Uniwersytet
im. Adama Mickiewicza
w Poznaniu

Wydział Matematyki i Informatyki

Bohdan Bondar, Marcin Jałoński, Aleksander Mendoza-Drosik
Numer albumu: sS432778,s434701,s434749

Solomonoff - kompilator transduktorów skończenie stanowych

Solomonoff - Finite state tranducer compiler

Praca inżynierska na kierunku **informatyka**
napisana pod opieką
dr Bartłomieja Przybylskiego

Poznań, luty 2021

Poznań, 23 stycznia 2021 r.

Oświadczenie

Ja, niżej podpisana **Bohdan Bondar, Marcin Jałoński, Aleksander Mendoza-Drosik**, studentka Wydziału Matematyki i Informatyki Uniwersytetu im. Adama Mickiewicza w Poznaniu oświadczam, że przedkładaną pracę dyplomową pt. *Solomonoff - kompilator transduktorów skończenie stanowych* napisałam samodzielnie. Oznacza to, że przy pisaniu pracy, poza niezbędnymi konsultacjami, nie korzystałam z pomocy innych osób, a w szczególności nie zlecałam opracowania rozprawy lub jej części innym osobom, ani nie odpisywałam tej rozprawy lub jej części od innych osób. Oświadczam również, że egzemplarz pracy dyplomowej w wersji drukowanej jest całkowicie zgodny z egzemplarzem pracy dyplomowej w wersji elektronicznej. Jednocześnie przyjmuję do wiadomości, że przypisanie sobie, w pracy dyplomowej, autorstwa istotnego fragmentu lub innych elementów cudzego utworu lub ustalenia naukowego stanowi podstawę stwierdzenia nieważności postępowania w sprawie nadania tytułu zawodowego.

[TAK/TAK/TAK] wyrażam zgodę na udostępnianie mojej pracy w czytelni Archiwum UAM

[TAK/TAK/TAK] wyrażam zgodę na udostępnianie mojej pracy w zakresie koniecznym do ochrony mojego prawa do autorstwa lub praw osób trzecich

Streszczenie

Słowa kluczowe: klasa

Abstract

Nam dui ligula, fringilla a, euismod sodales, sollicitudin vel, wisi. Morbi auctor lorem non justo. Nam lacus libero, pretium at, lobortis vitae, ultricies et, tellus. Donec aliquet, tortor sed accumsan bibendum, erat ligula aliquet magna, vitae ornare odio metus a mi. Morbi ac orci et nisl hendrerit mollis. Suspendisse ut massa. Cras nec ante. Pellentesque a nulla. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Aliquam tincidunt urna. Nulla ullamcorper vestibulum turpis. Pellentesque cursus luctus mauris.

Keywords: klasa

Tu możesz umieścić swoją dedykację.

Spis treści

Rozdział 1

Introduction

This project focuses on research in the field of automata theory and inductive inference. The main product of our work is the "Solomonoff" regular expression compiler for finite state transducers. Plenty of research has gone into development of the theory behind this system. As a result the transducers contain several features not know before.

The most innovative achievements is the lexicographic arctic semiring of weights, specialized adaptation of Glushkov's construction for subsequential transducers and the most significant flagship feature - built-in support for inductive inference and machine learning of transducers. Thanks to the cooperation with LearnLib and Dortmund University, Solomonoff supports learning algorithms such as RPNI and several of it's derivatives. Solomonoff contributes its own more specialized for transducers inference. We implemented OSTIA for efficient learning of deterministic transducers. For nondeterministic ones we developed our own OFTIA algorithm, that was not known before.

All those features together make Solomonoff a unique library that stands out from all the alternatives. We support most of the features of UNIX regexes, including look-aheads and look-behinds, which is unusual for automata-based regex engine. The key that allows Solomonoff for doing that is the possibility of emulating look-aheads with careful placement of transducer outputs. As a result Solomonoff can compete and do much more than existing projects such as RE2 developed by Google, BRICKS automata developed at Aarhus University or even to certain extent with Pearl/Java based regular expression engines. Another, much stronger competitor for Solomonoff is the OpenFST project developed by Google. Their Thrax grammars are capable of doing most of the things that Solomonoff can and they also support probabilistic automata. OpenFST is much older and more established in the scientific community. They support a lot more features that were developed by scientists, by the course of many years. Solomonoff cannot compete with this level of sophistication, but perhaps, what might seem like a limitation, is in fact our strongest advantage. Solomonoff focuses on functional transducers and enforces this property at compilation time. Any arising nondeterministic output is automatically rejected as an error. This allows Solomonoff to perform a lot more optimisations, the automata are smaller and their behaviour is more predictable. Moreover, lexicographic weights allow for precise disambiguation of nondeterministic paths whenever necessary and their most important advantage is that lexicographic semiring is not commutative and hence it does not "propagate" throughout entire automaton. This only increases Solomonoff's robustness and allows for (exponentially) smaller automata, without sacrificing predictability of the regular expression. On th contrary, probabilistic weights in Thrax, make the whole system, more heavyweight, unpredictable and difficult to maintain. The results are especially palpable when comparing our benchmarks. Solomonoff was written in Java and Thrax uses C++, but despite this our compiler is several magnitudes more efficient. We performed efficiency tests on a large corpus of linguistic data (dictionary with 6000 records). Solomonoff compilation times were around 2 seconds, whereas Thrax took 19 minutes. Solomonoffs automata were also much smaller, as thanks to Glushkov's construction, there is a 1:1 relationship between size of regular expression and size of transducer. As a result Thrax's transducer takes of 6336K of RAM, whereas Solomonoff only takes 738K. Execution time for such large corpora was about 10 miliseconds in Solomonoff (which is roughly the same performance as using Java's HashMap), while Thrax took about 250 miliseconds. One might argue that, such great differences are achievable, only because Thrax supports a lot more features. OpenFST uses epsilon transitions, while Solomonoff does not implement them and instead all automata are always and directly produced in epsilon-free form. OpenFST performs operations such as sorting of edges, determinization, epsilon-removal and minimization. Solomonoff always has all of its edges sorted and it doesn't need a special routine for it (which makes for additional performance gains), it has no epsilons to remove and it does not need determinization procedure, because it can pseudo-minimise nondeterministic transducers, using heuristics inspired by Kameda-Weiner's NFA minimization. One could, half-jokingly, summarize that the difference between Thrax and Solomonoff is like that between "Android and Apple". Thrax wants to support "all of the features at all cost", whereas Solomonoff carefully chooses the right features to support. We believe this approach will be our strongest asset, that will make our compiler a serious alternative to the older and more established OpenFST. An additional strength that favours Solomonoff over OpenFST is that we support inductive inference out-of-the-box, require no programming (regular expressions are the primary user interface and Java API is minimal and optional) and we provide automatic conversion from Thrax to Solomonoff, so that existing codebases can be easily migrated.

The characteristic feature of Solomonoff, is that its development focuses on the compiler and regular expressions instead of library API. Everything can be done without writing any Java code. It also allows the developers for much greater flexibility, because the internal implementation can be drastically changed at any time, without breaking existing regular expressions. There is very little public API that needs to be maintained. As a result backwards-compatibility is rarely an issue.

The primary philosophy used in implementing this library is the top-down approach. Features are added conservatively in a well thought-through manner. No features will be added ad-hoc. Everything is meant to fit well together and follow some greater design strategy. For comparison, consider the difference between OpenFst and Solomonoff.

- OpenFst has Matcher that was meant to compactify ranges. In Solomonoff all transitions are ranged and follow the theory of (S,k)-automata. They are well integrated with regular expressions and Glushkov's construction. They allow for more efficient squaring and subset construction. Instead of being an ad-hoc feature, they are well integrated everywhere.
- OpenFst had no built-in support for regular expression and it was added only later in form of Thrax grammars, that aren't much more than another API for calling library functions. In Solomonoff the regular expressions are the primary and only interface. Instead of having separate procedures for union, concatenation and Kleene closure, there is only one procedure that takes arbitrary regular expression and compiles it in batches. This way everything works much faster, doesn't lead to introduction of any ϵ-transitions and allows for more optimisation strategies by AST manipulations. This leads to significant differences in performance.

While, most of other automata libraries (RE2/BRICKS) were not designed for large codebases and have no build system, Solomonoff comes with it's very own build system out-of-the-box. Thrax is the only alternative that does have a "build system" but it's very primitive and relies on generating Makefiles. Solomonoff has a well integrated tool that assembles large projects, detects cyclic dependencies, allows for parallel compilation and performs additional code optimisations. It also ships with syntax highlighting and tools that assist code refactoring. Our project strives to make automata as easy to use and accessible to mass audiences as possible. For this reason we developed a website with online playground where visitors could test Solomonoff and experiment without any setup required. The backend technology we used is Spring Boot, because it allowed for convenient integration with existing API in Java.

While at the beginning our attempts focused on implementing the library is C, switching to Java turned out to be a major advantage. While, it's true that C allows for more manual optimisations, having a garbage collector proved to be a strong asset. The compilation relies on building automata in form of directed graphs. Each vertex itself is a separate Java object. The automaton is always kept trim because all the unreachable vertices are lost and free to be garbage collected at any time. If we tried to implement such data structure in C we would need to reinvent a simple (and most likely, less efficient) garbage collector ourselves. Manipulation of linked lists, linked graphs and other recursive data structures is not as convenient in C as it is in Java. Another advantage that Java gave us over C is that the plenty of existing infrastructure was already implemented in Java. Hence we became more compatible with Samsung's systems where Solomonoff could be deployed. We could also easily integrate our compiler with LearnLib.

For environments where Java is not an option, Solomonoff will allow for generating automata in form of C header files that can be easily included and deployed in production code. It will not allow for manipulation of automata from C level, but our compiler was never intended to be used programmatically. There is, in fact, not much benefit from doing so. If user needs to create their custom automata manually, it's much easier to generate them in AT&T format and then have them read by the compiler. Later they can by manipulated with regular expressions like any other automaton. Generating AT&T has also additional benefits, as such custom script for generating AT&T can be attached to Solomonoff build system as an external routine and then compilation can be optimised more efficiently. An example scenario would be having two independent routines that the build system could decide to run in parallel and then it could cache the results for subsequent rebuilds. For comparison, if anybody decided to use OpenFST manually from from C++, they would need to first learn the API (which is less user friendly than learning AT&T format) and then they would also need to implement caching and parallelization themselves. Moreover, the users are also tied to C++, whereas, AT&T format can be generated in any language. This shows, why API for programmatic manipulation offers little to no benefit, while imposes much more limitations. It should also be mentioned that Solomonoff allows user for implementing their own Java functions that could then be incorporated into the regular expressions as "native" calls.

To guarantee the better utility of Solomonoff, it has to provide ability of interactive evaluation in order to enable efficient and convenient work with the compiler. It can be also vastly useful while learning. The mentioned feature is implemented with a REPL. The biggest challenge here is adjusting the entire stack of compilation processes to work interactively with a sensible manner. The other condition of usefulness is a fast build automation. To make our system meet the need it has to support both parallel compilation and caching of previously compiled units. Also the ability to resolve dependencies may be be important in some more complex projects. It this case the main challenge is to find satisfying solution of ordering in a parallel compilation. This problem can be broken down to a topological sort in a direct acyclic graph. To make the use of Solomonoff even more practical it provides an export option for transducers in a way that allows to include them into a C code during a recompilation phase.

Rozdział 2

Transducers

The theory behind implementation and design of Solomonoff has been studied and developed over the course of 2 years. The early versions and our initial ideas looked very different to the final results we've achieved.

At the beginning it was meant to be a simple tools that focused only on deterministic Mealy automata. Such a model was very limited and it quickly became apparent that certain extensions would need to be made. In the end we implemented a compiler that supports nondeterministic functional weighted symbolic transducers.

2.1. Mealy automata and transducers

The standard definition of automaton found in introductory courses states that finite state automaton is a tuple $(Q, I, \delta, F, \Sigma)$ where Q is the set of states, $I \subset Q$ are the initial states, $F \subset Q$ are the final (or accepting) states, $\delta : Q \times \Sigma \rightarrow Q$ is the transition function and Σ is some alphabet.

A Mealy machine extends the above definition with output $(Q, I, \delta, F, \Sigma, \Gamma)$ where Γ is some output alphabet and transition function has the form of $\delta : Q \times \Sigma \rightarrow Q \times \Gamma$.

Such a model is frequently used in the field of formal methods. Many complex state-based systems can be modelled and simplified using Mealy machines. As an example consider a black-box computer program whose logs can be observed. The current snapshot of the program's memory determines its state. Depending on subsequent user input, we might observe different log traces. There are many existing machine learning and inference algorithms that can build an equivalent model of Mealy machine only by interacting with such a black-box system and reading its logs.

It can be proved that the expressive power of deterministic automata with output is strictly less than that of their nondeterministic counterparts. It is known as the prefix-preserving property. If a deterministic automaton reads input string $\sigma_1\sigma_2\sigma_1$ and prints $\gamma_1\gamma_1\gamma_2$, then at the next step it is only allowed to append one more symbol to the previously generated output. For instance we could not suddenly change the output to $\gamma_2\gamma_2\gamma_2\gamma_1$ after reading one more input symbol $\sigma_1\sigma_2\sigma_1\sigma_2$. The prefix $\gamma_1\gamma_1\gamma_2$ must be preserved.

The problems that we wanted to tackle with Solomonoff revolved around building sequence-to-sequence models. For instance we might want to translate from numbers written as English words into digits. A sentence like "one apple" should become "1 apple". The prefix preserving property would be too limiting because it often happens that the suffix of string has decisive impact on the translation. The phrase "once again" also starts with prefix "one" but it should not be translated as "1ce again"!

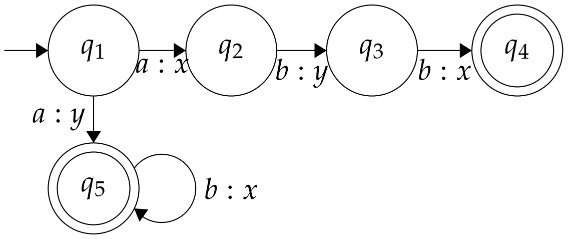
We intended to find the right balance between expressive power of nondeterministic machines and strong formal guarantees of Mealy automata. To achieve this, we initially decided to use multitape automata. The idea was to write all possible continuations of given output and store each in a separate output tape. Then upon reaching the end of string, the state would decide which tape to use as output.

Over the course of research and development we have discovered that the power of multitape Mealy machines was still too limiting for our purposes. In particular we could define congruence classes on pairs of strings similar to those in Myhill-Nerode theorem. Then it can be easily noticed that as long as the number of output tapes is finite, the number of congruence classes must be finite as well. A very simple counter-example would the language that for every string a, aa, aaa,... respectively prints output c, bc, bbc,...and so on. It could not be expressed using only a finite number of output tapes, because there are infinitely many ways to continue the output and none of them is a prefix of the other.

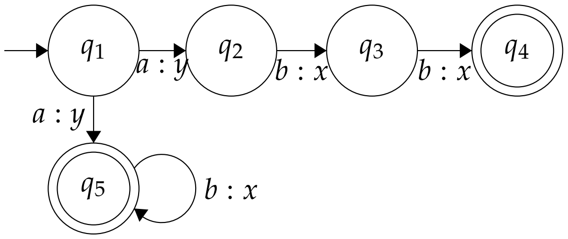
Yet another limitation of Mealy machines is that their δ function enforces outputs of the exact same length as inputs. In the field of natural language processing such an assumption is too strict. For instance, we might want to build a machine that translates sentences from one language to another. A word "fish" in English might have 4 letters but Spanish "pescado" is much longer. Our first idea was to use sequential transducers instead of the plain Mealy automata. Their definition allows the transition function to print output strings of arbitrary length $\delta : Q \times \Sigma \to Q^*$.

The nondeterministic transducers don't suffer from any of the above problems. Their expressive power exactly corresponds to that of regular transductions. It's a very strong and expressive class. The only way to obtain a stronger model would be by introducing context-free grammars and pushdown automata. The reason why we were hesitant to use this approach was because of the possible ambiguity of output. Nondeterministic transducers may contain epsilon transitions, which could lead to infinite number of outputs for any given input. Without epsilons, the number of outputs is finite but it's still possible to return more than one ambiguous output.

The model of automata that was finally implemented in Solomonoff, are the functional nondeterministic transducers. Functionality of transducer means that for any input, there may be at most one output. For instance the transducer below is not functional, because for input string *abb* it prints both outputs *xyx* and *yxx*.



This should not be confused with unambiguous automata. For example the transducer below is functional but ambiguous. The input *abb* prints only a single output *yxx*, albeit there are two ambiguous accepting path that produce it.



Functional nondeterministic transducers proved to provide the perfect balance of power with many strong formal guarantees. While epsilon transitions strictly increase power of transducers, when restricted only to functional automata, the erasure of epsilons becomes possible. Using advance-and-delay algorithm one can test functionality of any automaton in quadratic time. There exists a special version of powerset construction that can take any functional transducers and produce an equivalent unambiguous automaton. One can take advantage of unambiguity to build an inference algorithm for learning functional automata from sample data. The automata are closed under union, concatenation, Kleene closure and composition. Unlike non-functional transducers, they are not closed under inversion but we developed a special algebra that uniquely defines an invertible bijective transduction for any automaton. Glushkov's construction can be augmented to produce functional transducers. Lexicographic semiring of weights can be used to make functional automata more compact.

Once we decided to use the power of functional transducers, the next problem we had to solve was their optimisation. One of the most important operations in natural language processing is the context-dependent rewrite. The standard way of implementing it is by building a large transducer that handles all possible cases. In Solomonoff we've developed our own approach that produces much smaller automata. It is done with lexicographic weights.

Another important optimisation is the state minimisation. Many existing libraries implement separate functions for all regular operations and additional one for minimisation. A regular expression like $A(B+C)*$ would be then translated to a series of function calls like

$$minimise(concatenate(A, kleene\_closure(union(B, C))))$$

This was out initial idea too but later we stumbled upon a better approach. In Solomonoff we don't have separate implementation for those operations. Instead everything is integrated in form of one monolithic procedure that implements Glushkov's construction. For example, in order to compile the expression

$$aa(b+ca)*+c$$

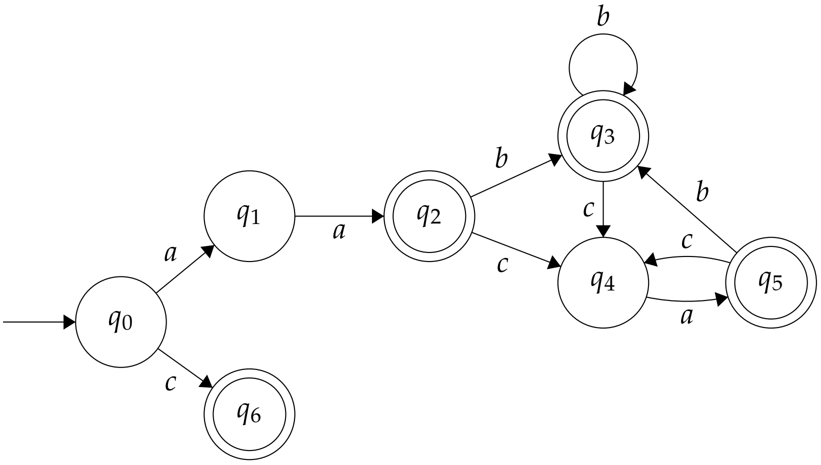
we first convert every symbol into a state and obtain the following intermediate "regular expression"

$$q_1q_2(q_3+q_4q_5)*+q_6$$

Then we add one more state at the beginning that will serve as initial

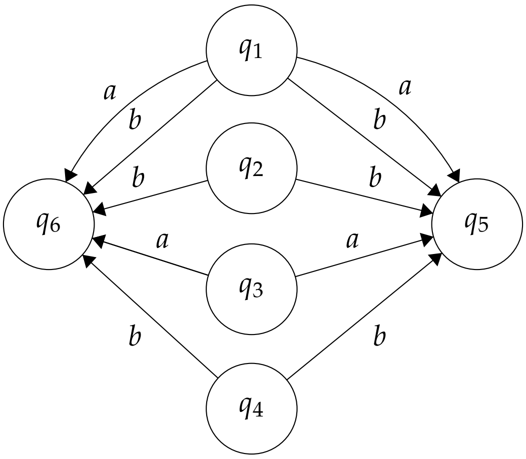
$$q_0(q_1q_2(q_3+q_4q_5)*+q_6)$$

Next we analyse it and determine, which state can be reached from any other. After "reading" q_1 we can "read" q_2. After q_2 we can either read q_3 or q_4. After q_3 we can read either q_3 again or go to q_4. Analogically for the remaining states. We also check, which states can appear at the end of the regular expression. In this example q_2, q_3, q_5 and q_6 are the final states because reading may end after reaching them. By putting all the above information together, we are able to produce the following automaton.

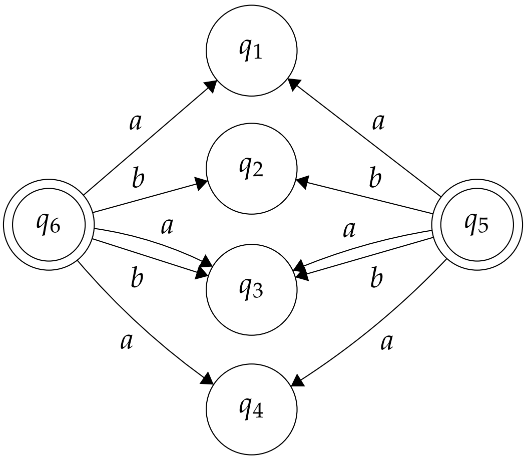


This way the produced automata are very small even without the need for minimisation. If the regular expression consists of n input symbols, then the resulting transducers has $n+1$ states. Because there is one-to-one correspondence between regular expression symbols and automata states, we are able to retain plenty of useful meta information. In particular each state can tell us exactly, which source file it comes from and the precise position in that file. This way, whenever compilation fails, user can see meaningful error messages.

The standard minimisation procedure used by most libraries works by finding the unique smallest deterministic automaton. Nondeterministic automata do not admit unique smallest representative and might be even exponentially smaller than their equivalent minimal deterministic counterparts. Finding the smallest possible nondeterministic automaton is a hard problem. For this reason Solomonoff implements a heuristic pseudo-minimisation algorithm that attempts to compress nondeterministic transducers and does not attempt to determinise them. Glushkov's construction already produces very small automata, hence any attempt at minimising them by determinisation would result in larger automata than the initial ones. Solomonoff's minimisation is inspired by Brzozowski's algorithm and is based on the duality of reachable and unobservable states. In simple terms, if two states have the exact same sets of incoming (or outgoing) transitions then they have no reachable (or observable) distinguishing sequence. For example in the fragment of automaton below, the states q_5 and q_6 are indistinguishable because they have the exact same incoming transitions. As a result, reaching one state always implies also reaching the other.



Analogically in the example below the states also are indistinguishable but this time the outgoing transitions are the same. Hence the effects of reaching one state are equivalent to the other.



The states q_5 and q_6 can be merged without affecting the language of automaton. Such a pseudo-minimisation procedure has been chosen, because it works especially well when combined with Glushkov's construction. The process of merging indistinguishable states is analogical to the process of isolating common parts of regular expression. For example the following

$$ab + aab + ac(b + bb)$$

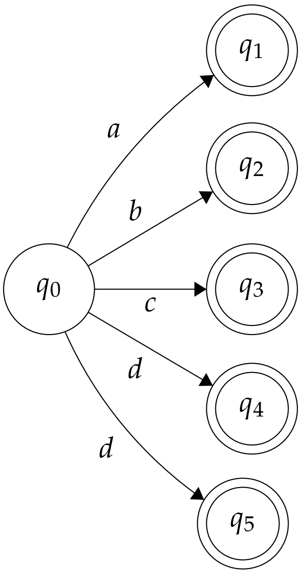
could be shortened to

$$a(\epsilon + a + c(\epsilon + b))b$$

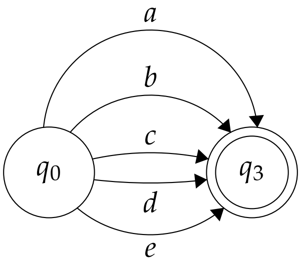
by isolating the common prefix a and suffix b. Extracting common prefixes exactly correspond to merging states with identical incoming transition, while common suffixes correspond to states with the same outgoing transitions. Sometimes there are cases that do not have any common prefix/suffix in the regular expression itself but still produce indistinguishable states in the automaton. For instance consider

$$a + b + c + d + e$$

which yields automaton



that could be minimised down to only two states



Interestingly, the regular expression could not be minimised any further. Hence we observe that our procedure does more than a simple syntactic manipulation could achieve.

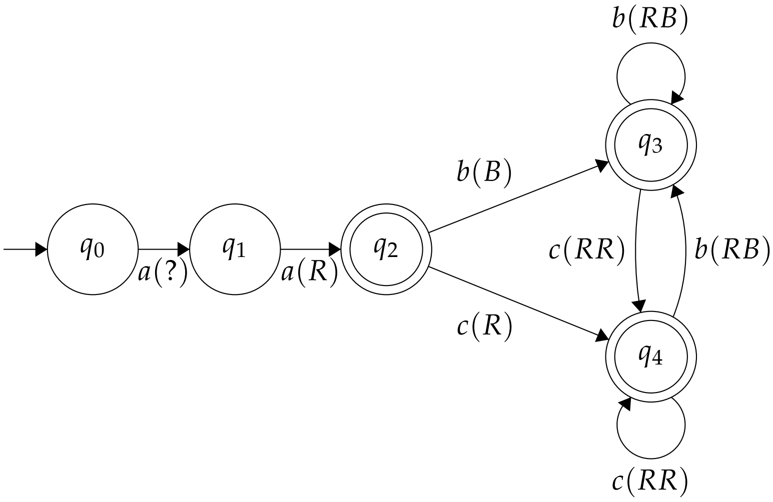
Glushkov's construction has one more advantage. Because every symbol becomes a state, it's very easy for the user to predict the exact placement of transitions between them. This way, it's easy to embed any property of the transition directly inside the regular expression. For example, suppose that we want to assign come colours to all transitions. Let B stand for blue and R for red. Then it's possible to embed these colours in a expression like follows

$$aRa(Bb + Rc)R*$$

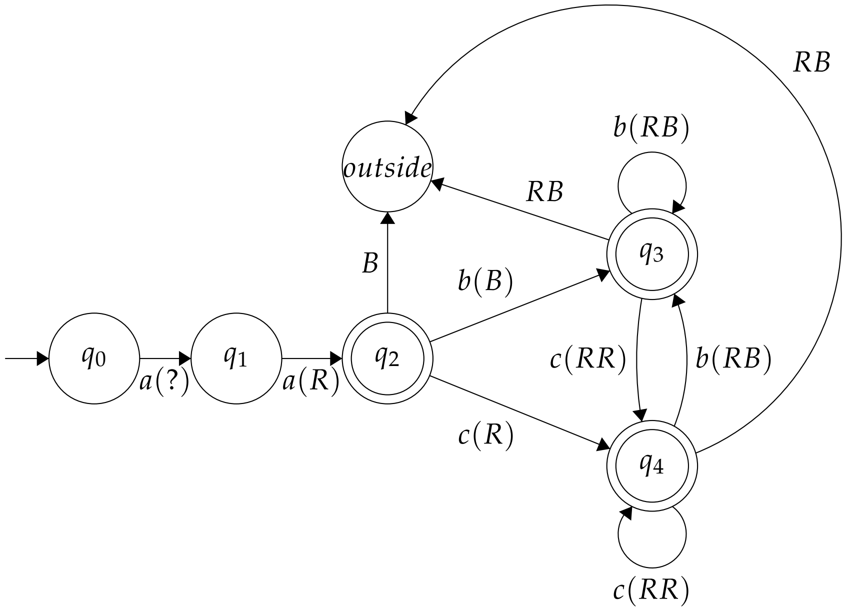
which becomes

$$q_1Rq_2(Bq_3 + Rq_4)R * B$$

We know that after q_1 we can read q_2 and along the way we have to cross the color R, because it stand in between q_1 and q_2. Hence it determines that the transition from q_1 to q_2 should be red. Similarly transition from q_2 and q_3 must be blue and from q_2 and q_4 is red. The transition from q_3 to q_3 will have colour RB, because R is under the Kleen closure and B is right before q_3. At this point we notice that there must be defined some way of mixing colours. In other words, any meta-information that we wish to embed in our regular expressions must at least form a monoid. The graph resulting from our example looks as follows



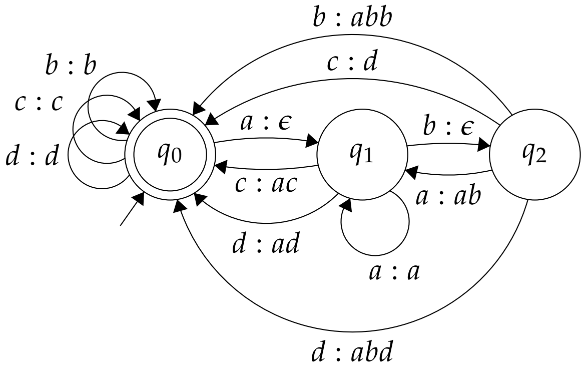
Because colours form a monoid, we can use neutral element as a default value for all unspecified edges like $a(?)$. Moreover, it is also possible to attach meta-information to final states. We could imagine that an accepting state is nothing more than a state with a special outgoing transition that goes "outside" of automaton and has no target state. In our example the colours of such "final" transitions are as follows



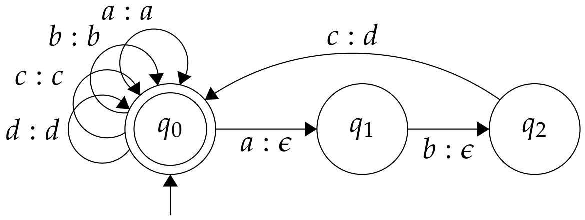
In Solomonoff, the meta-information of interest are transition outputs. We could imagine that strings composed of *R* and *B* symbols are the output of transducer. At this point the reader should appreciate how beautifully and naturally subsequential transducers (automata with output produced at accepting states) are derived from Glushkov's construction. As soon as we enrich regular expressions with meta-information, the state outputs emerge as if they were always there, merely hiding from the view.

Glushkov's construction together with minimisation procedure and embeded meta-information allowed for efficient implementation of union, concatenation and Kleene closure together with output strings. In order to make the compiler applicable to real-life linguistic problems it also must support context dependent rewrites. This is a heavyweight operations that often constitutes a major performance bottleneck. Our goal was to make it as efficient as possible. In order to solve this issue we developed a special lexicographic semiring. This is an innovative solution never seen before.

Suppose we want to replace every occurrence of *abc* with *d*. Even for such a simple scenario, the corresponding transducer will look rather complex

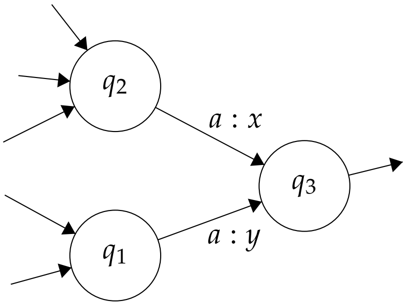


A much simpler alternative would be the nondeterministic transducer

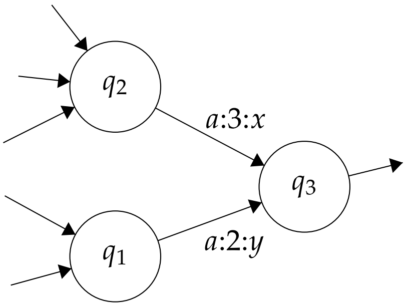


The problem with such solution is that for input strings like *aabcb* both outputs *aabcb* and *adb* are generated. If only there was a way to assign priority to some outputs in order to disambiguate them, then context dependent rewrites could be expressed by much simpler automata. This is precisely what lexicographic weights are for.

Consider the following fragment of automaton and suppose that it's possible to simultaneously nondeterministically reach both q_2 and q_1.

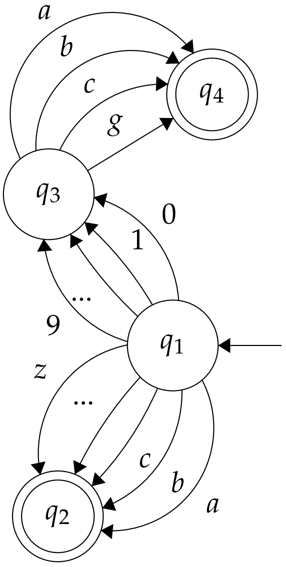


If the next input symbol is a, then the transducer will reach state q_3 and generate two outputs - one ending in x and the other in y. If the automaton later accepts it will produce at least two ambiguous outputs. Lexicographic weights allow us to assign priority to different transitions. In the following example, only the output ending in x will reach state q_3 and the other ending in y will be discarded because it came to q_3 over transition with lower weight.

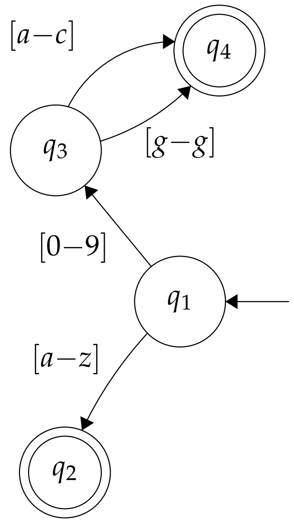


Lexicographic weights allow to make the transducers even more compact. We proved that there exist weighted automata exponentially smaller than even the smallest unweighted nondeterministic ones. In the presence of lexicographic weights, context-dependent rewrites become expressible directly in Glushkov's construction, without the need for calling any "external operations".

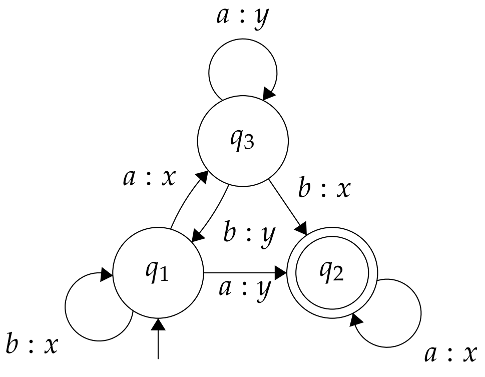
In the tasks of natural language processing it's common to work with large alphabets. User input might contain unexpected sequences like math symbols, foreign words or even emojis and other UNICODE entities. The regular expressions should handle such cases gracefully, especially when using wildcards such as `.*` or `\p{Lu}`. Representation of large character classes is a challenging task for finite state automata. In order to use the dot wildcard in UNICODE, the automaton would require millions of transitions, one for each individual symbol. In order to optimise this, Solomonoff employs symbolic transitions. While classic automata have edges labelled with individual symbols, our transducers have edges that span entire ranges. A range is easy to encode in computers memory. It's enough to store the first and last symbol. Below is an example of classical finite state automaton



and an equivalent symbolic automaton that uses closed ranges of symbols as transition predicates



Compilation of transducers is the core part of our library but in order to make the automata useful there must be a way to execute them. Deterministic transducers and Mealy machines could be evaluated in linear time. Nondeterministic automata are more expensive. The computation might branch and automaton could be in multiple states simultaneously. Using dynamic programming it's possible to build a table with rows representing consecutive symbols of input string and each column keeping track of one state. At each step of execution, one input symbol is read and one row in the table is filled based on the contents of previous row. Below is an example of nondeterministic finite state automaton and its corresponding evaluation table after reading input string *aba*.



By the end of evaluating such a table, it's enough to scan the last row to find a column of some accepting state. In the above example, such a state is q_2 and we can observe that it's active in the last row.

In case of transducers, we not only want to know, whether the string was accepted or not but also the output generated along the way. In order to do

symbol	q_1	q_2	q_3
initial configuration	1	0	0
a	0	1	1
b	1	1	0
a	0	1	1

this the table can be backtracked from the accepting state backwards. For the backtracking step to be possible, we need to store information about source state of each taken transition.

symbol	q_1	q_2	q_3
initial configuration	q_1	$\emptyset$	$\emptyset$
a	$\emptyset$	q_1	q_1
b	q_3	q_3	$\emptyset$
a	$\emptyset$	q_1	q_1

Assuming that every transition is uniquely determined by its symbol, source state and target state (in other words $\delta : Q \times \Sigma \times Q \to \Gamma^*$), it becomes possible to use such evaluation table to find the exact accepting path in automaton and collect all the outputs printed along the way.

This algorithm has been made even more efficient by using techniques from graph theory. Every automaton is a directed graph that could be represented as adjacency matrix. An example is shown below.

adjacency	q_1	q_2	q_3
q_1	b:x	a:y	a:x
q_2	$\emptyset$	a:x	$\emptyset$
q_3	b:y	b:x	a:y

Sparse graphs can be optimised and instead of using matrix it's possible to only store the list of adjacent vertices.

$$(q_1, b : x, q_1), (q_1, a : y, q_2), (q_1, a : x, q_3), (q_2, a : x, q_2), (q_3, b : y, q_3), ...$$

Nondeterministic automata very often are the perfect example of sparse graphs. Hence instead of using evaluation table, it's better to use a list of states nondeterministically reached at any given step of evaluation.

The backtracking can be made efficient by storing pointer to source state of any taken transition.

symbol	list
initial configuration	q_1
a	q_2, q_3
b	q_1, q_2
a	q_2, q_3

symbol	list
initial configuration	$q_1($ from $q_1)$
a	$q_2($ from $q_1), q_3($ from $q_1)$
b	$q_1($ from $q_3), q_2($ from $q_3)$
a	$q_2($ from $q_1), q_3($ from $q_1)$

Glushkov's construction relies on building three sets: the set of initial symbols, final symbols and 2-factor strings (that is, all substrings of length 2). The early prototype versions of Solomonoff would represent sets as bitsets, where each bit specifies whether element belongs to the set or not. This representation simplified implementation of many algorithms but was highly inefficient. Later implementation used hash sets instead. While hash maps have constant insertions and deletions, they ensure it at the cost of larger memory consumption. During benchmarks on real-life datasets, Solomonoff would often run out of memory and crash. The final version uses signly linkged graphs backed by arrays. This provides the highest efficiency with smallest memory footprint but it non-trivial to implement. The optimisation relies on representing sparse sets as arrays of elements. This approach proved to be the best choice, because Glushkov's construction inherently ensures uniqueness of all inserted elements (hence using hashes to search for duplicates before insertion was not necessary) and it does not use deletions. As a result any array was guaranteed to behave like a set.

The standard definition of Glushkov's construction produces set of 2-factors as its output. Then a separate procedure would be necessary to collect all such strings and convert them into a graph of automaton. We found a way to make it more efficient and build the graph directly. The step of building 2-factors was entirely bypassed. This provided even further optimisation.

One of the core features of Solomonoff is the algorithm for detecting ambiguous nondeterminism. Advance-and-delay procedure can check functionality of automaton in quadratic time. While the compiler does provide implementation of this procedure, it is not used for checking functionality. Instead we use a simpler algorithm for checking strong functionality of lexicographic weights. While the performance difference between the two is negligable, the advantage of our approach comes from better error messages in case of nondeterminism. If some lexicographic weights are in conflict with each other, compiler can point user to the exact line and column of text where the conflict arises, whereas advance-and-delay might miss the origin of problem and only fail further down the line.

2.2. Gluszkov's construction

In this section we provide formal and mathematically rigorous presentation of Glushkov's construction.

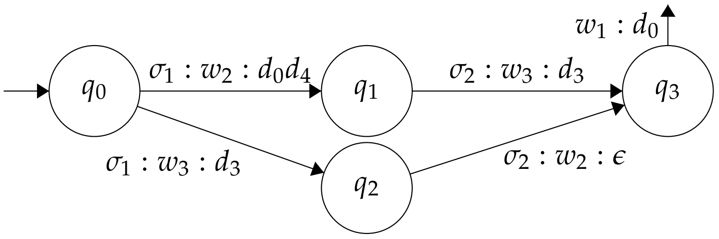
Let Σ be the (not necessarily finite) alphabet of automaton. Let χ be the set of subsets of Σ that we will call ranges of Σ. Let $\overline{\chi}$ be the closure of χ under countable union and complementation (so it forms a sigma algebra). For instance, imagine that there is total order on Σ and χ is the set of all intervals in Σ. Now we want to build an automaton whose transitions are not labelled with symbols from Σ, but rather with ranges from χ. Union $\chi_0 \cup \chi_1$ of two elements from χ "semantically" corresponds to putting two edges, $(q, \chi_0, q') \in \delta$ (for a moment forget about outputs and weights) and $(q, \chi_1, q') \in \delta$. There is no limitation on the size of δ. It might be countably infinite, hence it's natural that $\overline{\chi}$ should be closed under countable union. Therefore, χ is the set of allowed transition labels and $\overline{\chi}$ is the set of all possible "semantic" transitions. We could say that $\overline{\chi}$ is discrete if it contains every subset of Σ. An example of discrete $\overline{\chi}$ would be finite set Σ with all UNIX-style ranges $[\sigma\text{-}\sigma']$ included in χ.

Transducers with input Σ^* and output Γ^* can be seen as FSA working with single input $\Sigma^* \times \Gamma^*$. Therefore we can treat every pair of symbols (σ, γ) as an atomic formula of regular expressions for transducers. We can use concatenation $(\sigma, \gamma_0)(\epsilon, \gamma_1)$ to represent $(\sigma, \gamma_0\gamma_1)$. It's possible to create ambiguous transducers with unions like $(\epsilon, \gamma_0) + (\epsilon, \gamma_1)$. To make notation easier, we will treat every σ as (σ, ϵ) and every γ as (ϵ, γ). Then instead of writing lengthy $(\sigma, \epsilon)(\epsilon, \gamma)$ we could introduce shortened notation $\sigma : \gamma$. Because we would like to avoid ambiguous transducers we can put restriction that the right side of : should always be a string of Γ^* and writing entire formulas (like $\sigma : \gamma_1 + \gamma_2^*$) is not allowed. This restriction will later simplify Glushkov's algorithm.

We define $\mathcal{A}^\Sigma$ to be the set of atomic characters. For instance we could choose $\mathcal{A}^\Sigma = \Sigma \cup \{\epsilon\}$ for FSA/transducers and $\mathcal{A}^\Sigma = \chi$ for ranged automata.

We call $RE^{\Sigma:D}$ the set of all regular expression formulas with underlying set of atomic characters $\mathcal{A}^{\Sigma}$ and allowed output strings D. It's possible that D might be a singleton monoid $\{\epsilon\}$ but it should not be empty, because then no element would belong to $\Sigma^* \times D$. By inductive definition, if ϕ and ψ are $RE^{\Sigma:D}$ formulas and $d \in D$, then union $\phi + \psi$, concatenation $\phi \cdot \psi$, Kleene closure ϕ^* and output concatenation $\phi : d$ are $RE^{\Sigma:D}$ formulas as well. Define $V^{\Sigma:D} : RE^{\Sigma:D} \to \Sigma^* \times D$ to be the valuation function:
$V^{\Sigma:D}(\phi + \psi) = V^{\Sigma:D}(\phi) \cup V^{\Sigma:D}(\psi)$
$V^{\Sigma:D}(\phi \cdot \psi) = V^{\Sigma:D}(\phi) \cdot V^{\Sigma:D}(\psi)$
$V^{\Sigma:D}(\phi^*) = (\epsilon, \epsilon) + V^{\Sigma:D}(\phi) + V^{\Sigma:D}(\phi)^2 + ...$
$V^{\Sigma:D}(\phi : d) = V^{\Sigma:D}(\phi) \cdot (\epsilon, d)$
$V^{\Sigma:D}(a) = a$ where $a \in \mathcal{A}^{\Sigma:D}$
Some notable properties are:
$x : y_0 + x : y_1 = x : (y_0 + y_1)$
$x : \epsilon + x : y + x : y^2 ... = x : y^*$
$(x : y_0)(\epsilon : y_1) = x : (y_0 y_1)$
$x_0 : (y_0 y') + x_1 : (y_1 y') = (x_0 : y_0 + x_1 : y1) \cdot (\epsilon : y')$
$x_0 : (y' y_0) + x_1 : (y' y_1) = (\epsilon : y') \cdot (x_0 : y_0 + x_1 : y1)$
Therefore we can see that expressive power with and without : is the same.

It's also possible to extend regular expressions with weights. Let $RE_W^{\Sigma:D}$ be a superset of $RE^{\Sigma:D}$ and W be the set of weight symbols. If $\phi \in RE_W^{\Sigma\to D}$ and $w_0, w_1 \in W$ then $w_0\phi$ and ϕw_1 are in $RE_W^{\Sigma\to D}$. This allows for inserting weight at any place. For instance, the automaton below



could be expressed using

$$((\sigma_1 : d_0 d_4) w_2 (\sigma_2 : d_3) w_3 + (\sigma_1 : d_3) w_3 \sigma_2 w_2) : d_0$$

The definition of $V^{\Sigma:D}(\phi w)$ depends largely on W but associativity $(\phi w_1) w_2 = \phi(w_1 + w_2)$ should be preserved, given that W is an additive monoid. This also implies that $w_1 \epsilon w_2 = w_1 w_2$, which is semantically equivalent to the addition $w_1 + w_2$.

First step of Glushkov's algorithm is to create a new alphabet Ω in which every atomic character (including duplicates but excluding ϵ) in ϕ is treated as a new individual character. As a result we should obtain new rewritten formula

$\psi \in RE_W^{\Omega\to D}$ along with mapping $\alpha : \Omega \to \mathcal{A}^\Sigma$. This mapping will remember the original atomic character, before it was rewritten to unique symbol in Ω. For example

$$\phi = (\epsilon : x_0)x_0(x_0 : x_1x_3)x_3w_0 + (x_1x_2)^*w_1$$

will be rewritten as

$$\psi = (\epsilon : x_0)\omega_1(\omega_2 : x_1x_3)\omega_3w_0 + (\omega_4\omega_5)^*w_1$$

with $\alpha = \{(\omega_1, x_0), (\omega_2, x_0), (\omega_3, x_3), (\omega_4, x_1), (\omega_5, x_2)\}$.

Next step is to define function $\Lambda : RE_W^{\Omega\to D} \rightharpoonup (D \times W)$. It returns the output produced for empty word ϵ (if any) and weight associated with it. (We use symbol $\rightharpoonup$ to highlight the fact that Λ is a partial function) For instance in the previous example empty word can be matched and the returned output and weight is (ϵ, w_1). Because both D and W are monoids, we can treat $D \times W$ like a monoid defined as $(y_0, w_0) \cdot (y_1, w_1) = (y_0y_1, w_0 + w_1)$. We also admit $\emptyset$ as multiplicative zero, which means that $(y_0, w_0) \cdot \emptyset = \emptyset$. We denote W's neutral element as 0. This facilitates recursive definition:

$\Lambda(\psi_0 + \psi_1) = \Lambda(\psi_0) \cup \Lambda(\psi_1)$ if at least one of the sides is $\emptyset$, otherwise error
$\Lambda(\psi_0\psi_1) = \Lambda(\psi_0) \cdot \Lambda(\psi_1)$
$\Lambda(\psi_0 : y) = \Lambda(\psi_0) \cdot (y, 0)$
$\Lambda(\psi_0w) = \Lambda(\psi_0) \cdot (\epsilon, w)$
$\Lambda(w\psi_0) = \Lambda(\psi_0) \cdot (\epsilon, w)$
$\Lambda(\psi_0^*) = (\epsilon, 0)$ if $(\epsilon, w) = \Lambda(\psi_0)$ or $\emptyset = \Lambda(\psi_0)$, otherwise error
$\Lambda(\epsilon) = (\epsilon, 0)$
$\Lambda(\omega) = \epsilon$ where $\omega \in \Omega$

Next step is to define $B : RE_W^{\Omega\to D} \to (\Omega \rightharpoonup D \times W)$ which for a given formula ψ returns set of Ω characters that can be found as the first in any string of $V^{\Omega\to D}(\psi)$ and to each such character we associate output produced "before" reaching it. For instance, in the previous example of ψ there are two characters that can be found at the beginning: ω_1 and ω_4. Additionally, there is ϵ which prints output x_0 before reaching ω_1. Therefore $(\omega_1, (x_0, 0))$ and $(\omega_3, (\epsilon, 0))$ are the result of $B(\psi)$. For better readability, we admit operation of multiplication $\cdot : (\Omega \rightharpoonup D \times W) \times (D \times W) \to (\Omega \rightharpoonup D \times W)$ that performs monoid multiplication on all $D \times W$ elements returned by $\Omega \rightharpoonup D \times W$.

$B(\psi_0 + \psi_1) = B(\psi_0) \cup B(\psi_1)$
$B(\psi_0\psi_1) = B(\psi_0) \cup \Lambda(\psi_0) \cdot B(\psi_1)$
$B(\psi_0w) = B(\psi_0)$
$B(w\psi_0) = (\epsilon, w) \cdot B(\psi_0)$
$B(\psi_0^*) = B(\psi_0)$
$B(\psi_0 : d) = B(\psi_0)$

$B(\epsilon) = \emptyset$
$B(\omega) = \{(\omega, (\epsilon, 0))\}$
It's worth noting that $B(\psi_0) \cup B(\psi_1)$ always yields function (instead of relation) because every Ω character appears in ψ only once and it cannot be both in ψ_0 and ψ_1.

Next step is to define $E : RE_W^{\Omega \to D} \to (\Omega \rightharpoonup D \times W)$, which is very similar to B, except that E collects characters found at the end of strings. In our example it would be $(\omega_3, (\epsilon, w_0))$ and $(\omega_5, (\epsilon, w_1))$. Recursive definition is as follows:
$E(\psi_0 + \psi_1) = E(\psi_0) \cup E(\psi_1)$
$E(\psi_0\psi_1) = E(\psi_0) \cdot \Lambda(\psi_1) \cup B(\psi_1)$
$E(\psi_0 w) = E(\psi_0) \cdot (\epsilon, w)$
$E(w\psi_0) = E(\psi_0)$
$E(\psi_0^*) = E(\psi_0)$
$E(\psi_0 : d) = E(\psi_0) \cdot (d, 0)$
$E(\epsilon) = \emptyset$
$E(\omega) = \{(\omega, (\epsilon, 0))\}$

Next step is to use B and E to determine all two-character substrings that can be encountered in $V^{\Omega \to D}(\psi)$. Given two functions $b, e : \Omega \rightharpoonup D \times W$ we define product $b \times e : \Omega \times \Omega \rightharpoonup D \times W$ such that for any $(\omega_0, (y_0, w_0)) \in b$ and $(\omega_1, (y_1, w_1)) \in c$ there is $((\omega_0, \omega_1), (y_0 y_1, w_0 + w_1)) \in b \times e$. Then define $L : RE_W^{\Omega \to D} \to (\Omega \times \Omega \rightharpoonup D \times W)$ as:
$L(\psi_0 + \psi_1) = L(\psi_0) \cup L(\psi_1)$
$L(\psi_0\psi_1) = L(\psi_0) \cup L(\psi_1) \cup E(\psi_0) \times B(\psi_1)$
$L(\psi_0 w) = L(\psi_0)$
$L(w\psi_0) = L(\psi_0)$
$L(\psi_0^*) = L(\psi_0) \cup E(\psi_0) \times B(\psi_0)$
$L(\psi_0 : d) = L(\psi_0)$
$L(\epsilon) = \emptyset$
$L(\omega) = \emptyset$
One should notice that all the partial functions produced by B, E and L have finite domains, therefore they are effective objects from computational point of view.

The last step is to use results of L, B, E, Λ and α to produce automaton $(Q, q_\epsilon, W, \Sigma, D, \delta, \tau)$ with
$\delta : Q \times \Sigma \to (Q \rightharpoonup D \times W)$
$\tau : Q \rightharpoonup D \times W$
$Q = \{q_\omega : \omega \in \Omega\} \cup \{q_\epsilon\}$
$\tau = E(\psi)$

$(q_{\omega_0}, \alpha(\omega_1), q_{\omega_1}, d, w) \in \delta$ for every $(\omega_0, \omega_1, d, w) \in L(\psi)$
$(q_{\epsilon}, \alpha(\omega), q_{\omega}, d, w) \in \delta$ for every $(\omega, d, w) \in B(\psi)$

2.3. Code specification

The exact implementation of most functions in Solomonoff requires deep technical understanding of automata theory. The examples and intuitive explanations provided at the beginning of this chapter convey the essence of most implementations. This section is meant to guide the reader through general code structure, conventions and their justification.

The process of writing compilers is very different from a standard engineering process. The code must follow rigorous specification and there is little room for bugs. The development of Solomonoff, since the earliest stages could be shortly summarised as "mathematically rigorous programming", rather than "ad-hoc software engineering". Therefore our Java implementation follows strict conventions of contract-oriented programming. All preconditions, postconditions and invariants are asserted using runtime analysis. We also make extensive use of the "typeclass pattern". Almost everything is an interface. Each of the interfaces corresponds to some formal mathematical definition and stores a handful of axioms (each axiom is a method). Non-axiom methods are canonically represented as pure static functions. For example consider set of finite sequences Ω_X of elements from set X. A string $\sigma_1\sigma_2\sigma_1$ over alphabet $\Sigma = \{\sigma_1, \sigma_2\}$ might be represented as a sequence $\langle\sigma_1, \sigma_2, \sigma_1\rangle$ of Ω_Σ. An axiomatization of finite sequences might be a (dependent) pair (n, f) of some number n and function f that to every integer between 0 and n assigns some element of X. Formally written as $(n, f) : \mathbb{N} \times (0...n \to X)$. Then in Java we could represent it as class

```
class Ω<X>{
    int n();
    X f(int i); //requires 0<=i<n
}
```

Then a function that operates on Ω_X, like concatenation for instance, would have to be represented using a pure static method.

```
static <X> Ω<X> concat(Ω<X> left, Ω<X> right){...}
```

Sometimes, different implementations of Ω_X might use different underlying data structures, which in turn allow for more efficient implementations of certain functions. For example the generic implementation of `concat` might create a new instance with a new backing array and copy the elements of the two previous sequences. However, if we know that the exact implementation uses linked lists, a much more efficient procedure could work in constant time. For this reason Solomonoff sometimes defines the non-axiomatic methods directly as part of interface, not because they are necessary, but because can be implemented more efficiently. The programmer should still be aware of the underlying formal specification.

Our Java code also strictly follows the specification of linear types. Even though Java compiler does not have any support for borrowing, ownership and lifetimes, that does not mean the code cannot be properly annotated in such a manner. For example consider the above definition of `concat`. It might produce a new instance for each call, but a more efficient implementation would mutate one of the arguments. The only problem is that mutations might leads to bugs and data races when not used properly. To avoid that, we explicitly specify (in the comments), which function arguments are linear. For instance, if we specified that `left` argument is "consumed", then we should never use it after calling `concat`. Suppose that Ω_X is implemented using linked lists and the right argument is appended to the left one. We could write something like this to concatenate `a + b1 + b2`

```
concat(a,b1);
concat(a,b2);
```

but this solution might break as soon as the implementation is changed. The canonical way of writing code with linear types would be instead

```
a2 = concat(a,b1);
//a is consumed and should never be touched again!
a3 = concat(a2,b2);
//a2 is consumed and should never be touched again!
```

Programmer must also keep in mind that in order to invoke `concat(left,right)` we must also make sure that we own the `left` argument. If a reference to this variable was simultaneously used in some other place and we consumed it here, that other place would have no way of knowing that `left` is no longer valid. Therefore attention must be paid, whenever we copy references from one place to the other (which is commonly known as "borrowing").

Every function that mutates its arguments, can also be represented as linear function without mutations. For instance

```
static <X> void concat(Ω<X> left, Ω<X> right)
```

is the mutating version of

```
/**consumes left*/
static <X> Ω<X> concat(Ω<X> left, Ω<X> right)
```

In object-oriented languages we have one more possible approach

```
void concat(Ω<X> right)
```

where `this` instance is implied to be linear (`this=left`).

In Solomonoff, the convention is to use the second approach for static functions, but third approach whenever the method is part of an interface, although there are some exceptions. In particular, note that the static functions require all generic arguments to be mentioned explicitly. When there are too many of them, it becomes more convenient and readable to turn static methods into interface methods and let all the generics be declared only once globally as the interface generics. Instead of

```
static <X1,X2,X3,Y> Y f1(X1 x1, X2 x2, X3 x3){...}
static <X1,X2,X3,Y> Y f2(X1 x1, X2 x2, X3 x3){...}
```

we obtain more concise notation

```
interface ShareGenerics<X1,X2,X3,Y>{
    default Y f1(X1 x1, X2 x2, X3 x3){...}
    default Y f2(X1 x1, X2 x2, X3 x3){...}
}
```

This approach should be considered merely as syntactic sugar. Those functions are usually not meant to be extended in any way. Most of the methods in Solomonoff are pure and do not have any state. Even the seemingly impure functions become pure when linear logic is taken into account.

Rozdział 3

Build system

The primary goal of this project was do develop the compiler backend. Initially little attention was paid to structuring larger projects composed of multiple source files and dependencies. While smaller applications can work fine with a single script of regular expressions, in the tasks on natural language processing its common to build large codebases composed of millions of linguistic rules. Being able to manage and organise them efficiently becomes a major issue.

3.1. Build system as extension of compiler

The compiler builds automata using our special version fo Glushkov's construction, which is capable of compiling subexpressions independently of each other. The results are returned in form of singly linked graph. This format allows one to parallelise compilation and split work across multiple files. For instance user could create two files `file1.mealy` and `file2.mealy`, define some variable in the first one and then use it in the second.

```
file1.mealy:
    var1 = ’abc’
file2.mealy:
    var2 = (’prefix’ var1 ’suffix’)*
```

Such a feature is not native to the compiler backend itself (it only parses continuous string of input) and has been delegated to the build system as an independent application instead.

The easiest approach to implementing build system would be by concatenating all source files into one large steam of input and then feed it into the compiler. For instance

```
concatenatedFiles.mealy:
    //from file1.mealy
    var1 = 'abc'
    //from file2.mealy
    var2 = ('prefix' var1 'suffix')*
```

It could be easily done with a simple Bash script or a couple of Makefiles. Despite being very straightforward, such a solution has multiple flaws. The order of concatenation matters a lot. The compiler requires that every variable is defined before being used. Should the order get mistakenly swapped, the compilation would fail. For instance

```
concatenatedFiles.mealy:
    //from file2.mealy
    var2 = ('prefix' var1 'suffix')*
    // var1 used before definition!
    //from file1.mealy
    var1 = 'abc'
```

As a result, it would be user's obligation to ensure correct order of concatenation, which might quickly become unmaintainable in large projects. Second problem stems from linear types. The compiler follows the semantics of linear logic and a variable once consumed should not be reused. An explicit copy is always necessary. Hence the order of definition and usage of all variables is even more important than in other general-purpose languages that do not have linear types. For instance this will compile

```
X = 'a'
Y = !!X 'b'
Z = X 'c'
```

but this one will fail

```
X = 'a'
Z = X 'c'
Y = !!X 'b'
```

Third problem is that managing dependencies and building packages would become impossible. For example we might imagine code, which includes some library

```
include libX
X = f  // f is defined in libX
```

If in the future a new version of the library is released, it might happen that variable X is added and suddenly our code becomes invalid because we're trying to redefine X.

Our initial thought was to introduce a special function `import!(’filepath’)`, which would load any variable from a precompiled binary file. Then the user would write code as follows

```
X = import!(’lib/libX/f’)
```

This approach suffers from one logistic problem. If user decided to split their code into several files and then import the necessary automata at will, the build system would need to detect the correct compilation order of all files. For instance if there were two files like these

```
file X.mealy:
    a = ’a’
file Y.mealy:
    b = import!(’X/a’)
```

then the build system would need to first compile `X.mealy` and produce binary file `X` and only then compilation of `Y.mealy` would become possible. The major problem appears when cyclic dependencies between files arise

```
file X.mealy:
    X = import!(’Y/Y’)
file Y.mealy:
    Y = import!(’X/X’)
```

While it’s possible to create a built tool capable of detecting such problems and notifying the user, its main disadvantage is that in some cases, there might be cyclic dependency between files, despite not introducing cyclic dependencies between automata themselves. As a result, logically valid regular expressions would be prematurely rejected by build system. For example

```
file X.mealy:
    X1 = import!(’Y/Y1’)
    X2 = ’a’
file Y.mealy:
    Y1 = ’b’
    Y2 = import!(’X/X2’)
```

The final solution we settled for was to split compilation into three phases. First the build system scans all files (in parallel) and builds abstract syntax trees of all variable definitions. In those trees there would be references to other variables that could be defined anywhere else in the project. Then in the second phase the variables would be scanned and a dependency graph would be built. We used a specialised data structure for efficiently working with directed acyclic graphs.

If at any point a cycle was introduced, the build system would throw an error. Finally in the last phase every node in the graph would be compiled in parallel. For optimal performance, multiple threads would take vertices in topological order. If there is a directed edge from vertex v_1 to v_2 (meaning that variable v_2 depends on v_1), then v_1 will be added to the queue before v_2. Threads process queue entries in the FIFO order.

3.2. Implementation

In the first phase, the parser that scans source files has been reused from the compiler backend. More precisely, the grammar of syntax is taken from the library but the build system has its own implementation of `SolomonoffGrammarListener`. Parsing is then performed using the standard ANTLR functions

```
SolomonoffGrammarLexer lexer =
  new SolomonoffGrammarLexer(sourceFile);
SolomonoffGrammarParser parser =
  new SolomonoffGrammarParser(new CommonTokenStream(lexer));

parser.addErrorListener(new BaseErrorListener() {
    public void syntaxError(Recognizer<?, ?> recognizer,
            Object offendingSymbol,
            int line,
            int charPositionInLine,
            String msg,
            RecognitionException e) {
         System.err.println("line " + line
         + ":" + charPositionInLine
         + " " + msg + " " + e);
    }
});

final SolomonoffWeightedParser listener =
  new SolomonoffWeightedParser(collector, sourceFile, compiler);
ParseTreeWalker.DEFAULT.walk(listener, parser.start());
```

The listener is responsible for building abstract syntax tree for each variable and storing all definitions. Here is a fragment of ANTLR grammar that forms the list of variable definitions

```
funcs :
    funcs exponential='!!'? ID '=' mealy_union  # FuncDef
    |
;
```

Whenever some definition is parsed, the following listener callback is invoked

```
@Override
public void exitFuncDef(FuncDefContext funcDefContext) {
    String currentVariableID = funcDefContext.ID().getText();
    SolomonoffWeighted def = stack.pop();
    collector.define(currentVariable, def);
}
```

The integral part of listener is the `collector` object. It is responsible for storing all defined variables and remembering their respective abstract syntax trees. Initially, it was enough to store a `HashMap` within the `SolomonoffWeightedParser` object and register variables using

```
HashMap<String,SolomonoffWeighted> idToAST =
    new HashMap<>();

@Override
public void exitFuncDef(FuncDefContext funcDefContext) {
    String currentVariableID = funcDefContext.ID().getText();
    SolomonoffWeighted def = stack.pop();
    idToAST.put(currentVariable, def);
}
```

Later it was changed to a more complicated implementation, which is

```
ConcurrentHashMap<String,SolomonoffWeighted> idToAST;
SolomonoffWeightedParser(
    ConcurrentHashMap<String,SolomonoffWeighted> shared){
    this.idToAST = shared;
}
@Override
public void exitFuncDef(FuncDefContext funcDefContext) {
    String currentVariableID = funcDefContext.ID().getText();
    SolomonoffWeighted def = stack.pop();
    idToAST.put(currentVariable, def);
}
```

In such a way, it became possible to create multiple instances of `SolomonoffWeightedParser` and run all of them in parallel, while the results of parsing could still be collected into one joint map. The use of `ConcurrentHashMap` instead of regular HashMap allows for concurrent insertions, without requiring any locks or synchronization.

The abstract syntax tree closely follows the definition of Solomonoff's syntax. Its formal grammar is fairly simple and mimics the mathematical definition of regular expressions

```
mealy_union
:
    (mealy_concat bar='|') * mealy_concat # MealyUnion
;

mealy_concat
:
    mealy_concat mealy_Kleene_closure
    | mealy_Kleene_closure # MealyEndConcat
;

mealy_Kleene_closure
:
    mealy_prod  (star='*' | plus='+' | optional='?')
    | mealy_prod # MealyNoKleeneClosure
;

mealy_prod
:
    mealy_atomic colon=':' StringLiteral # MealyProduct
    | mealy_atomic colon=':' Codepoint # MealyProductCodepoints
    | mealy_atomic # MealyEpsilonProduct
;

mealy_atomic
:
    StringLiteral # MealyAtomicLiteral
    | Range # MealyAtomicRange
    | '(' mealy_union ')' # MealyAtomicNested
;
```

In more mathematical terms it could be rewritten as

```
regex ::=
    regex | regex // union
    regex regex // concatenation
    regex* // Kleene closure
    : regex // output/product
    STRING_LITERAL
    ( regex )
```

When presented in such a form, the algebraic data types become apparent. A regular expression is either one of those 6 cases. Hence the above grammar could be represented as Haskell-style algebraic data type

```
type Regex =
    Union Regex Regex
    | Concat Regex Regex
    | Kleene Regex
    | Output Regex
    | Brackets Regex
    | String
```

Every class in Java works like an algebraic product of its member fields. For instance the class

```
class X{
    A a;
    B b;
}
```

is a Cartesian product of classes A and B. Mathematically $X = A \times B$. If a is an instance of A (in other words $a \in A$) and b is an instance of B (mathematically $b \in B$) then the pair (a, b) could become an instance of X (written $(a, b) \in X$). In fact, the notation for constructor $newX(a, b)$ has been inspired by mathematics. Here X is a function that returns memeber of set X ($X : A \times B \to X$) and the pair (a, b) is applied to X. The brackets $()$ should not be interpreted as syntax for function call but rather as syntax for pairs and tuples.

Every interface is the algebraic sum of its implementations. For instance in the snippet below

```
interface X{}
class XA{
    A a;
}
class XB{
    B b;
}
```

the type X is a sum of A and B (written as $X = A \cup B$). The classes XA and XB are nothing more than projection functions. Any instance of $a \in A$ can be applied to $XA(a)$ in order to produce instance of $a \in X$. Similarly for $b \in B$ and $XB(b) \in X$.

As a result the previous Haskell code could be directly translated into Java as

```
interface Regex{}
class Union implements Regex{
    Regex a;
    Regex b;
}
class Concat implements Regex{
    Regex a;
    Regex b;
}
class Kleene implements Regex{
    Regex a;
}
class Output implements Regex{
    Regex a;
}
class String implements Regex{
}
```

While the standard mathematical definition of regular expressions has no place for variables, Solomonoff does indeed extend the language of expressions with variable names. As a result we obtain one more class

```
class Variable implements Regex{
    java.lang.String identifier;
}
```

Another deviation from mathematical definition appears when it comes to braces. It should be pointed out that they do not serve any semantic purpose. They are merely a syntactic construct. In particular, if we used reverse Polish notation, then the brackets would be unnecessary. For instance

$$((\text{'a'}|\text{'b'})\text{'c'})*$$

would become

$$* \cdot |\text{'a''b''c'}$$

where $\cdot$ is an explicit notation of concatenation. The existence of braces is only required by the parser but the abstract syntax tree could function without them. In a sense, every node of the tree works like brackets by itself. Hence we do not need

```
class Brackets implements Regex{
    Regex a;
}
```

Lastly, a few notes should be made about the `String` class. One of the weak points of Java is that it artificially enforces rigid hierarchy of interfaces, even thought it would often be perfectly reasonable to add more interfaces for already existing classes, such as java.lang.String. Duck typing is a much more powerful and mathematically inspired paradigm, than object oriented programming. Thankfully, the typeclass pattern can be used to circumvent this issue. We can wrap any existing type in a new class and then implement any interface of our choice.

```
 class String implements Regex{
     java.lang.String str;
 }
```

It works almost like the `impl` keyword from Rust

```
impl Regex for String{...}
```

The only downside of the typeclass pattern in Java is the inconvenience of overly verbose notation. It forces us to later wrap every `java.lang.String` in our own class, before being able to use it like an instance of `Regex`.

The abstract syntax tree is a recursive data structure and any algorithm that manipulates them must be inductive. One of the most important algorithms is the compilation procedure. It mainly uses functions provided by compiler backend, hence it's exact implementation might look uninteresting.

```
interface Regex{
    G compile();
}
class Union implements Regex{
    Regex a;
    Regex b;
    G compile(){
        return specs.union(a.compile(),b.compile());
    }
}

... the rest is done analogically...

class String implements Regex{
    java.lang.String str;
    G compile(){
        return specs.fromStr(str);
    }
}
```

The G class stands for singly-linked graph that encodes structure of the transducer. The above procedure strictly follows Glushkov's construction. The exact details are implemented by specs, which holds reference to the compiler backend.

To sum up, in the first phase, build system reads all source files and collects abstract syntax trees for every definition. The files are parsed in parallel. A pool of threads needs to be initialised and a queue of pending tasks is formed. This could be represented using the following code.

```
ExecutorService pool = Executors.newWorkStealingPool();
Queue<Future<Void>> queue = new LinkedList<>();
ConcurrentHashMap<String, SolomonoffWeighted> shared =
    new ConcurrentHashMap<>();
for (final File sourceFile : sourceFiles) {
    queue.add(pool.submit(() -> {
        SolomonoffGrammarLexer lexer =
            new SolomonoffGrammarLexer(sourceFile);
        SolomonoffGrammarParser parser =
            new SolomonoffGrammarParser(new CommonTokenStream(lexer));

        parser.addErrorListener(new BaseErrorListener() {...});

        final SolomonoffWeightedParser listener =
            new SolomonoffWeightedParser(shared);
        ParseTreeWalker.DEFAULT.walk(listener, parser.start());
        return null;
    }
}
```

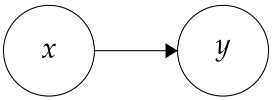
In the next phase, the build system creates a dependency graph. This requires having access to all variables and their syntax trees. Hence it's necessary to wait for the parsing to complete.

```
for (Future<Void> task : queue) {
    task.get(); // blocks until finished
}
```

In the dependency graph, every vertex represents an individual variable with unique identifier. A dependency between two variables is formed when one variable appears in the syntax tree in the definition of another. For instance in the following code

```
x = ’a’ | ’b’
y = ’a’ (x ’b’)*
```

there are two variables x and y, where y depends on x. This would translate to dependency graph on two vertices with one edge



The simplest way to build such a graph would be to scan every syntax tree searching for variables and then add edges accordingly. The inductive procedure could be implemented as follows

```
interface Regex{
    void forEachVariable(Consumer<Variable> callback);
}
class Union implements Regex{
    Regex a;
    Regex b;
    void forEachVariable(Consumer<Variable> callback){
        a.forEachVariable(callback);
        b.forEachVariable(callback);
    }
}

... the rest is done analogically...

class Variable implements Regex{
    void forEachVariable(Consumer<Variable> callback){
         callback.accept(this);
    }
}

class String implements Regex{
    java.lang.String str;
    void forEachVariable(Consumer<Variable> callback){
    }
}
```

The dependency graph is implemented using JGraphT library. It comes with a specialised data structure for directed acyclic graphs that guarantees acyclicity and throws exception whenever a newly added edge violates this property.

```
DirectedAcyclicGraph<String, Object> dependencyOf =
    new DirectedAcyclicGraph<>(null, null, false);
```

The graph can be built by iterating all the definitions stored in shared map.

```
for (Map.Entry<String, SolomonoffWeighted> def : shared.entrySet()) {
    String identifier = def.getKey();
    SolomonoffWeighted syntaxTree = def.getValue();
    syntaxTree.forEachVariable(var->{
        dependencyOf.add(
            var.identifier, // source vertex
            identifier, // target vertex
            new Object() // unique edge
        );
    });
}
```

The above code is simple but it requires to first build the syntax tree and then scan it. As a result every node of the tree is visited twice - first time during creation and second time during scanning. It also requires using the forEachVariable function. It's possible to make this procedure even simper and more efficient. The dependencies could be collected on the spot as the parsing progresses. Because multiple files are parsed in parallel and DirectedAcyclicGraph is not thread-safe, an intermediate temporary storage for edges was necessary. We could first collect all of the edges into a list of pairs.

```
List<Pair<String,String>> dependsOn;
```

The problem with such approach is that certain edges might be duplicated. For instance consider the regular expression that uses x twice

```
x = 'a' | 'b'
y = 'a' (x 'b' x)*
```

This would lead to adding edge Pair.of("y","x") twice into dependsOn. To prevent this we use a set instead of list. The constructor of parser listener takes the following form

```
Set<Pair<String,String>> dependsOn;
ConcurrentHashMap<String,SolomonoffWeighted> idToAST;
Set<Pair<String,String>> dependsOn;
SolomonoffWeightedParser(
         ConcurrentHashMap<String,SolomonoffWeighted> shared,
         Set<Pair<String,String>> dependsOn){
    this.idToAST = shared;
    this.dependsOn = dependsOn;
}
```

The set can be made thread-safe by using the concurrent implementation from Java standard library.

```
ConcurrentHashMap.newKeySet()
```

The parser listener is a state-based device. When a certain rule of the formal grammar is fully parsed, several of the listener's callbacks are fired. For instance when the following rule `FuncDef` is recognized

```
funcs :
    funcs exponential='!!'? ID '=' mealy_union  # FuncDef
    |
;
```

it will first fire `enterFuncDef`, then all the events hiding under `mealy_union` are recursively executed and at the very end comes the `exitFuncDef`. If the expression contained a reference to some variable identifier at any point, then the event `exitMealyAtomicVarID` will be fired at some point during evaluation of `mealy_union` callbacks. As a result the following code can be used to collect all dependencies into a set of pairs.

```
String currentVariable;

@Override
public void enterFuncDef(FuncDefContext funcDefContext) {
    currentVariable = funcDefContext.ID().getText();
}
@Override
public void exitFuncDef(FuncDefContext funcDefContext) {
    SolomonoffWeighted def = stack.pop();
    idToAST.put(currentVariable, def);
}
@Override
public void exitMealyAtomicVarID(MealyAtomicVarIDContext ctx) {
    final String id = ctx.ID().getText();
    dependsOn.add(Pair.of(currentVariable,id));
    stack.push(new Variable(id));
}
```

When the parsing finishes, all of the collected dependencies need to be converted into edges of a directed graph:

```
for (Pair<String, String> dependency : dependsOn) {
    dependencyOf.addEdge(
        dependency.right(),
        dependency.left(),
        new Object()
    );
}
```

If at any point a cyclic dependency is detected, the method addEdge will throw an exception and build system will exit prematurely. This concludes the second phase of build procedure.

The third and last phase is to compile all syntax trees is the correct order and parallelise it as much as possible. The compilation procedure presented before was a minimal prototype that could work with a single variable-free syntax tree. The challenge appears when we wish to compile multiple trees that reference each other through use of variables. The simplest and most naive solution would be to substitute the variables for their appropriate referenced trees. Such substitution could be achieved as follows

```
interface Regex{
     Regex substitute(HashMap<String,Regex> substitution);
}
class Union implements Regex{
     Regex a;
     Regex b;
     Regex substitute(HashMap<String,Regex> substitution){
          return new Union(
              a.substitute(substitution),
              b.substitute(substitution)
          );
     }
}

... the rest is done analogically...

class Variable implements Regex{
     String identifier;
     Regex substitute(HashMap<String,Regex> substitution){
          return substitution.get(identifier);
     }
}

class String implements Regex{
     java.lang.String str;
     Regex substitute(HashMap<String,Regex> substitution){
          return this;
     }
}
```

As a result all syntax trees would become variable-free. This comes at the cost of possibly exponential blow-up in size of certain trees. For example consider the expression

```
f1 = ’a’
f2 = !!f1 f1
f4 = !!f2 f2
f8 = !!f4 f4
f16 = !!f8 f8
f32 = !!f16 f16
```

The automaton defined by $f32$ would accept string of exponential length 32, which requires as many states. The major advantage is that now all trees can be independently compiled in parallel. Despite this, the compilation might in reality get worse than if we serially compiled trees with variables. That’s because the overall number of nodes to be compiled could increase exponentially and the number of truly parallel threads (the number of cores in CPU) is usually small. Hence a better approach is needed.

The right solution is to compile all trees in parallel and block on variables that have not yet been processed.

```
interface Regex{
     G compile(Function<String,Regex> blockingSubstitution);
}
class Union implements Regex{
     Regex a;
     Regex b;
     G compile(Function<String,Regex> blockingSubstitution){
          return specs.union(
              a.compile(blockingSubstitution),
              b.compile(blockingSubstitution)
          );
     }
}

... the rest is done analogically...

class Variable implements Regex{
     String identifier;
     G compile(Function<String,Regex> blockingSubstitution){
          return blockingSubstitution.apply(identifier);
     }
}

class String implements Regex{
     java.lang.String str;
     G compile(Function<String,Regex> blockingSubstitution){
          return specs.fromStr(str);
     }
}
```

The `blockingSubstitution` mimics the `HashMap.get` function but is more generalised. It allows us to provide lambda implementation that lookups a "lazy" map. Such a feature can be obtained by keeping a map of future promises.

```
ConcurrentHashMap<String, SolomonoffWeighted> shared;
ConcurrentHashMap<String,Future<G>> compiled;

... phase 1 and 2 ...

for(String variable : dependencyOf.vertexSet()){
        SolomonoffWeighted syntaxTree = shared.get(variable);
    compiled.put(id, pool.submit(() ->
        syntaxTree.compile((referencedVar)->
            compiled.get(referencedVar).get() // may block
    );
}
```

The above snippet is simple but taking a closer look will reveal that its in fact incorrect. The `compiled.get(referencedVar)` might return `null` if dependencies are not submitted before all their dependants. To prevent this we need to use topological order on directed acyclic graphs. The JGraphT provides a ready-made `TopologicalOrderIterator`.

Every time we perform `pool.submit` the task is added to queue of tasks that will be picked up by some available thread in `ExecutorService`.By submitting those tasks in topological order we also guarantee optimal queue arrangement. The minimal number of threads will block on `get()` function as a result.

3.3. Command-line interface

Rozdział 4

Web technologies

4.1. Editor and REPL console

Creation of user-friendly interface was an important part of the project. The greatest challenge lied in finding the most intuitive way of presenting a complicated and highly advanced system. The main component was the language of regular expressions itself. User should be able to edit its code with ease. The second key feature was the ability to execute the code. In many Turing-complete languages, every expression can be evaluated into some value, which could the be printed back to the user. For example in python’s REPL, typing $2 + 2$ yields 4.

```
>>> 2 + 2
4
```

and after running a regex, user obtains an object containing all matched groups

```
>>> re.compile(’a|b*’).match(’xxaxxbxxbbbx’)
<re.Match object; span=(0, 0), match=’’>
```

In Solomonoff the problem is not so trivial. The regular expressions could in principle be evaluated down to formal languages. For example

```
’a’ (’b’ | ’c’ | ’ef’ ) ’d’
```

would return a language consisting of strings

```
’abd’, ’acd’, ’aefd’
```

The issue wish such approach is that not all languages are finite. The expression

```
’a’*
```

would be evaluated as infinite set

```
'', 'a', 'aa', 'aaa', ...
```

Some regexes, might be finite but of exponential size. For instance

```
('0' | '1') ('0' | '1') ('0' | '1') ('0' | '1')
```

yields set of all bit-strings of length 4. Presenting user with the result in form of formal languages would be often impractical or impossible.

As a result, our REPL does not evaluate expressions. The results of compilation are not printed in any form. Instead the interface is meant to be silent when compilation is successful. Only errors are printed.

There are many different approaches to implement user interface for REPL. One of them would be having a single editor window with all the code in it and the REPL output printed on the margins next to each respective line. This provides a very immersive user experience for Turing-complete languages. For regular expressions i'ts not as spectacular. Instead we decided to use two windows - one for code editor and the other for REPL console. All interaction with regular expressions is performed via special commands built into the console. Those commands could not be used inside the code editor, as they are not part of the language itself. In order to evaluate a transducer user would type the following line into REPL input

```
:eval NAME 'input string'
```

Sometimes, user might want to see all the strings that belong to a given language. While there might be infinitely many of them, it's possible to ask user how large sample to generate. To achieve this the following line can be used

```
:rand_sample NAME of_size NUMBER
```

Automata can also be interpreted as directed graphs. This property makes was used to further enhance user interface. Automaton's graph will be shown after typing this command

```
:vis NAME
```

Those and many other functionalities have been implemented in the browser-based version of REPL.

The implementation is not trivial. One of the ways to achieve such results would be by implementing a parser that could halt mid-parsing. For example user could first type

```
x = ('x' |
```

and hit return button. The parser should notice that the expression is not finished and it has to wait for the next line of input. Then as the user types the next line

```
'y' )
```

a full and valid expression could be recognised and parser could return. This approach is used by some programming languages. It's difficult to implement and requires the grammar to be appropriately structured. We later abandoned this idea due to the problematic nature of Solomonoff's grammar. In particular, it does not use semicolons to separate statements. For example

```
x = 'a'
y = 'b'
```

could be written in a single line

```
x = 'a' y = 'b'
```

The equality sign determines start of new statement. By its very nature, parsing this, requires a lookahead of one token into the future. When the input is read in fragments, line by line, such a lookahead is not possible to obtain. User could first type

```
x = 'a' y
```

which would be recognized by parser as concatenation of string exttt'a' with variable exttty. If the user then types

```
 = 'b'
```

in the upcoming line, then the previous results of parsing would have to be discarded and the entire input reparsed again. Hence we decided to simplify the REPL and assume that every line of input fully defines the entirety of expression. As a result it's not possible to split input into multiple lines when using console. This is not a serious limitation, because multiline expressions could still be written in the editor window instead of console.

The division of user interface into editor and console has one more advantage. It closely mimics the layout of command-line interface, where the typical workflow is to edit source code in local files using any text editor of user's choice and the REPL is kept open all the time alongside the editor. Many existing modes for Emacs follow similar convention.

The REPL is implemented on the server-side as a REST API endpoint.

```
@PostMapping("/repl")
public ReplResponse repl(HttpSession httpSession,
    @RequestBody String line)
```

Every user has their own instance of REPL

```
Repl repl = (Repl) httpSession.getAttribute("repl");
```

which holds a reference to the compiler and a set of built-in commands

```
 public static class Repl {
    private static class CmdMeta<Result> {
        final ReplCommand<Result> cmd;
        final String help;
        final String template;

        private CmdMeta(ReplCommand<Result> cmd,
               String help, String template) {
            this.cmd = cmd;
            this.help = help;
            this.template = template;
        }
    }

    HashMap<String, Repl.CmdMeta<String>> commands;
    OptimisedHashLexTransducer compiler;
}
```

whenever user types some command on the REPL console

```
:cmd arg1 arg2 arg3
```

it gets parsed as

```
String firstWord = "cmd";
String remaining = "arg1 arg2 arg3";
```

and then the appropriate command implementation is looked up in the map

```
final Repl.CmdMeta<String> cmd = commands.get(firstWord);
return cmd.cmd.run(httpSession, compiler, log, debug, remaining);
```

The rest controller contains implementations of many such commands

```
public static final ReplCommand<String> REPL_LIST = ...
public static final ReplCommand<String> REPL_EVAL = ...
public static final ReplCommand<String> REPL_RUN = ...
public static final ReplCommand<String> REPL_EXPORT = ..
public static final ReplCommand<String> REPL_IS_DETERMINISTIC = ...
public static final ReplCommand<String> REPL_LIST_PIPES = ...
public static final ReplCommand<String> REPL_EQUAL = ...
public static final ReplCommand<String> REPL_RAND_SAMPLE = ...
public static final ReplCommand<String> REPL_CLEAR = ...
public static final ReplCommand<String> REPL_UNSET = ...
public static final ReplCommand<String> REPL_RESET = ...
public static final ReplCommand<String> REPL_LOAD = ...
public static final ReplCommand<String> REPL_VIS = ...
```

All of those definitions above are lambda expressions that use library functions of the compiler. The parameters taken by those lambda expressions are as follows

```
public interface ReplCommand<Result> {
    Result run(
        HttpSession httpSession,
        OptimisedHashLexTransducer compiler,
        Consumer<String> log,
        Consumer<String> debug,
        String args) throws Exception;
}
```

As an example, consider the command

```
:eval f 'abc'
```

which evaluates transducer extttf for input string exttt'abc'. On the frontend JavaScript will perform REST query

```
const response = await fetch('repl', {
     method: 'POST',
     body: ":eval f 'abc'"
})
```

which will be received by server

```
@PostMapping("/repl")
public ReplResponse repl(HttpSession httpSession,
        @RequestBody String line) {
    Repl repl = (Repl) httpSession.getAttribute("repl");
    final String result = repl.run(
        httpSession,
        line, // ":eval f 'abc'"
        s -> out.append(s).append('\n'), // console output
        s -> { } // debug logs are not displayed
    );
    ...
}
```

and the extttrepl.run method will query the appropriate implementation to call for the exttt:eval command.

```
final String firstWord = "eval";
final String remaining = "f ’abc’";
final Repl.CmdMeta<String> cmd = commands.get(firstWord);
return cmd.cmd.run(httpSession, compiler, log, debug, remaining);
```

This in turn will trigger the following lambda function

```
ReplCommand<String> REPL_EVAL =
        (httpSession, compiler, logs, debug, args) -> {
    String[] parts = args.split("\\s+", 2); // f ’abc’
    String name = parts[0]; // f
    String input = parts[1]; // ’abc’
    G transducer = compiler.getTransducer(name);
    String output = compiler.specs.evaluate(transducer, input);
    return output == null ? "No match!" : output;
};
```

The output is sent back to JavaScript in form of JSON

```
const replResult = JSON.parse(await response.text())
```

Remaining commands are implemented in a similar way.

Several of the functions may require access to HTTP session. In particular it’s worth analysing the exttt:load command. It’s purpose is to emulate the process of loading source code from file. Whenever user types some code in the editor, it needs to be transported to server, then parsed and compiled. All the defined transducers results need to be saved for later use. The simplest way of achieving this, would be by extending the REST API as

```
class ReplInput{
     String command;
     String editorContent;
}
public ReplResponse repl(
    HttpSession httpSession,
    @RequestBody ReplInput)
```

and query it using

```
const response = await fetch(’repl’, {
    method: ’POST’,
    body: {
        command: replCommand,
        editorContent: editor.getValue()
    }
})
```

The downside of such solution is that the editor content could become large and sending it would require more internet bandwidth and time. Often user only wants to execute simple short REPL commands that do not require sending the entire code. Sometimes the code might be required but resending it might be omitted as long as user has not modified it. Hence the process of uploading code to the server has been delegated to a separate REST call.

```
const response = await fetch('upload_code', {
     method: 'POST',
     body: code
})
```

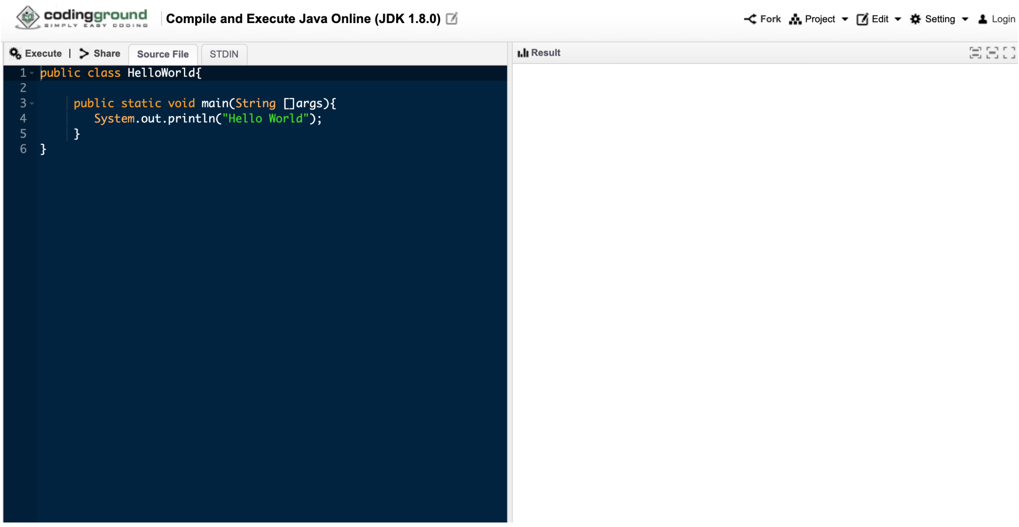
The code is then stored in HTTP session, so the REST endpoint has a very simple implementation

```
@PostMapping("/upload_code")
public void uploadCode(HttpSession httpSession,
        @RequestBody String text) {
    httpSession.setAttribute("code", text);
}
```

Aside from the editor and REPL there is one more window on the webpage. It is dedicated for tutorial and short documentation. While it does not enhance the functionality of the website per se, it plays an important role. The Solomonoff compiler is a very niche and specialised tool. There are no similar tools and any user coming to the website is not expected to be familiar with its usage. The primary purpose of the website is not to be a replacement for user's IDE and terminal. Instead it serves as an all-in-one introductory tutorial, interactive playground and a marketing campaign. We want to make the learning materials easily accessible and abundant. Building a strong community is the back-bone of every open-source project.

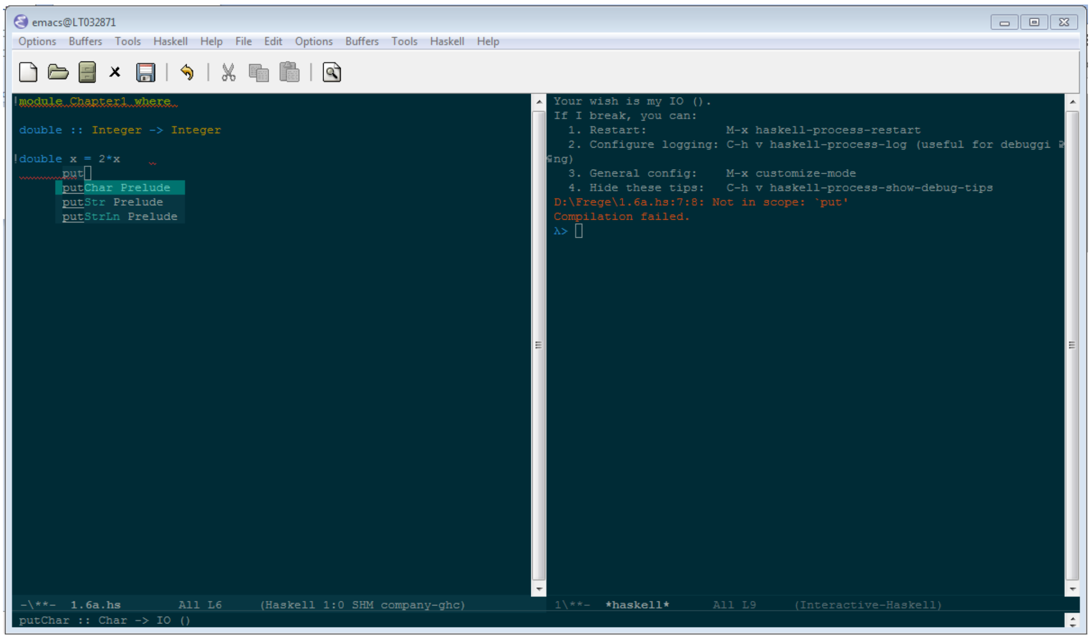
4.2. Design

The website has seen numerous design changes. Since the beginning we knew there must be a way to interact with the code but the exact best way of presenting it to the user was not so self-evident. There exist numerous different approaches that are highly dependent on the language. For example the online Java compiler consists of only two windows - one for code and one for compiler output.



```
public class HelloWorld{

     public static void main(String []args){
        System.out.println("Hello World");
     }
}
```

Java does not have REPL, hence there is no need to implement console input. A slightly different has been taken by Haskell mode for emacs.



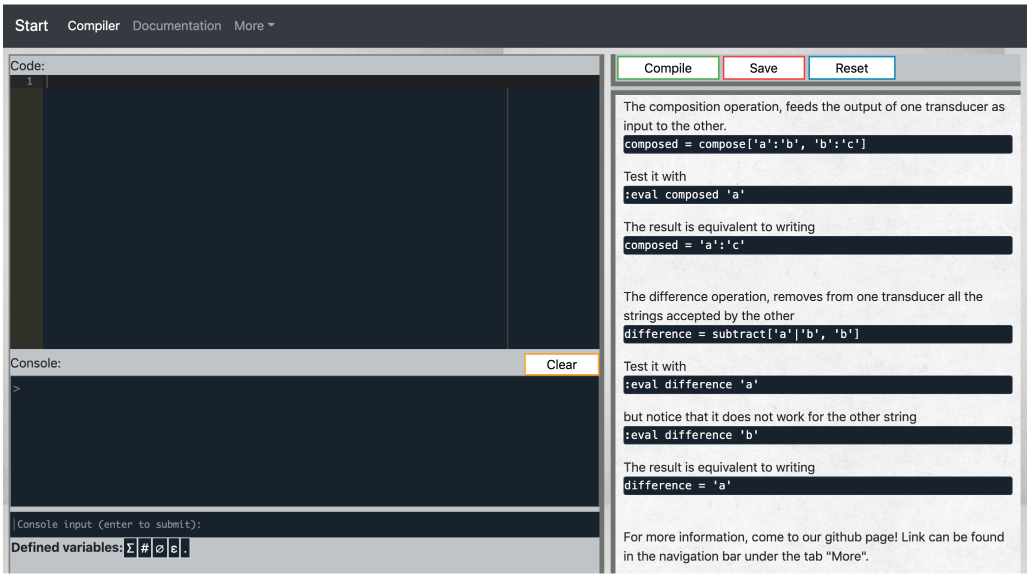
emacs@LT032871
Options Buffers Tools Haskell Help File Edit Options Buffers Tools Haskell Help

```
module Chapter1 where

double :: Integer -> Integer

double x = 2*x
      put
      putChar Prelude
      putStr Prelude
      putStrLn Prelude
```

```
Your wish is my IO ().
If I break, you can:
  1. Restart:           M-x haskell-process-restart
  2. Configure logging: C-h v haskell-process-log (useful for debuggi
ng)
  3. General config:    M-x customize-mode
  4. Hide these tips:   C-h v haskell-process-show-debug-tips
D:\Frege\1.6a.hs:7:8: Not in scope: `put'
Compilation failed.
λ>
```

```
-\**-  1.6a.hs        All L6      (Haskell 1:0 SHM company-ghc)
1\**-  *haskell*      All L9      (Interactive-Haskell)
putChar :: Char -> IO ()
```

Here it is indeed possible to evaluate smaller snippets of code in the right window, while the left one is solely dedicated to editing local files that can be saved persistently. This is the approach that we settled for in our online playground for Solomonoff.



The composition operation, feeds the output of one transducer as input to the other.

```
composed = compose['a':'b', 'b':'c']
```

Test it with

```
:eval composed 'a'
```

The result is equivalent to writing

```
composed = 'a':'c'
```

The difference operation, removes from one transducer all the strings accepted by the other

```
difference = subtract['a'|'b', 'b']
```

Test it with

```
:eval difference 'a'
```

but notice that it does not work for the other string

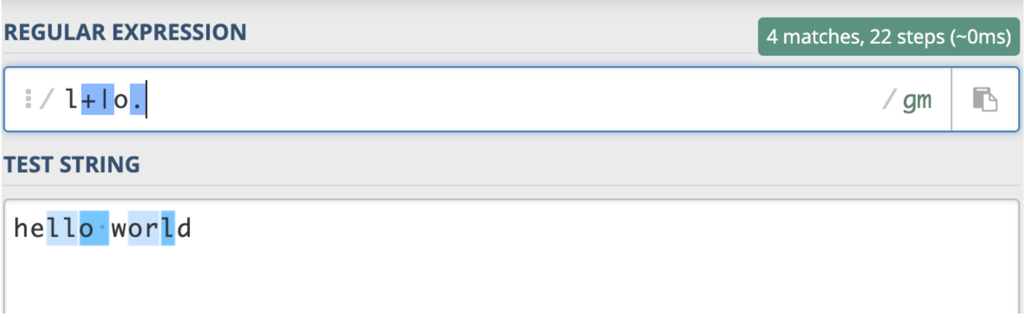
```
:eval difference 'b'
```

The result is equivalent to writing

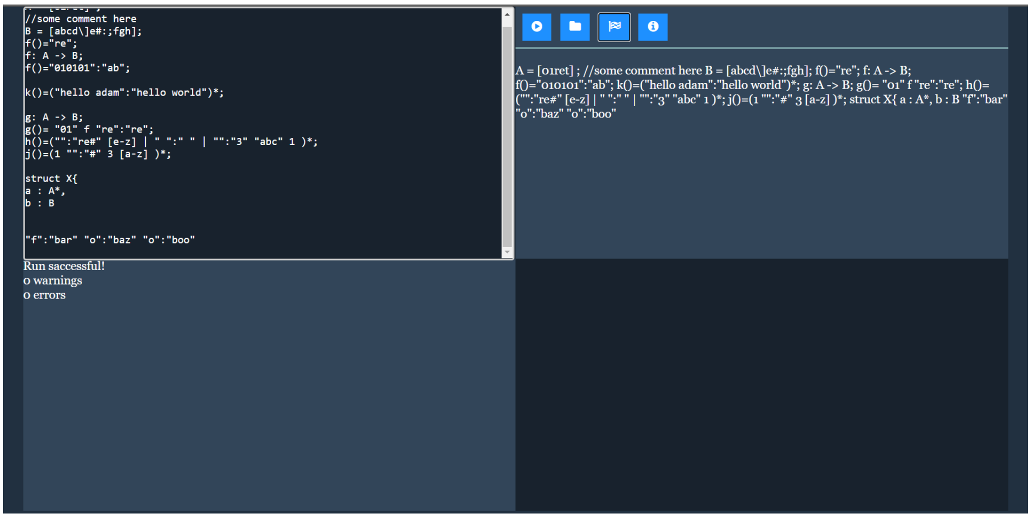
```
difference = 'a'
```

For more information, come to our github page! Link can be found in the navigation bar under the tab "More".

Before reaching such final version we have experimented with another approach that seemed more natural for regular expressions. Here is an example of a similar website that evaluates UNIX regexes.



Initially we tried to mimic such approach with some modifications. Solomonoff is much more complex than UNIX regexes. It allows variables, functions, comments and the overall code could consist of multiple lines. Hence a dedicated multiline editor window was required like in case of Java or Haskell.

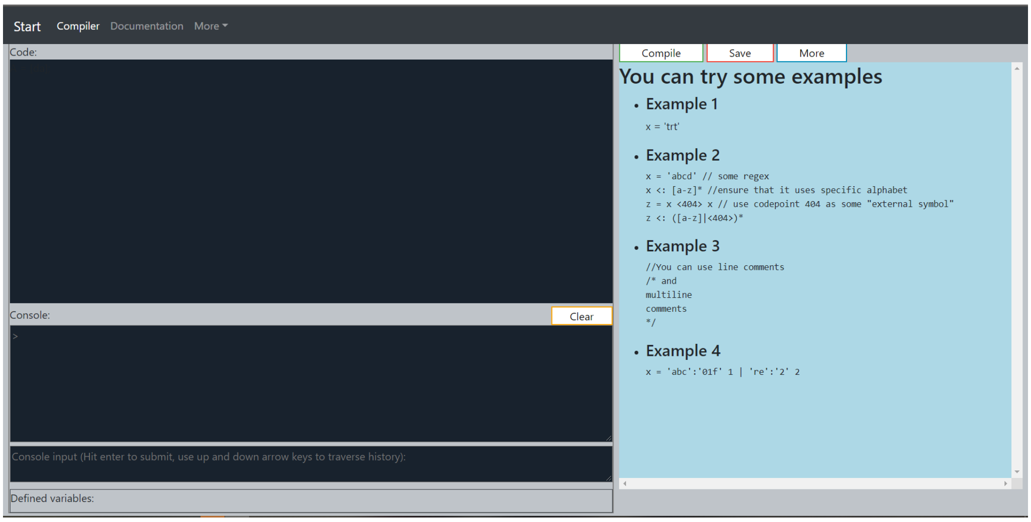


```
//some comment here
B = [abcd\]e#:;fgh];
f()="re";
f: A -> B;
f()="010101":"ab";

k()=("hello adam":"hello world")*;

g: A -> B;
g()= "01" f "re":"re";
h()=("":"re#" [e-z] | " ":" " | "":"3" "abc" 1 )*;
j()=(1 "":"#" 3 [a-z] )*;

struct X{
a : A*,
b : B


"f":"bar" "o":"baz" "o":"boo"
```

Run saccessful!
0 warnings
0 errors

A = [01ret] ; //some comment here B = [abcd\]e#:;fgh]; f()="re"; f: A -> B; f()="010101":"ab"; k()=("hello adam":"hello world")*; g: A -> B; g()= "01" f "re":"re"; h()= ("":"re#" [e-z] | " ":" " | "":"3" "abc" 1)*; j()=(1 "":"#" 3 [a-z])*; struct X{ a : A*, b : B "f":"bar" "o":"baz" "o":"boo"

The upper left window was dedicated to code. The lower left was meant to hold the test input string and the upper right would show the resulting transducer output. Such an approach seemed perfect at the beginning, when Solomonoff was still in early development. Over time, the language became increasingly complex. Several features were added that allowed for visualizing graphs of automata, sampling their languages, testing their formal properties and querying more complex information about them. The interface couldn't keep up with the full range of possibilities offered by he compiler. Hence, we decided to scrap the idea with two input-output windows and tried to emulate console-like REPL instead. The first version looked as follows



Console input (Hit enter to submit, use up and down arrow keys to traverse history):

Defined variables:

You can try some examples

- Example 1

```
x = 'trt'
```

- Example 2

```
x = 'abcd' // some regex
x <: [a-z]* //ensure that it uses specific alphabet
z = x <404> x // use codepoint 404 as some "external symbol"
z <: ([a-z]|<404>)*
```

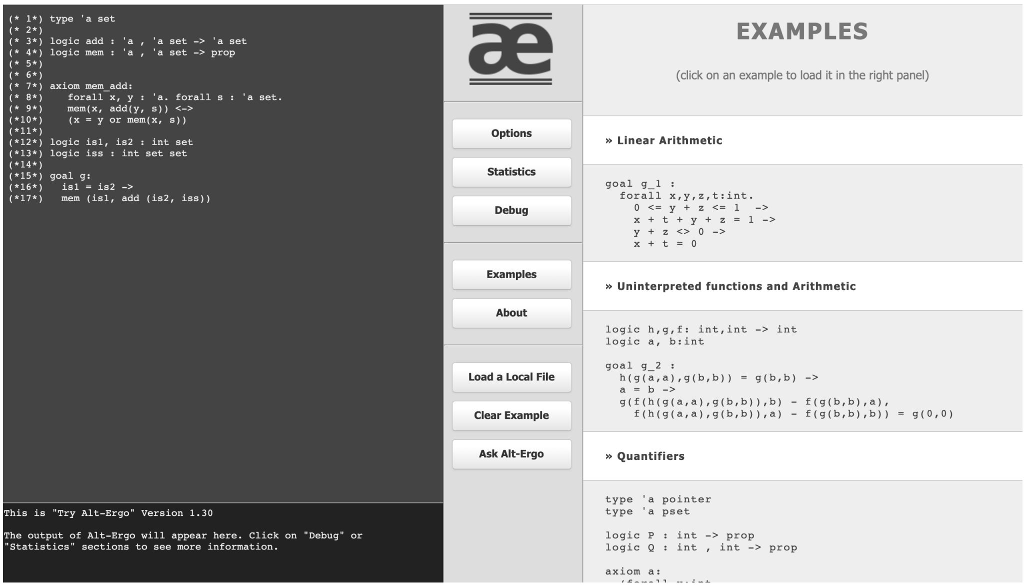
- Example 3

```
//You can use line comments
/* and
multiline
comments
*/
```

- Example 4

```
x = 'abc':'01f' 1 | 're':'2' 2
```

It was at this point when we first tried to add a window for documentation. A great source of inspiration was the Alt-Ergo online playground. It comes with many helpful examples, which can be automatically copied to the editor upon clicking on them.



```
type 'a set

logic add : 'a , 'a set -> 'a set
logic mem : 'a , 'a set -> prop


axiom mem_add:
   forall x, y : 'a. forall s : 'a set.
   mem(x, add(y, s)) <->
   (x = y or mem(x, s))

logic is1, is2 : int set
logic iss : int set set

goal g:
  is1 = is2 ->
  mem (is1, add (is2, iss))
```

```
This is "Try Alt-Ergo" Version 1.30

The output of Alt-Ergo will appear here. Click on "Debug" or
"Statistics" sections to see more information.
```

(click on an example to load it in the right panel)

» Linear Arithmetic

```
goal g_1 :
  forall x,y,z,t:int.
    0 <= y + z <= 1  ->
    x + t + y + z = 1 ->
    y + z <> 0 ->
    x + t = 0
```

» Uninterpreted functions and Arithmetic

```
logic h,g,f: int,int -> int
logic a, b:int

goal g_2 :
  h(g(a,a),g(b,b)) = g(b,b) ->
  a = b ->
  g(f(h(g(a,a),g(b,b)),b) - f(g(b,b),a),
    f(h(g(a,a),g(b,b)),a) - f(g(b,b),b)) = g(0,0)
```

» Quantifiers

```
type 'a pointer
type 'a pset

logic P : int -> prop
logic Q : int , int -> prop

axiom a:
```

Later we extended this idea to a full tutorial with explanations, rather than a simple list of copyable examples.

4.3. Ace editor

At the same time as the layout of page evolved, we also actively developed the editor itself. While initially we used a simple HTML textarea element, we soon replaced it with the Ace editor. It allowed for integrating many additional features such as syntax highlighting, auto-suggestions, code snippets and marking errors. The core component of working with Ace was the necessity of developing our own syntax highlighting. All the previously mentioned come out-of-the box for existing languages, like JavaScript, C and Python. The matters get much more complicated, when one attempts to define their custom language. Ace documentation for syntax developers tends to be rather sparse.

There exists a common framework followed by most syntax highlighters. Their configuration consists of two main components - the highlighting rules and the styling rules. The former consist of a set of regular expressions. Any region matched by a certain expression is marked with a list of styles. Then each style decides about the colour. Many existing text editors come with their own syntax highlighter and the required configurations may differ, albeit each format could be automatically converted into any other.

By using the Iro editor, developer can develop only a single set of rules and then have them automatically converted to any other syntax highlighter. This includes support for Ace, SublimeText, TextMate and Atom. Most of the other editors, like Intellij, Eclipse, Notepad++ etc. use one of the above standards.

The general format of Iro is as follows

```

styles [] {

     .comment : style {
          color                 = #688557
          italic                = true
          ace_scope             = comment
          textmate_scope        = comment
          pygments_scope        = Comment
     }

}

main : context {

     : pattern {
          regex          \= (//.*)
          styles []       = .comment;
     }

}
```

each pattern matches some fragment of code and assigns a style to it. Then the style defines colour and scope. The scope is later used as a hint for various other tools that rely on syntax recognition. The colours themselves are also a mere hint. The end user might use an editor that supports various colour palettes. In particular many editors come with optional dark and light theme. Depending on user's choice, our colouring might be overriden. Hence the assigned scope is more important than the colour hint.

Most of the language grammars are context-free and cannot be recognized with a simple regular expression. Hence syntax highlighters allow for using stacks. A good example of this are the block comments. The regular expressions that apply outside of comments should not apply inside them. Iro allows developer to manipulate the stack using push and pop command.

```
: inline_push {
     regex          \= (/\*)
```

```
    styles []       = .comment;
    default_style   = .comment
    : pop {
        regex       \= (\*/)
        styles []    = .comment;
    }
}
```

Solomonoff's syntax highlighter uses stack to correctly recognize comments and string literals enclosed in quotes and angle brackets. The rest of syntax highlighter is fairly simple and only matches key characters, such as equal signs, brackets, Kleene stars and union vertical pipes as well as variable identifiers.

After the syntax highlighter was developed, Iro automatically generated necessary Ace files. All those configurations are written JavaScript. In order to make them work, it's necessary to clone Ace's repository and compile the custom syntax along with the rest of sources. The compiled project needs to be hosted on website along with remaining JavaScript files. The Ace editor can be initialized using the following lines

```
var editor = ace.edit("editor");
editor.session.setMode("ace/mode/mealy");
```

where extttmealy is the name of our custom syntax. The editor has been styled using monokai theme

```
editor.setTheme("ace/theme/monokai");
```

because its dark palette of colours gives the website a sharp and modern look. The dark theme is preferred by most users and is very popular nowadays. Moreover it's more relaxing to look at and doesn't irritate the eye. This point is especially important, because Solomonoff is targetted at tech-oriented audiences, so there is a good chance that our users will spend many hours looking at the editor.

Solomonoff comes with several built-in functions. To make the interface more intuitive and ergonomic the editor needs to provide auto-suggestions with the full range available functions. Below is a list presenting some of the more important options.

```
editor.setOptions({
    enableBasicAutocompletion: [{
        getCompletions: (editor, session, pos,
         prefix, callback) => {
            callback(null, [{
                name: 'subtract[',
                value: 'subtract[]',
                score: 1,
                meta: 'difference of two languages'
            },
            {
                name: 'rpni!(',
                value: 'rpni!()',
                score: 1,
                meta: 'RPNI inference algorithm'
            },
            {
                name: 'rpni_mealy!(',
                value: 'rpni_mealy!()',
                score: 1,
                meta: 'RPNI for Mealy machiens'
            },
            {
                name: 'ostia!(',
                value: 'ostia!()',
                score: 1,
                meta: 'OSTIA inference for transducers'
            },
            {
                name: 'compose[',
                value: 'compose[]',
                score: 1,
                meta: 'transducer composition'
            },
            {
                name: 'inverse[',
                value: 'inverse[]',
                score: 1,
                meta: 'transducer inversion'
            }
            ]);
        },
    }],
    enableSnippets: true,
    enableLiveAutocompletion: true
});
```

The extttvalue field is the text that autocompletion will produce when selected. The extttmeta argument provides a short explanation that will be shown to the user. We decided to set extttenableLiveAutocompletion so that the dropdown box with all available functions will automatically show up as soon as the user starts typing. Some less experienced users might not be aware that the suggestions can be triggered manually by pressing extttCTRL+SPACE. The downside is that in some contexts the autosuggestion will pop up despite not being necessary. This could irritate some users. Perhaps the best approach would be to make the auto-suggestions configurable. A user could set the editor properties according to their own liking. The only problem was that adding user customizations would increase the complexity of final product. Our goal was not to create a fully functioning IDE. The website is meant to work only as a showcase. Hence the final decision was to make the website as friendly to the newcomers as possible even at the cost of making the experienced users less comfortable. The general consensus was that users that like our product will quickly download the compiler locally and use it in conjunction with their own editor of choice.

Initially, the Ace was only used in the main editor window. The REPL console would be made of two text areas, one serving as editable input line and the other for console output, which was permanently set as non-editable. By design, the REPL commands could only be used in console and placing them in the main editor would only result in syntax errors. For instance

```
:eval f ’input’
```

would only work in REPL, despite not being a valid Solomonoff code per se. As auto-suggestions were added, it became apparent that showing REPL commands tin main editor would be very misleading

```
editor.setOptions({
     enableBasicAutocompletion: [{
          getCompletions: (editor, session, pos,
          prefix, callback) => {
               callback(null, [
          ...
               {
                    name: ’:eval’,
                    value: ’:eval’,
                    score: 1,
                    meta: ’evaluate transducer’
               }
          ...
               ]);
          },
     }],
     enableSnippets: true,
     enableLiveAutocompletion: true
});
```

On the other hand, not showing any hints related to REPL commands would seems like major shortcoming of the online playground. To address this issue it was later decided tha the REPL input line should also use Ace. As a result the website ended up with two instances of Ace editor. Both very similar to each other. The only difference being that auto-suggestions in REPL input would also display REPL commands, whereas the main code editor would not.

Using Ace in REPL input also happened to solve another problem. Every REPL command would have their own format of argument. For example the exttt:eval would take transducer name and then some input string. The visualization only needs transducer name

```
:vis f
```

The most unintuitive is the random sample command which has two possible formats

```
:rand_sample f of_size 10
```

and

```
:rand_sample f of_length 10
```

The former generates 10 random member strings, whereas the latter generates all member strings up to length of 10. The extttof_size and extttof_length argument decides, which fo the two modes of generation to use. The initial idea to address this issue was to add user help displayed by the exttt:? command.

```
> :?
:vis [ID]
    Shows graph diagram of automaton
:rand_sample [ID] [of_size/of_length] [NUM]
    Generates random sample of input:output pairs produced
    by ths transducer
:ls
    Lists all currently defined transducers
:clear
    Clears REPL console
:run [ID] [STRING]
    Runs pipeline for the given input
:unset [ID]
    Deletes a variable
:is_det [ID]
    Tests whether transducer is deterministic
```

```
:eval [ID] [STRING]
    Evaluates transducer on requested input
...
```

User could the consult this cheat sheet to determine format of arguments for each command. Such a solution was simple but it certainly didn't make for the most ergonomic user interface. With Ace it became possible to use code snippets instead. Below are a few examples.

```
getCompletions: (editor, session, pos, prefix, callback) => {
     callback(null, [
     {
          name: ':eval',
          value: ':eval',
          snippet: ':eval ${1:transducer_name} \'${2:input_string}\'',
          score: 1,
          meta: 'evaluate transducer'
     },
     {
          name: ':rand_sample of_size',
          value: ':rand_sample',
          snippet: ':rand_sample ${1:transducer_name} of_size ${2:number}',
          score: 1,
          meta: 'randomly generate sample'
     },
     {
          name: ':rand_sample of_length',
          value: ':rand_sample',
          snippet: ':rand_sample ${1:transducer_name} of_length ${2:number}',
          score: 1,
          meta: 'randomly generate sample'
     },
     ]);
}
```

The structure of arguments for each command was intuitively encoded in form of blanks that need to be filled in each snippet. Each blank is specified using the exttt${} braces.

4.4. WebAssembly

The early versions of compiler were written in C. With help of Emscripten it was possible to then port the code to WebAssembly and call every function directly from JavaScript. Such a solution was preferable, because everything worked on client-side and we could host the website free-of-charge on GitHub Pages. The WASM interface exposed two functions

```
function compile(){
        console.log(input.value)
        Module.ccall(’compile’,
        ’void’,
        [’string’],
        [input.value])
}
function runMealy(){
        console.log(output.value)
        var c = Module.ccall(’run_global’,
        ’string’,
        [’string’],
        [output.value])
        console.log(c);
        output.value = c;
}
```

First one received a string of source code and the other executed defined transducer. Emscripten also generated a layer of JavaScript code in

```
<script async type="text/javascript" src="web_mealy_compiler.js"></script>
```

that smoothened process of WebAssembly integration. At that stage, the compiler was simple and such minimalist API was enough. Later a lot of new functionalities were added and the project requirements shifted to Java. Several approaches for compiling Java code to WASM were attempted but with unsatisfying results.

The first framework we attempted to use was JWebAssembly. We successfully compiled and integrated toy projects but as complexity increased, the limitations became more apparent. The project page itself admitted that JWebAssembly is not yet production ready. Parts of Java standard library were missing, exceptions and threads had limited support and any attempts at converting our compiler to WASM resulted in uncountable of errors.

The next library we attempted was CheerpJ. It was the most promising option. The project compiled successfully and could run in browser. The problem was that support for WASM is still at experimental stage and instead CheerpJ generated JavaScript transcription of our code wrapped in heavyweight runtime. The framework is a commercial product and is primarily targeted at porting Swing/JavaFX applications to browser. There was no documentation explaining how to expose JavaScript API or how to call any Java functions programmatically. There was also no way of manually writing HTML and CSS. Instead the produced applications mimicked typical desktop graphical interface. If we chose to use CheerpJ, we'd be forced to write user interface in Java Swing technology.

The last available option was to use TeaVM. It suffered similar problems to JWebAssembly. Parts of Java standard library were not supported. Parts of compiler code had to be rewritten to use workaround functions. For instance the

```
X x = hashMap.computeIfAbsent(k,key->new X());
```

had to be rewritten as

```
if(hashMap.conatinsKey(k)){
    x = hashMap.get(k);
}else{
    x = new X();
    hashMap.put(k,x);
}
```

Another problem was with porting all the libraries and dependencies. Parts of ANTLR code could not be compiled and we resorted to cloning the original ANTLR repo and manually rewriting parts of its code. Clever workarounds had to be implemented but in some cases, the code could not work in browser in any form and had to be removed. After enough modifications, our code compiled successfully. When ran in browser, the execution time was prohibitively slow and even the simplest tasks could not be completed in a satisfying time.

As a result, we were forced to forego the idea of compiling Java to WASM and switched to using backend instead. Java has long and extensive history of being used as server-side language with plethora of tools and frameworks available - Spring being one of the most popular.