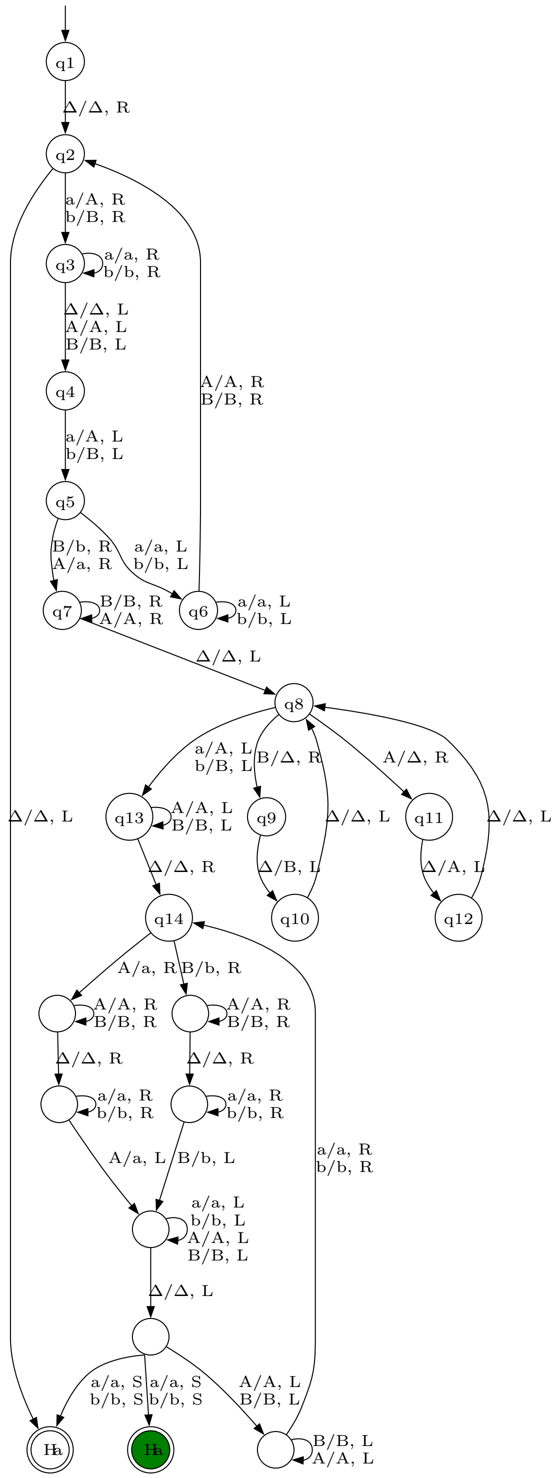


Figure 11: Batreaux (version 30-05-2023) output of the complex description of language XX, in comparison to the reference diagram on a basis of Bolhuis. Unmerged reference states and transitions are colored blue. Unmerged Batreaux states are colored green. Combined states and transitions are colored black.

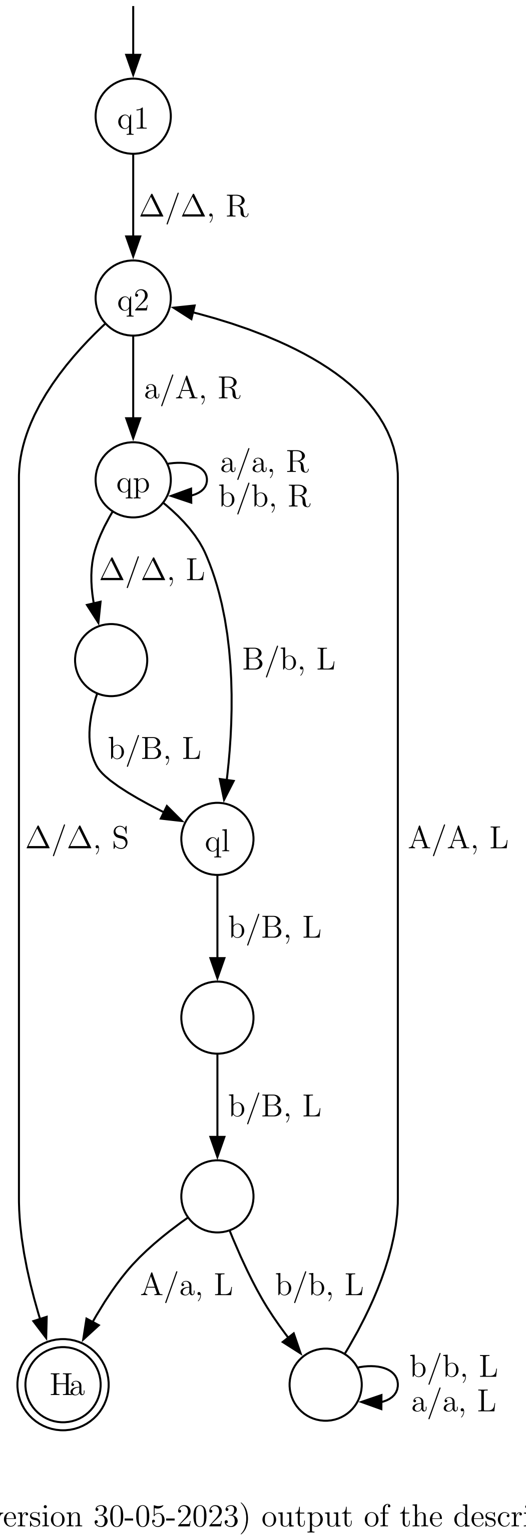


Figure 12: Batreaux(version 30-05-2023) output of the description of language $a^n b^{2n}$.

10 Conclusions and Further Research

In this paper I have described how a compiler could be created on the basis of GOLD to convert a subset of natural language to a Turing machine description. I have also shown that GPT3 and GPT4 cannot execute this task correctly, mostly because of not storing information. By making use of a generated parser and following the steps a compiler should make, complexity could be minimized for the given task. Also the use of an Intermediate code that is well thought out helps a clean implementation. Finally, the processing of 'pseudo-rules' in separate functions, instead of doing the processing completely on the rule itself, reduces complexity and duplicate code. Further research is needed in the area of the language the program supports itself. The expressed grammar can get more complicated than needed. This may be solved by first creating a version of GOLD that supports LALR(n) grammar instead of only LALR(1). If the grammar is then processed using this modified version of GOLD the grammar can retain its level of complexity. The language can then be extended with more sentences. Also more forms of output can be added to the program. Output to format that online Turing Machine simulators can process would be the most relevant of this category.

References

[1] dreampuf (Github username actual name unknown). *Graphviz Online*. 2020. URL: https://dreampuf.github.io/GraphvizOnline/ (visited on 03/28/2024).

[2] K. Bolhuis. "Beschrijving figuur". In: *personal communication (from a well-informed outsider)* (2023).

[3] Devin D. Cook. *Design and development of a grammar oriented parsing system*. 1997. URL: http://goldparser.org/download.htm (visited on 03/28/2024).

[4] DevToolsDaily. *DevTools Daily*. 2020. URL: https://www.devtoolsdaily.com/graphviz/ (visited on 03/28/2024).

[5] Antoine du Hamel. *Viz.js playground*. 2021. URL: https://aduh95.github.io/viz.js-playground/ (visited on 04/24/2024).

[6] Wenxiang Jiao et al. *Is ChatGPT a Good Translator? Yes With GPT-4 As the Engine*. 2023. (Visited on 03/28/2024).

[7] A. M. Turing. "On computable numbers, with an application to the Entscheidungsproblem". In: *Proceedings of the London Mathematical Society* (1936). DOI: https://doi.org/10.1112/plms/s2-42.1.230.

A Bolhuis Description

This is the description for Language XX is created by the well-informed outsider Bolhuis, translated into English. The graph that has been shown upon creating this description is Figure 13

```
... The first states are all called q. After q14, they no longer have names.
I start at the top left of a page. An arrow goes to q1, and then an arrow (delta/delta, R) goes to q2. From q2, there is also an arrow (delta/delta, L) leading to the state Ha. The state Ha has a double enclosure. It could mean that it does not play a role in the state figure or is not involved in the interaction of the following q states.
From q2, another arrow (a/A, R b/B, R) goes towards a third state, q3, from there an arrow (delta/delta, L A/A, L B/B, L) goes towards state q4, from there an arrow (a/A, L b/B, L) goes towards q5, from there an arrow (a/a, L b/b, L) goes towards q6, and from there, an arrow (A/A, R B/B, R) goes back to q2.

So it seems that something is happening with those q states continuously, but it also returns to the beginning.
However, in the meantime, there are some detours at some of the states.
From q3, there is an arrow going back (a/a, R b/b, R) to q3, and from q6, there is an arrow going back (a/a, L b/b, L) to q6.
From q5, an arrow departs towards a more complex figure of q states.
From q5 (B/b, R A/a, R) to q7.
From q7, there is an arrow (B/B, R A/A, R) to q7.
From q7 (delta/delta, L) to q8, from there (B/delta, R) to q9, from there an arrow (delta/B, L) to q10, and from there (delta/delta, L) back to q8.
From q8, there is also an arrow (A/delta, R) to q11, and from there an arrow (delta/A, L) to q12, and from there (delta/delta, L) back to q8.

From q8 (a/A, L b/B, L) to q13.
From q13, an arrow (A/A, L B/B, L) to itself.

From q13, an arrow (delta/delta, R) to q14.

From q14, an arrow goes back to itself in six steps.
Step 1 arrow (A/a, R) to a new state
Then, pointing to itself with an arrow (A/A, R B/B, R)
Step 2 arrow (delta/delta, R)
Pointing to itself with an arrow (a/a, R b/b, R)
Step 3 arrow (A/a, L)
Pointing to itself there with an arrow (a/a, L b/b, L)
And again with an arrow (A/A, L B/B, L)
Step 4 arrow (delta/delta, L)
From this, also an arrow (a/a, S b/b, S) to a separate state Ha
Step 5 arrow (A/A, L B/B, L)
Self-referencing an arrow (B/B, L A/A, L)
And in step 6 with an arrow (a/a, R b/b, R) back to q14
From q14, another arrow (B/b, R) to a next point
Referring to itself with (A/A, R B/B, R)
From this point, with (delta/delta, R) to the next
Referring to itself with (a/a, R and b/b, R)
And from this point, the arrow (B/b, L) to the point after step 3 above
```

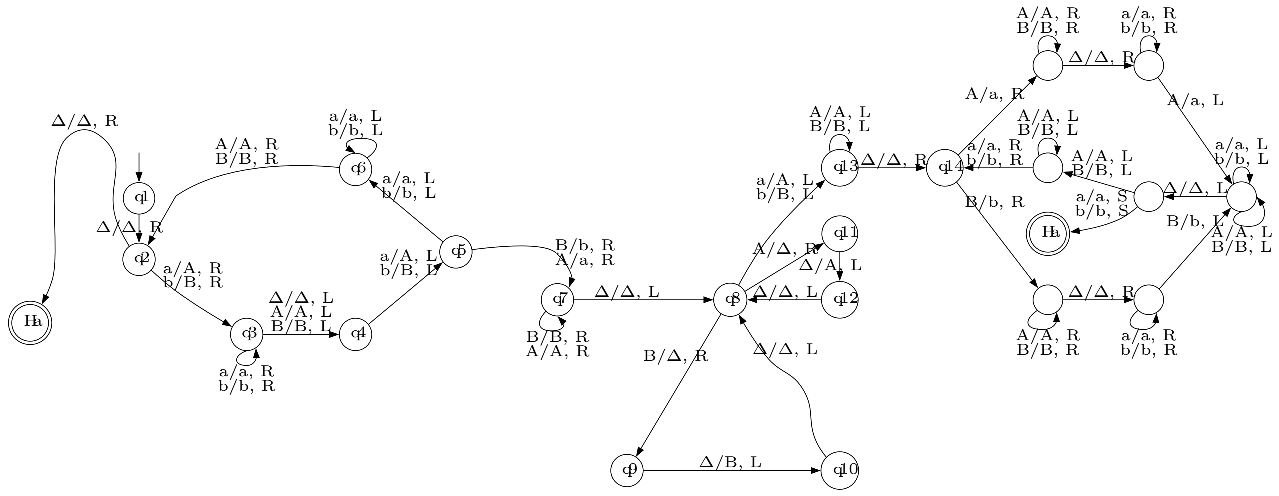


Figure 13: The graph that was been shown to Bolhuis.

B Simple Description

This is the description for Language XX that is somewhat more formal than the Bolhuis description.

```
...
Draw an arrow pointing to q1. From q1, draw an arrow to the next circle labeled
    'Δ/Δ, R'. From there, there are 2 arrows. One points to Ha with the label '
    Δ/Δ, L'. The other points to the next circle with 'a/A, R' and 'b/B, R'
    written on it. This next circle refers back to itself with the labels 'a/a,
    R' and 'b/b, R'. It also points to the next circle, labeled with 'Δ/Δ, L', '
    A/A, L', and 'B/B, L'. This circle, in turn, points to a new circle with 'a/
    A, L' and 'b/B, L' written on it. Let's call this circle q5. From q5, there
    are 2 arrows. The first points to a new circle, which refers back to itself.
     It has 'a/a, L' and 'b/b, L' written on it. This circle also refers to the
    circle we discussed earlier, the one that q2 points to. This is done with
    the labels 'A/A, R' and 'B/B, R'. From q5, another arrow departs with the
    labels 'B/b, R' and 'A/a, R'. Let's name the new circle it leads to q7.

q7 refers to itself with the labels: 'B/B, R' and 'A/A, R'. q7 also refers to a
     new circle with the label 'Δ/Δ, L'. Let's call this circle q8, which refers
     to 3 circles. Firstly, it points to a circle with the label 'B/Δ, R'. This
    points to the next one with the label: 'Δ/A, L'. That, in turn, refers back
    to q8. Also, q8 points to another new circle, with the label 'A/Δ, R'. This
    points to the next one with the label: 'Δ/A, L'. That, in turn, refers back
    to q8 with the text 'Δ/Δ, L'. Lastly, q8 also refers to another new circle,
    with the labels 'a/A, L' and 'b/B, L'. This circle has an arrow pointing
    back to itself. The label reads: "A/A, L and B/B". This circle also refers
    to the next one with "Δ/Δ, R". Let's name this circle q14.

q14 has 2 outgoing arrows. Firstly, one with 'A/a, R'. The incoming one refers
    to itself with 'A/A, R' and 'B/B, R'. It also refers to a new one with 'Δ/Δ,
     R'. The incoming one refers to itself with 'a/a, R' and 'b/b, R' and to the
     next one with 'A/a, L'. Let's call this next one qv. qv refers to itself
    twice and once to the next one. One of the self-references contains the
    labels: 'a/a, L', 'b/b, L'. The other contains the labels: 'A/A, L' and 'B/B
    , L'. qv also refers to the next one with the text 'Δ/Δ, L'. This next one
    refers to Ha with 'a/a, S' and 'b/b, S' and to the next one. This happens
    with labels 'A/A, L' and 'B/B, L'. The next one refers to itself with labels
     'A/A, L' and 'B/B, L'. It also refers to q14, which happens with the labels
     'a/a, R' and 'b/b, R'. q14 has another arrow. This one leads to a new
    circle with the label 'B/b, R'. The receiving circle refers to itself with
    the labels 'B/B, R' and 'A/A, R'. It also refers to the next one with the
    label 'Δ/Δ, R'. This next one has an arrow pointing to itself with the
    labels 'a/a, R' and 'b/b, R'. There is also an arrow from this one to qv. It
     contains the text 'B/b, L'.
```

C Complex Description

This is the description for Language XX that is describing in a more abstract manner. It doesn't explicitly call all the states. States q_3 and q_4 aren't named explicitly for example.

```
We start in q1. From there, we read a delta that we leave on the tape while
    moving right to a new state q2. From there, we can move by reading delta we
```

```
   keep on tape to the left to Ha. We can also transition to a new state by
   replacing an 'a' or 'b' with a capital letter then moving to the right. We
   then move over all small 'a's and 'b's to the right until encountering a
   capital letter or a delta. We then move to the left. From this new state, we
    replace 'a' or 'b' with a capital letter, then move to the left. The state
   we are in is then referred to as q5.
From q5, we can read a lowercase letter, then move left. From there, we read
   small letters, then move left until encountering a capital letter. We then
   move right to q2. From q5, we can also read a capital letter then replace it
    with a lowercase letter. We then move to the right. From this state, we
   read capital letters until encountering a delta. We then move left. The
   state we are in is then referred to as q8.
From there, we can read a B which we replace with a delta. We then move to the
   right on the tape. We can then read a delta, which we replace with a capital
    letter B. We then move left. We then read a delta, move left, and end up
   back in q8.
From there, we can also read an A, which we replace again with a delta. We then
    move to the right on the tape. We can then read a delta, which we replace
   with a capital letter A. We then move left. We then read a delta, move left,
    and end up again in q8.
There is one other connection that departs from q8. This reads a lowercase
   letter then replaces it with a capital letter, then moves left. it reads all
    capital letters until encountering a delta. Then it moves to the right to a
    new state. Let's call this state q14.
From q14, we can replace a capital letter b with a lowercase b while moving to
   the right. We now read capital letters until encountering a delta. Then we
   read small letters until encountering a capital letter B. We replace it with
    a lowercase letter. We then move left. Let's call the current state qm. qm
   reads over capital letters or small letters to the left until it encounters
   a delta. From this state, we can move to Ha by reading a lowercase letter.
   We then remain stationary. We can also read a capital letter then move to a
   new state to the left. From there, we read capital letters until
   encountering a small letter. We then move to the right to q14.
From q14, we can replace a capital letter a with a lowercase a while moving to
   the right. We now read capital letters until encountering a delta. Then we
   read small letters until encountering a capital letter A. We replace it with
    a lowercase letter, and end up back in qm. We then move left.
```

D Grammar of Batreaux

This is the grammar for Language Batreaux version 0.8.

```
"Name"     = 'Batreaux'
"Author"   = 'David N.'
"Version"  = 'Version 0.8a'
"About"    = 'A grammar for Turing Machine descriptions'
"Case Sensitive" = 'false'

"Start Symbol" = <Start>

! ------------------------------------------------
! Character Sets
! ------------------------------------------------
```

```

{WS}            = {Whitespace} - {CR} - {LF}
{String Chars} = {Printable} + {HT} - ["]
{SQ Chars}     = {Printable} + {HT} - [’’]

! -------------------------------------------------
! Terminals
! -------------------------------------------------

Whitespace     = {WS}+
NewLine        = {CR}{LF} | {CR} | {LF}

Identifier     = {Letter}{AlphaNumeric}*
NumberCountWord= {Number}{Number}*’st’ | {Number}{Number}*’th’
Digits         = {Number}{Number}*
SingleQuoteStr = [’’]{SQ Chars}*[’’]
DoubleQuoteStr = ["]{String Chars}*["]
SingleQuoteStrs= [’’]{SQ Chars}*[’’]’s’
DoubleQuoteStrs= ["]{String Chars}*["]’s’
Quote          = [’’]

! -------------------------------------------------
! Rules
! -------------------------------------------------

! synonym rules
<nl Opt> ::= NewLine <nl Opt>       !Zero or more
          |  !Empty
<end statement> ::= NewLine | ’.’

! <nl opt> removes blank lines before first statement
<read>     ::= ’observe’ | ’read’ | ’notice’ | ’notices’ | ’reads’ | ’noticed’
   | ’observes’ | ’traverse’ | ’traverses’
<in>       ::= ’in’ |
<arrow>    ::= ’arrow’|’connection’ !|’transition’
<arrows>   ::= <arrow>| ’arrows’|’connections’|’transitions’
<goes>     ::= ’goes’|’leads’|’leading’|’is’|’going’|’points’|’pointing’|’
   referring’
<move>     ::= ’move’|’walk’|’walking’|’walks’|’moves’|’moving’|’transition’|’
   go’|’going’|’goes’
<departs>  ::= ’moves’ ’away’|’departs’
<is Opt>   ::= ’is’  |
<goes Opt> ::= <goes> |
<then Opt> ::= ’then’|
<replace>  ::= ’replace’|’substitute’|’replaces’|’replaced’|’substitutes’|’
   substituted’
<with Opt> ::= ’with’|’with’ ’the’ ’annotation’|’with’ ’the’ ’inscription’|
<withaOpt> ::= ’with’|
<withorin> ::= ’with’|’in’
<withto>   ::= ’with’|’to’
<from>     ::= ’from’ | ’beginning’ ’in’ | ’starting’ ’from’|’starting’ ’in’
<also Opt> ::= ’also’ | ’likewise’ | ’similary’| ’as’ ’well’ | ’again’ |
<the Opt>  ::= ’the’ |
<other Opt>::= ’other’| ’another’ |
<has>       ::= 'has' | 'contains'
<to>        ::= 'to' | 'towards'
<fromthere>::= 'from' 'that' 'point' | 'from' 'there' | 'starting' 'there' |'
    beginning' 'there'
<state>     ::= 'ball'|'blub'|'round'|'state'|'point'|'circle'|'node'
<countword>::= 'first'|'second'|'third'|'fourth'|'fifth'|'sixth'|'seventh'|'
    eighth'|'nineth'
            |'tenth'|'eleventh'|'twelfth' |'thirteenth'|'fourteenth'|'fifteenth
    '|'sixteenth'|'seventeenth'|'eighteenth'|'nineteenth' |'twentieth' |
    NumberCountWord
<all>       ::= 'all'|'every'|'all' 'of' 'the'
<back Opt> ::= 'back' |
<again Opt>::= 'again'|
<begin>     ::= 'begin' | 'starts' | 'start' | 'initiate'
<enclosure>::= 'enclosure'|'cirkel'
<enclosurs>::= 'enclosures'|'cirkels'
<a or an>  ::= 'a' | 'an'
<aoranOpt> ::= <a or an>|
<aantheOpt>::= <a or an>|'the'|
<comma Opt>::= ',' |
<there Opt>::= 'there'|
<and Opt>  ::= 'and'|
<it Opt>   ::= 'it' |
<arrow Opt>::= <aantheOpt> <arrow> |
<andcomOpt>::= 'and' ',' | 'and'|
<numberID> ::= 'one'|'two'|'three'|'four'|'five'|'six'|'seven'|'eight'|'nine'|'
    ten'
            |'eleven'|'twelve'|'thirteen'|'fourteen'|'fifteen'|'sixteen'|'
    seventeen'|'eighteen'|'nineteen'|'twenty'|Digits
<new>       ::= 'new' | 'separate' | 'brand' 'new' | 'next'
<colonOpt> ::= ':'|
<after at> ::= 'after'|'at'|'before'
<abovbefor>::= 'above'|'before'|
<draw Opt> ::= 'draw'|
<then cOpt>::= 'then'|'then' ','|
<refto its>::= 'self' 'referencing' | 'self-referencing'
<personfrm>::= 'we' | 'I' | 'one'
<now Opt>  ::= 'now'| 'currently' |
<person it>::= <personfrm> | 'it'
<p it Opt> ::= <personfrm> | 'it' |
<personOpt>::= <personfrm> |
<keep>      ::= 'keep' | 'keeps' | 'leave' | 'leaves'
<kept>      ::= 'kept' | 'left'
<tape>      ::= 'the' 'tape' | 'tape' | 'memory' | 'the memory' | 'there'
<leftright>::= 'left' | 'right'
<can>       ::= 'can' | 'is' 'able' 'to' | 'might' | 'could'| 'will'
<until>     ::= 'until' !|'to' 'the' 'point'
<are>       ::= 'are' | 'am' | 'is'
<referred> ::= 'referred' 'to' 'as' | 'called' | 'named'
<lets Opt> ::= 'let'Quote's' |
<call>      ::= 'call' | 'name' | 'annotate' | 'describe'
<stay>      ::= 'stay' | 'stays' | 'stand' | 'stands' | 'remain' 'stationary'
<encounter>::= 'encounter'|'encounters'|'encountering'|'came' 'across'|<read>

```

```


! pseudo rules
! S for stand...
<id Of a i>::= Identifier | 'a' | 'i' | 's'
<id Of an> ::= Identifier | 'a' | 'an'
<act Ann>  ::= <id Of a i> '/' <id Of a i> ',' <id Of a i>
<descr Ann>::= <act Ann> <descr Ann> | <act Ann> 'and' <descr Ann> | <act Ann>
! TODO... <qoute Ann>::= '"'<act Ann> <descr Ann> | <act Ann>
<annotatie>::= '(' <descr Ann> ')'
<sidprefix>::= 'a' <countword>  | 'a' <new>
<thecurloc>::= 'current' 'location' | 'current' 'point'
<cur loc>  ::= 'this' | 'this' 'point' | 'this' 'new' 'state' | 'this' 'state'
    | 'here'
<curlocimp>::= <cur loc> |  'the' 'current' 'state' | 'the' 'current' 'point'
<statehax> ::= <the Opt> <state> | <sidprefix> <state>
<state id> ::= <sidprefix> <state> <comma Opt> Identifier <comma Opt>
            |  <the Opt> <state> <comma Opt> Identifier <comma Opt>
            |  <the Opt> <thecurloc> <comma Opt>
            |  <the Opt> 'next' <comma Opt>
            |  <the Opt> <state> <after at> 'step' <numberID> <abovbefor>
            |  <sidprefix> <state> <comma Opt>
            |  Identifier <comma Opt>
            | <cur loc> <comma Opt>
<andfrom X>::= <then Opt> <arrow Opt> <annotatie> <goes Opt> <back Opt> <to> <
    state id>
<andfromto>::= 'and' <andfrom X> |
<and los>  ::= <from> <state id> <there Opt> <goes Opt> <aoranOpt> <arrow> <
    goes Opt> <back Opt> <annotatie> <to> <state id>
<ftt Mult> ::= <andcomOpt> <fromthere> <comma Opt> <andfrom X> <ftt Mult> | <
    andcomOpt> <fromthere> <comma Opt> <andfrom X>
<in Xsteps>::= 'in' <numberID> 'steps'
<fttoflos> ::= <ftt Mult> | <andcomOpt> <and los> |
<doubl enc>::= 'a' 'double' <enclosure> | 'two' <enclosurs>
<ordarrspc>::= <arrow> <goes> 'back'| <arrow> <goes> 'forwards'| <arrow> !
    directional arrow specification
<steps go> ::= 'halt' 'accept'|'halt' 'reject'|<state id>
<step prfx>::= <and Opt> <in> 'step' <numberID> <colonOpt> <and Opt> | <and Opt
    >
<tostatOpt>::= <to> <state id> |
<which Opt>::= 'which' | 'that' |
<can isOpt>::= <can> | <is Opt>
<still Opt>::= 'still' |
<all Opt>  ::= 'all' |
<quote str>::= DoubleQuoteStr | SingleQuoteStr
<quotestrs>::= DoubleQuoteStrs | SingleQuoteStrs


<symbol id>::= <id Of an> | <id Of an> <id Of an> | <id Of an> <id Of an> <id
    Of an> <id Of an>
            | <id Of an> <id Of an> <id Of an> | <id Of an> <id Of an> <id Of
    an> <quote str>
            | <quote str> | <id Of an> <quote str> | <id Of an> Identifier <
    quote str>
```

```
            | <quote str> Identifier | <id Of an> <quote str>  Identifier
            | <id Of an> Identifier <quote str> Identifier
<symbols i>::= <id Of an> | <id Of an> <id Of an> | <id Of an> <id Of an> <id Of an> |
            | <quotestrs> | <id Of an> <quotestrs> | <id Of an> <id Of an> <quote str>
      | <quote str>
            ! all ’a’ hats, all uppercase ’a’ hats
            | <quote str> Identifier | <id Of an> <quote str>  Identifier

<symbolsid>::= <symbol id> | <symbol id> ’,’ | <symbol id> ’or’ <symbolsid> | <symbol id> ’,’ <symbolsid>
<symbolssi>::= <symbols i> | <symbols i> ’,’ | <symbols i> ’or’ <symbolssi> | <symbols i> ’and’ <symbolssi> | <symbols i> ’,’ <symbolsid>
<kw Opt>   ::= <which Opt> <personfrm> <keep> ’on’ <tape> | <which Opt> ’is’ <kept> ’on’ <tape>
            |  <which Opt> <personfrm> ’do’ ’not’ ’replace’ ! todo: don’t and won’t
            |  <which Opt> <personfrm> ’will’ ’not’ ’replace’
            |  <which Opt> <personfrm> ’replace’ <also Opt> ’with’ <symbolsid>
            |  ’replace’ ’it’ <also Opt> ’with’ <symbolsid>
            |  <which Opt> ’is’ <also Opt> ’replaced’ ’with’ <symbolsid> |
! to dir/state id...
<ontapeOpt>::= ’on’ <tape> |
<lr dir>   ::= <to> <the Opt> <leftright> <ontapeOpt> <comma Opt> | <leftright> <ontapeOpt> <comma Opt>
<lr dirOpt>::= <lr dir> |
<tostateid>::= <to> <state id> | <and Opt> ’end’ ’up’ <back Opt> <again Opt> ’in’ <state id>

<andwhile> ::= ’then’ | ’while’
<andmove>  ::= <andwhile> <personOpt> <move> | <move>
<andlr dir>::= <andmove> <lr dir>
<readsuOpt>::= <andmove> <lr dir> | <andmove> <lr dir> <tostateid> | <andmove> <tostateid> | <andmove> <tostateid> <lr dir> |

! move by reading X to Y
! move to Y by reading X
<mtbr>     ::= ’by’ ’reading’ <symbolsid> | ’if’ <personfrm> <read> <symbolsid>
<replacing>::= ’by replacing’ <symbolsid> <withto> <symbolsid>
            | ’if’ <personfrm> <replace> <symbolsid> <withto> <symbolsid>
<replread> ::= <mtbr> <kw Opt> | <replacing>
<ft Opt>   ::= <fromthere> <comma Opt> |
<canalsOpt>::= <can> <also Opt> |
<from prfx>::= <ft Opt> <person it> <now Opt> <also Opt> <canalsOpt>
<untilenc> ::= <until> <p it Opt> <encounter> <symbolsid>
<untileOpt>::= <untilenc> |
<allof1sym>::= <all> <symbolssi> | <symbolsid>


<then etc> ::= ’then’ <replace> ’it’ <withto> <symbolsid>| <andlr dir>|’then’ <replace> ’it’ <withto> <symbolsid> <andlr dir>
      | <which Opt> <personfrm> <replace> ’with’ <symbolsid> | <which Opt> <personfrm> <replace> ’with’ <symbolsid> <andlr dir> |
<fm pos>   ::= <lr dir> | <replread> | <tostateid>
                        |  <lr dir> <replread> | <replread> <lr dir>
            |  <lr dir> <tostateid>| <tostateid> <lr dir>
            | <replread> <tostateid>|<tostateid> <replread>
            | <lr dir> <replread> <tostateid>| <tostateid> <replread> <andlr
   dir>
            | <replread> <tostateid> <lr dir>| <lr dir> <tostateid> <replread>
            | <tostateid> <lr dir> <replread>| <replread> <lr dir> <tostateid>



! actual rules
<fromread> ::= <from prfx> <then Opt> <read> <other Opt> <allof1sym> <kw Opt> <
   readsuOpt> <untileOpt>
<from repl>::= <from> <state id> <personOpt> <there Opt> <can isOpt> <also Opt>
    <replace> <symbolsid> <withto> <symbolsid> <andlr dir>
! TODO: read *all* symbols (plural) or symbols singular
<arrowgoes>::= <withaOpt> <aoranOpt> <arrow> <goes Opt> <there Opt> <to> <state
    id> <andfromto>
<drawarrow>::= 'draw' <aoranOpt> <arrow> <goes Opt> <there Opt> <to> <state id>
    <andfromto>
<fromxtoy> ::= <from> <state id> <personOpt> <there Opt> <can isOpt> <also Opt>
    <draw Opt> <aantheOpt> <other Opt> <ordarrspc> <annotatie> <also Opt> <goes
    Opt> <back Opt> <to> <state id> <fttoflos>
<from step>::= <from> <state id> <personOpt> <there Opt> <can isOpt> <also Opt>
    <draw Opt> <aantheOpt> <other Opt> <ordarrspc>  <to> <steps go> <in Xsteps>
<x has y>  ::= <state id> <it Opt> <has> <doubl enc>
<fromxga a>::= <from> <state id> <with Opt> <annotatie> <to> <state id> <
   fttoflos>
<arrowstep>::= <withaOpt> <aoranOpt> <arrow> <annotatie> <back Opt> <tostatOpt>
<point to> ::= <goes Opt> <to> <state id> <there Opt> <withaOpt> <arrow Opt> <
   annotatie>
<self ref> ::= <refto its> <there Opt> <withaOpt> <arrow Opt> <annotatie>
<againrule>::= 'again' <there Opt> <goes Opt> <withaOpt> <arrow Opt> <annotatie
   >
! High level state




<start in> ::= <ft Opt> <personOpt> <begin> <withorin> <state id>
<from read>::= <from> <state id> <personOpt> <there Opt> <can isOpt> <also Opt>
    <read> <other Opt> <allof1sym> <then etc> <untileOpt>
<from move>::= <from> <state id> <personOpt> <there Opt> <can isOpt> <also Opt>
    <move> <to> <state id> <replread>
<frommove> ::= <from prfx> <then Opt> <move> <fm pos>
<moveuntil>::= <from prfx> <then Opt> <move> 'over' <all> <symbolssi> <lr dir>
   <untilenc>
<weaminOpt>::= <personfrm> <are> 'in' |
<isorcan>  ::= <can> 'be' | 'is'
<idofqoute>::= <quote str> | Identifier
<isrefered>::= <statehax> <weaminOpt> <isorcan> <then Opt> <referred> <
   idofqoute>
<callstate>::= <lets Opt> <call> <curlocimp> <idofqoute>
```

```
<staystill>::= <from prfx> <stay> <still Opt> | <from prfx> 'then' <stay> <
    still Opt>
<conn from>::= 'there' <are> <numberID> <other Opt> <arrows> <which Opt> <
    departs> 'from' <state id>


<transit>  ::= <state id> <read> <symbolsid> <then etc>

<tostidOpt>::= <tostateid> |
<repl rule>::= <from prfx> <replace> 'it' <withto> <symbolsid> <tostidOpt>
<stateread>::= <state id> <read> 'over' <all Opt> <symbolsid> <lr dirOpt> <
    untilenc>

<rule Act>::=  <step prfx> <then cOpt> <arrowgoes>
            |  <step prfx> <then cOpt> <fromxtoy>
            |  <step prfx> <then cOpt> <fromxga a>
            |  <step prfx> <then cOpt> <x has y>
            |  <step prfx> <then cOpt> <from step>
            |  <step prfx> <then cOpt> <arrowstep>
            |  <step prfx> <then cOpt> <point to>
            |  <step prfx> <then cOpt> <againrule>
            |  <step prfx> <then cOpt> <self ref>
            |  <step prfx> <then cOpt> <drawarrow>

            |  <step prfx> <then cOpt> <start in>
            |  <step prfx> <then cOpt> <fromread>
            |  <step prfx> <then cOpt> <from move>
            |  <step prfx> <then cOpt> <frommove>
            |  <step prfx> <then cOpt> <moveuntil>
            |  <step prfx> <then cOpt> <from repl>
            |  <step prfx> <then cOpt> <from read>
            |  <step prfx> <then cOpt> <isrefered>
            |  <step prfx> <then cOpt> <callstate>
            |  <step prfx> <then cOpt> <staystill>
            |  <step prfx> <then cOpt> <conn from>
            |  <step prfx> <then cOpt> <transit>
            |  <step prfx> <then cOpt> <repl rule>
            |  <step prfx> <then cOpt> <stateread>

<rule>     ::= <rule Act> <end statement>
<Program>  ::= <rule> <nl Opt> <Program> |
<Start>    ::= <nl opt> <Program>
```