



UNIWERSYTET
IM. ADAMA MICKIEWICZA
W POZNANIU

Wydział Matematyki i Informatyki

Bohdan Bondar, Marcin Jałowski, Aleksander Mendoza-Drosik
Numer albumu: 432778, 434701, 434749

Solomonoff — Finite state transducer compiler
Solomonoff — kompilator transduktorów skończonego stanu

Praca inżynierska na kierunku **Computer Science**
napisana pod opieką
dra. Bartłomieja Przybylskiego

Poznań, Luty 2021

Poznań, Luty 9, 2021 r.

Oświadczenie

Ja, niżej podpisana **Bohdan Bondar, Marcin Jałowski, Aleksander Mendoza-Drosik**, studentka Wydziału Matematyki i Informatyki Uniwersytetu im. Adama Mickiewicza w Poznaniu oświadczam, że przedkładaną pracę dyplomową pt. *Solomonoff — Finite state transducer compiler* 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

Solomonoff jest wyspecjalizowanym kompilatorem transduktorów skończonych. Użytkownik jest w stanie budować i manipulować automatami przy pomocy specjalnego języka wyrażeń regularnych. Implementacja bazuje na rozszerzeniu konstrukcji Głuszkowa. Projekt ten stara się zaadresować wiele problemów w uprzednio istniejących bibliotekach, takich jak OpenFST i RE2, lecz również poszerzyć ich zdolności o kilka innowacyjnych i niespotykanych wcześniej rozwiązań. Jednym z głównych celów Solomonoffa jest łatwość i przystępność nawet dla lingwistów i innych nietechnicznych użytkowników. Z kompilatora można korzystać w całości z poziomu wyrażeń regularnych, bez konieczności pisania własnego kodu w Javie. Kompilator współdziała ze zintegrowanym systemem do budowania i zarządzania kodem oraz jego demo jest dostępne na interaktywnej stronie internetowej, z której można korzystać bez konieczności posiadania lokalnej instalacji kompilatora. Solomonoff wspiera programowanie z poziomu interaktywnej konsoli REPL, dostępnej zarówno z lokalnego wiersza poleceń jak i izolowanej wersji przedlądarkowej. W interfejsie wiersza poleceń możliwa jest praca z kodem zawartym w wielu plikach źródłowych. Zamiast wykorzystywać gotowe rozwiązanie, takie jak GNU Make, Ninja lub Maven, zarządzanie projektem jest oparte na systemie specjalnie przystosowanym do Solomonoffa. Wspierana jest kompilacja równoległa, odnajdowanie zależności i budowanie paczek. Mniej zaawansowani użytkownicy mogą korzystać z uproszczonego i bardziej ergonomicznego środowiska w przeglądarce. Dostępny jest tam graficzny interfejs użytkownika zaprojektowany z myślą o estetyce i oparty na technologii edytora Ace działający wraz z serwerem Spring.

Słowa kluczowe: kompilator, automaty, transduktory, java, build system, REPL, strona internetowa

Abstract

Solomonoff is a dedicated compiler of finite state transducers. User can assemble and manipulate the automata with a domain specific language of regular expressions. Its implementation is based on an extension of Glushkov's construction. The project addresses many problems present in previously existing libraries such as OpenFST and RE2 but it also comes with several innovative features not seen before. One of the main goals of Solomonoff is to be easily accessible to linguists and non-technical users. Anything could be done without writing a single line of Java code, the compiler comes with a well integrated build system and a convenient online playground can be accessed from browser without any need for local installation. Solomonoff supports development in interactive REPL environment, which has both local command-line version and its sandboxed browser-based equivalent. The terminal interface allows for modularised programming and storing code across multiple files. Instead of using an existing tool like GNU Make, Ninja or Maven, the build system is specially designed for Solomonoff. It supports parallel compilation, dependency resolving and package management. Less experienced users may opt to use simpler and more ergonomic programming environment from their browser. It provides graphical user interface designed with care for aesthetics. Its implementation is based on Ace editor and communicates with Spring backend.

Keywords: compiler, automata, transducers, java, build system, REPL, website

Contents

Chapter 1. Introduction	6
Chapter 2. Transducers	9
2.1 Mealy automata and transducers	9
2.2 Gluszkov's construction	25
2.3 Code specification	35
Chapter 3. Build system	39
3.1 Advantages of using build systems	39
3.2 Overview of selected build systems	40
3.2.1 GNU Make	40
3.2.2 Ninja	42
3.2.3 Maven	43
3.3 Solomonoff	46
3.3.1 System description	46
3.3.2 Problem	46
3.3.3 Solutions	47
Chapter 4. Web technologies	56
4.1 Editor and REPL console	56
4.2 Spring backend	59
4.3 Design	64
4.4 Ace editor	70
4.5 WebAssembly	77

CHAPTER 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 known before.

The most innovative achievements is the lexicographic arctic semiring of weights[1], 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 its derivatives. We contributed a specialized algorithm for inference of transducers to the LearnLib library. In particular, we implemented OSTIA for efficient learning of deterministic transducers. For nondeterministic ones we developed our own OFTIA algorithm, which was not known before. Additionally we provide an alternative version of OSTIA extended with heuristic search.

All those features together make Solomonoff a unique library that stands out from the alternatives. We support most of the features of UNIX regexes. It’s possible to achieve mechanics equivalent to those of look-aheads and look-behinds, which is unusual for an automata-based regex engine. The key to Solomonoff’s expressiveness 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 in collaboration with University of New York. There is a vast intersection of features supported by both libraries, although OpenFST puts heavy emphasis on probabilistic transducers, whereas Solomonoff has better support for formal verification and

machine learning. Our project won't be a replacement for OpenFST in the foreseeable future but there are areas where Solomonoff performs better. This is particularly palpable in efficiency as indicated by several of our benchmarks.

Unlike most other automata libraries, Solomonoff's development focuses on the compiler and regular expressions instead of library API. Everything can be achieved without writing Java code. Such approach also allows the developers for much greater flexibility, because the internal implementation can be drastically changed at any time, without breaking existing regular expressions. The public API is minimalist. As a result backwards-compatibility is rarely an issue.

Another feature of Solomonoff, which is uncommon among regular expression engines is its integrated build system. Our compiler was designed for large codebases consisting of multiple files with dependencies. 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.

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 an online playground where visitors could test Solomonoff and experiment without any setup required. The backend technology we used is Spring Boot, because it allows for convenient integration with existing API in Java. The website provides downloads of prepackaged JAR files as well as syntax highlighting rules for several popular text editors.

To allow for easy experimentation and to lower the barrier of entry, Solomonoff comes with an interactive shell. It allows for efficient and convenient work with the compiler from command-line interface, where local changes can be hot-swapped and reloaded on a live compiler session. Developers could then test their regular expressions without the need to fully recompile their project.

The REPL console can be easily and painlessly explored without local installation. The website comes with an online version of compiler, which can be accessed from any major browser (IE not supported). Most of the command-line features can be used in the online environment, with a few exceptions of features specific only to locally-running instances (especially those allowing for saving and reading files).

The online interface can serve as a user-friendly playground for newcomers. It is not meant to be a replacement for a locally running compiler. Due to the necessity of communicating with the backend server and limited computational resources shared among possibly a large number of users, the online REPL has

major performance limitations. The command-line interface has been designed and optimised with the purpose of providing high efficiency for codebases containing thousands of lines. It supports parallel compilation and caching of previously compiled units.

Developers have the ability to modularise their projects. Precompiled automata can be stored and loaded from packages. The build-system is capable of resolving dependencies.

Solomonoff is an open source project (uses WTFPL licence) available from the GitHub repository under the following link

<https://github.com/aleksander-mendoza/SolomonoffLib>

CHAPTER 2

Transducers

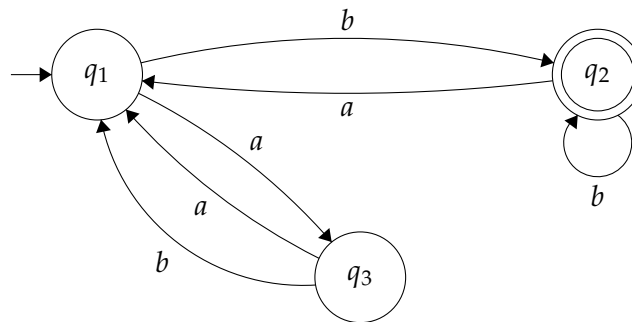
ALEKSANDER MENDOZA-DROSIK

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 tool that focused only on deterministic Mealy automata. Such a model was very limited and it quickly became apparent that certain extensions would be necessary. In the end we implemented a compiler that supports nondeterministic functional weighted symbolic transducers.

2.1 MEALY AUTOMATA AND TRANSDUCERS

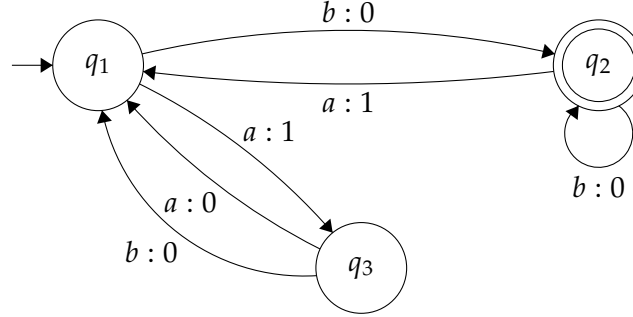
The standard definition of automaton found in introductory courses [2, 3, 4] states that finite state automaton is a tuple $(Q, I, \delta, F, \Sigma)$ where Q is the set of states, $I \subset Q$ are the initial states, $F \subset Q$ are the final (or accepting) states, $\delta : Q \times \Sigma \rightarrow Q$ is the transition function and Σ is some alphabet. The automaton can either accept or reject a certain input string. Below is an example in the form of a state graph.



The accepting state $q_2 \in F$ is marked with a double border. Such automaton

accepts strings like bb,bbb,bab,aab but rejects ϵ,a,ba,aa and so on.

A Mealy machine [5] 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 $\delta : Q \times \Sigma \rightarrow Q \times \Gamma$. For example the below machine



produces output 1000 for input $aabb$. Some articles do not include the set F in definition of Mealy machines and instead assume that all states are accepting. Automata with all states final are called prefix-closed. Here we do not assume Mealy machines to be prefix-closed.

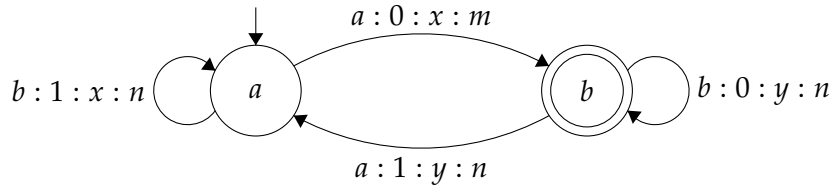
The ability to produce output along each transition allows for modelling reactive systems. Any program or phenomenon that reads input from its environment and “reacts” to it by producing output in one form or another is a reactive system. It is frequently used in the field of formal methods. Many complex state-based systems can be modelled and simplified using Mealy machines [6]. As an example consider a black-box computer program whose logs can be observed [7]. 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 [1]. It is known as the prefix-preserving property [6]. 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 “zero bugs” should become “0 bugs”. The prefix preserving property would be too limiting because it often happens that the suffix of string has a decisive impact on the

translation. The phrase “zeroed bit” also starts with the prefix “zero” but it should not be translated as “0ed bit”!

We intended to find the right balance between expressive power of non-deterministic machines and strong formal guarantees of Mealy automata. To achieve this, we initially decided to use multitape automata [8]. For example the automaton below has 4 tapes



First tape is designated as the input state reading strings over the alphabet $\{a, b\}$. The remaining three tapes are the output tapes, printing strings in the respective alphabets $\{0, 1\}$, $\{x, y\}$, $\{m, n\}$. In theory there could be any (finite) number of input and output tapes.

Our 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.

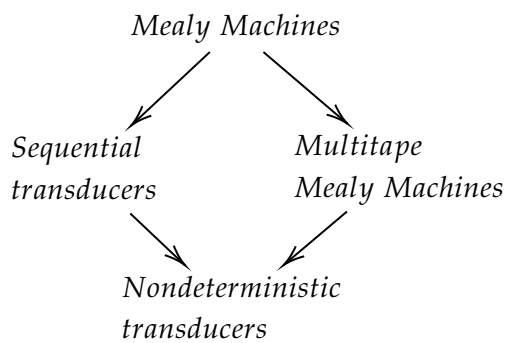
Over the course of research and development we have discovered that the power of multitape Mealy machines was still too limiting for our purposes. In particular we could define congruence classes on pairs of strings similar to those in Myhill-Nerode theorem [2]. Then it can be easily noticed that as long as the number of output tapes is finite, the number of congruence classes must be finite as well. A very simple and intuitive counter-example would be the language that for every string a, aa, aaa, \dots respectively prints output c, bc, bbc, \dots and so on. It could not be expressed using any 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 $\delta : Q \times \Sigma \rightarrow Q^*$.

The nondeterministic transducers [9] [10] [11] 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

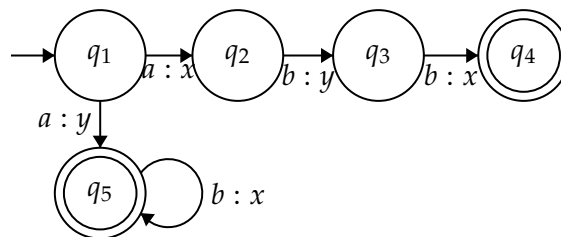
stronger model would be by introducing context-free grammars and pushdown automata. The reason why we were hesitant to use this approach was because of the possible ambiguity of output [12]. Nondeterministic transducers may contain epsilon transitions, which could lead to infinite number of outputs [1] 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 hierarchy of automata presented so far could be illustrated as follows.

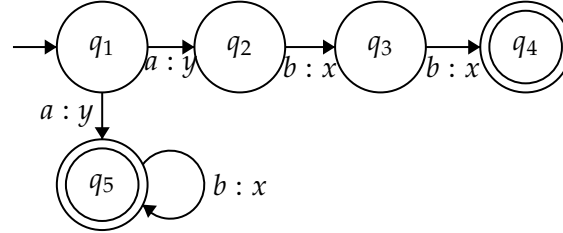


Mealy machines are the simplest and most basic model. Both sequential transducers and multitape automata extend the expressive power of Mealy machines in different ways but neither is more powerful than the other. Finally, nondeterministic transducers are on top of the hierarchy. This is a simplified view. The full family of different models capable of producing output is much larger. The transduction hierarchy is more complex than Chomsky hierarchy, even within the layer of regular transductions (not to mention context-free and context-sensitive transductions).

The model of automata that was finally implemented in Solomonoff, is the functional nondeterministic transducer [8]. Functionality of a 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 paths 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 (with the exception of epsilon transitions coming from initial state). Using advance-and-delay [12] 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 [13] 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 [1].

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 our initial idea too but later we stumbled upon a better approach. In Solomonoff we don't have separate implementations for those operations.

Instead everything is integrated in the form of one monolithic procedure that implements Glushkov's construction. For example, in order to compile the expression

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

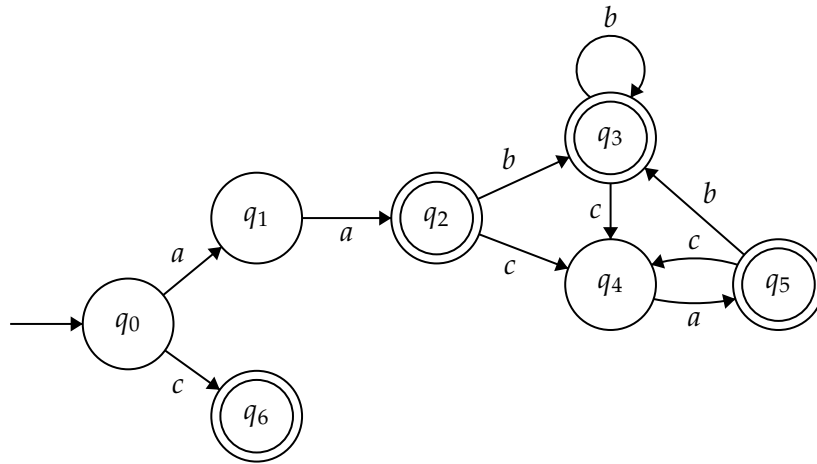
we first convert every symbol into a state and obtain the following intermediate "regular expression"

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

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

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

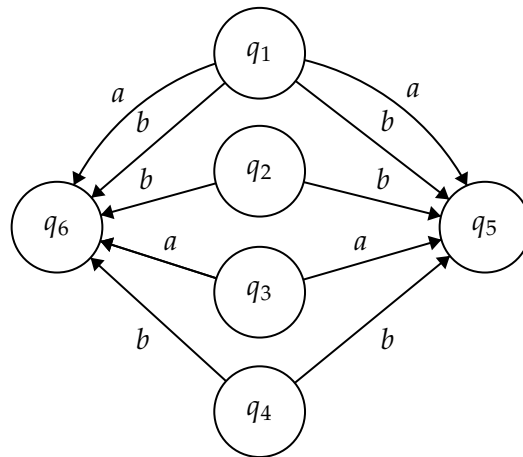
Next we analyse it and determine, which state can be reached from any other. After "reading" q_1 we can "read" q_2 . After q_2 we can either read q_3 or q_4 . After q_3 we can read either q_3 again or go to q_4 . Analogically for the remaining states. We also check, which states can appear at the end 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.



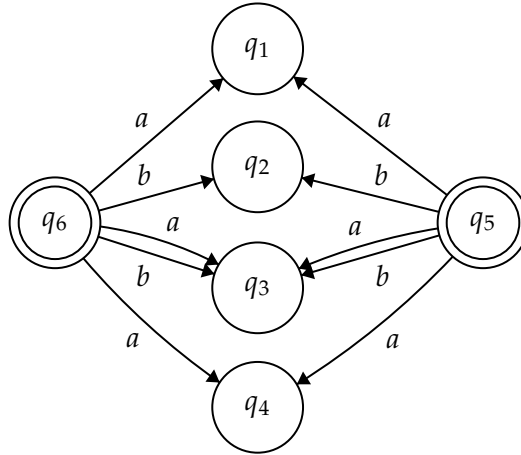
This way the produced automata are very small even without the need for minimisation. If the regular expression consists of n input symbols, then the resulting transducers has $n + 1$ states. Because there is one-to-one correspondence between regular expression symbols and automata states, we are able 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 [14]. 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 [15] 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

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

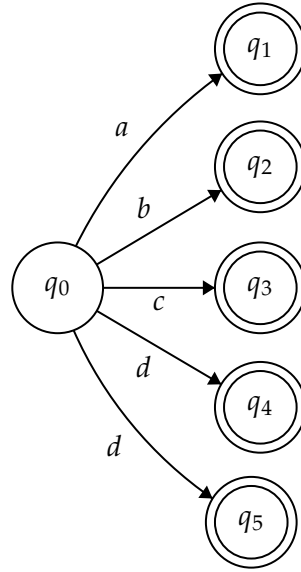
could be shortened to

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

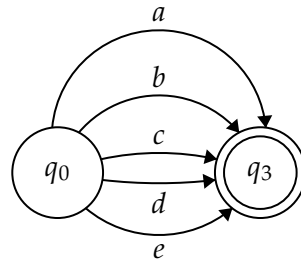
by isolating the common prefix a and suffix b . Extracting common prefixes exactly correspond to merging states with identical incoming transition, while common suffixes correspond to states with the same outgoing transitions. Sometimes there are cases that do not have any common prefix/suffix in the 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 possible to embed any property of the transition directly inside the regular expression. For example, suppose that we want to assign some colours to all transitions. Let B stand for blue and R for red. Then we can embed these colours in an expression like below

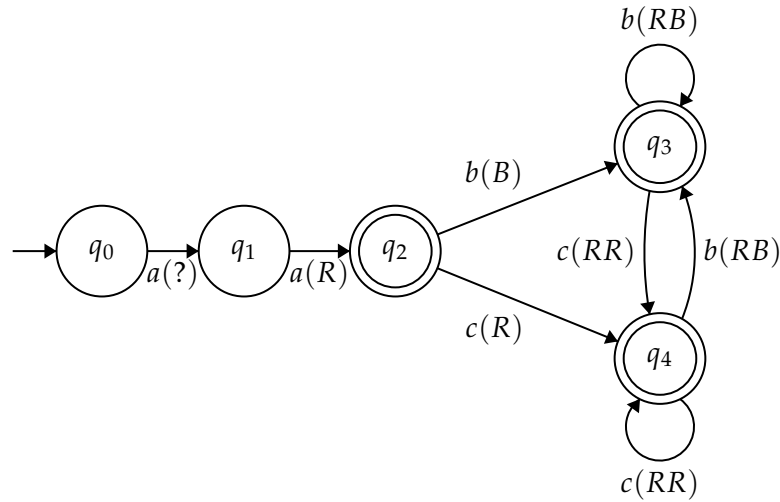
$$aRa(Bb + Rc)R^*$$

which becomes

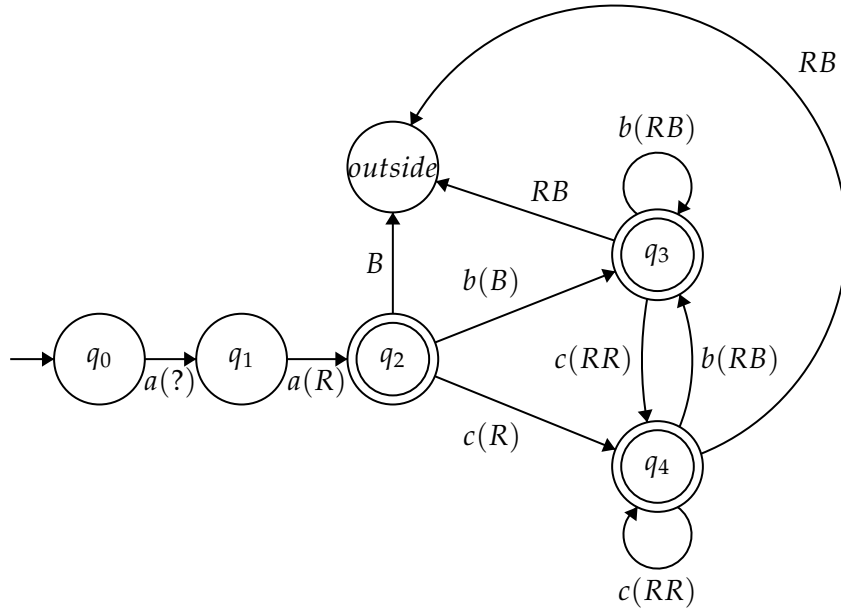
$$q_1Rq_2(Bq_3 + Rq_4)R^*B$$

We know that after q_1 we can read q_2 and along the way we have to cross the color R , because it stands 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



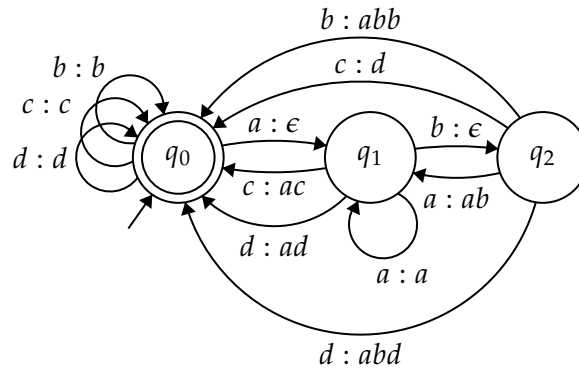
Because colours form a monoid, we can use neutral element as a default value for all unspecified edges like $a(?)$. Moreover, it is also possible to attach meta-information to final states. We could imagine that an accepting state is nothing more than a state with a special outgoing transition that goes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



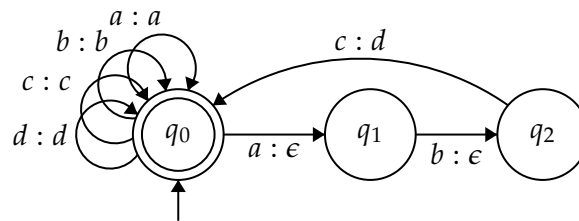
In Solomonoff, the meta-information of interest are transition outputs. We could imagine that strings composed of R and B symbols are the output of a transducer. At this point the reader should appreciate how beautifully and naturally subsequential transducers (automata with output produced at accepting 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 embedded 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 operation 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 [1]. 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

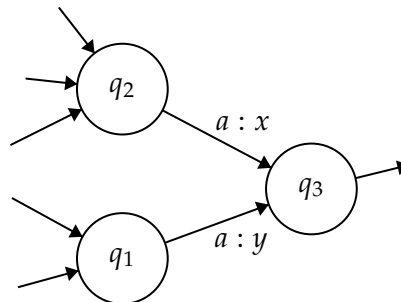


A much simpler alternative would be the nondeterministic transducer

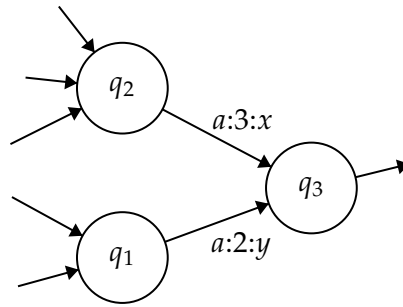


The problem with such a solution is that for input strings like *aabcb* both outputs *aabcb* and *adb* are generated. If only there was a way to assign priority to some outputs in order to disambiguate them, then context dependent rewrites could be expressed by much simpler automata. This is precisely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

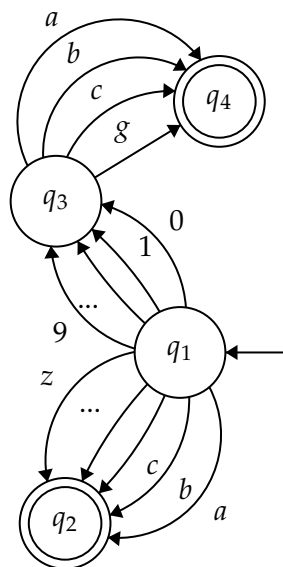


If the next input symbol is a , then the transducer will reach state q_3 and generate two outputs — one ending in x and the other in y . If the automaton later accepts it will produce at least two ambiguous outputs. Lexicographic weights allow us to assign priority to different transitions. In the following 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.

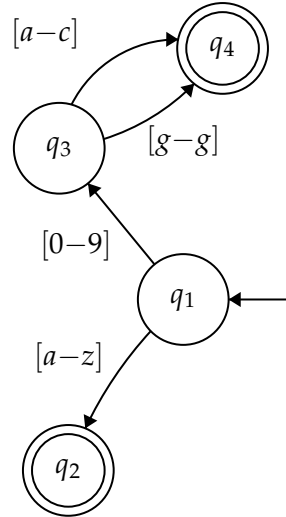


Lexicographic weights allow to make the transducers even more compact. We proved that there exist weighted automata exponentially smaller than even the smallest unweighted nondeterministic ones. In the presence of lexicographic weights, context-dependent rewrites become expressible directly in Glushkov's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

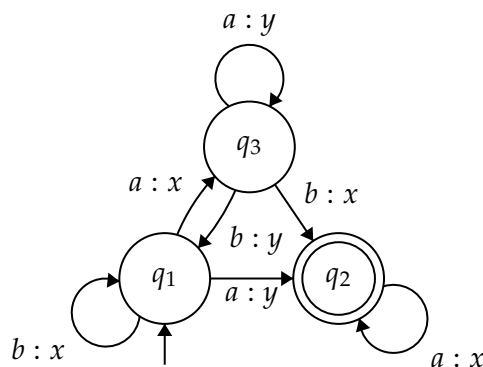
In the tasks of natural language processing it's common to work with large alphabets. User input might contain unexpected sequences like math symbols, foreign words or even emojis and other UNICODE entities. The regular expressions should handle such cases gracefully, especially when using wildcards 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 [16]. While classic automata have edges labelled with individual symbols, our transducers have edges that span entire ranges. A range is easy to encode in computer 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 the 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 the 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.

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

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 this the table can be backtracked from the accepting state backwards. For the backtracking step to be possible, we need to store information about the 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 $\delta : Q \times \Sigma \times Q \rightarrow \Gamma^*$), it becomes possible to use such an 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 an adjacency matrix [17]. 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.

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

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

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

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

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 [13] strings (that is, all substrings of length 2). The early prototype versions of Solomonoff would represent sets as bitsets, where each bit specifies whether an 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 singly linked graphs backed by arrays. This provides the highest efficiency with the smallest memory footprint but it is 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 a 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 ambigu-

ous 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 [1]. While the performance difference between the two is negligible, the advantage of our approach comes from better error messages in case of nondeterminism. If some lexicographic weights are in conflict with each other, the compiler can point the user to the exact line and column of text where the conflict arises, whereas advance-and-delay might miss the origin of the 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.

The theory of automata is primarily founded on the theory of semigroups and monoids. A set X together with operation $\oplus : X \times X \rightarrow X$ forms a semigroup if it satisfies associativity $(x_1 \oplus x_2) \oplus x_3 = x_1 \oplus (x_2 \oplus x_3)$. If the semigroup contains a neutral element 0_X such that $0_X \oplus x = x \oplus 0_X = 0_X$ then it forms a monoid. For example the set of positive (zero excluded) integers together with addition operation forms a semigroup, while a set of non-negative integers (zero included) forms a monoid. Because of associativity, the placement of brackets does not matter and can be omitted altogether. As a result instead of writing

$$((x_1 \oplus (x_2 \oplus x_3)) \oplus x_4) \oplus (x_5 \oplus x_6)$$

we can just write

$$x_1 x_2 x_3 x_4 x_5 x_6$$

Every element of X can be presented in such a way. String of elements of X connected together with the operation \oplus is called a word. There might be more than one word denoting the same element. For example, the number 6 could be framed as $0 + 1 + 2 + 3$ or $0 + 6 + 0$ or just 6. All of those are different words denoting the same number. A free semigroup is one in which no two distinct words denote the same element. The canonical example of free semigroup is the set of all non-empty strings under concatenation. No two strings are equal, unless their notations are syntactically the same. Definition of free monoid is similar to free semigroup, with the exception that neutral elements do not change denotation of a word. For example the following words must all be equal $0 + 3 + 4 = 3 + 0 + 4 = 3 + 4 + 0 + 0$ but all of

$4 + 3 \neq 3 + 4 \neq 1 + 1 + 1 + 4 \neq 1 + 5 + 1$ must be different from each other. The standard example of free monoid is the set of all strings with concatenation operation. No two strings are equal, except for the ones that are concatenated with the empty string ϵ .

Given two monoids X and Y with operations \oplus and \odot respectively, it's possible to build a new one by performing their direct product $X \times Y$. Their joint operation is defined as $(x_1, y_1) \cdot (x_2, y_2) = (x_1 \oplus x_2, y_1 \odot y_2)$.

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 $\bar{\chi}$ be the closure of χ under countable union and complementation (so it forms a sigma algebra). For instance, imagine that there is a 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 $\bar{\chi}$ should be closed under countable union. Therefore, χ is the set of allowed transition labels and $\bar{\chi}$ is the set of all possible "semantic" transitions. We could say that $\bar{\chi}$ is discrete if it contains every subset of Σ . An example of discrete $\bar{\chi}$ would be a finite set Σ with all UNIX-style ranges $[\sigma - \sigma']$ included in χ .

Transducers with input Σ^* and output Γ^* can be seen as a finite state automaton 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}^Σ 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 the underlying set of atomic characters \mathcal{A}^Σ 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

$V^{\Sigma:D} : RE^{\Sigma:D} \rightarrow \Sigma^* \times D$ to be the valuation function:

$$V^{\Sigma:D}(\phi + \psi) = V^{\Sigma:D}(\phi) \cup V^{\Sigma:D}(\psi)$$

$$V^{\Sigma:D}(\phi \cdot \psi) = V^{\Sigma:D}(\phi) \cdot V^{\Sigma:D}(\psi)$$

$$V^{\Sigma:D}(\phi^*) = (\epsilon, \epsilon) + V^{\Sigma:D}(\phi) + V^{\Sigma:D}(\phi)^2 + \dots$$

$$V^{\Sigma:D}(\phi : d) = V^{\Sigma:D}(\phi) \cdot (\epsilon, d)$$

$$V^{\Sigma:D}(a) = a \text{ where } a \in \mathcal{A}^{\Sigma:D}$$

Some notable properties are:

$$x : y_0 + x : y_1 = x : (y_0 + y_1)$$

$$x : \epsilon + x : y + x : y^2 \dots = x : y^*$$

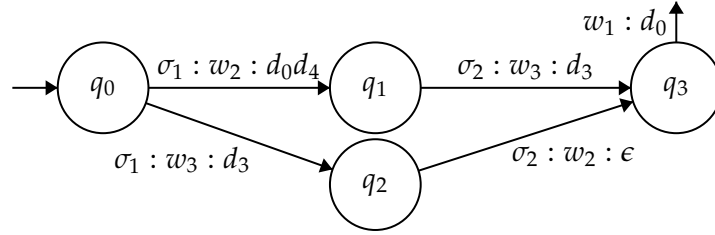
$$(x : y_0)(\epsilon : y_1) = x : (y_0 y_1)$$

$$x_0 : (y_0 y') + x_1 : (y_1 y') = (x_0 : y_0 + x_1 : y_1) \cdot (\epsilon : y')$$

$$x_0 : (y' y_0) + x_1 : (y' y_1) = (\epsilon : y') \cdot (x_0 : y_0 + x_1 : y_1)$$

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 $\phi \in RE_W^{\Sigma \rightarrow D}$ and $w_0, w_1 \in W$ then $w_0 \phi$ and ϕw_1 are in $RE_W^{\Sigma \rightarrow D}$. 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 $\psi \in RE_W^{\Omega \rightarrow D}$ along with mapping $\alpha : \Omega \rightarrow \mathcal{A}^\Sigma$. This mapping will remember the original atomic character, before it was rewritten to a unique symbol in Ω . For example

$$\phi = (\epsilon : x_0)x_0(x_0 : x_1 x_3)x_3 w_0 + (x_1 x_2)^* w_1$$

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 the function $\Lambda : RE_W^{\Omega \rightarrow D} \rightharpoonup (D \times W)$. It returns the output produced for empty word ϵ (if any) and weight associated with it. (We use the symbol \rightharpoonup to highlight the fact that Λ is a partial function) For instance in the previous example the 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

$\Lambda(\epsilon) = (\epsilon, 0)$

$\Lambda(\omega) = \emptyset$ where $\omega \in \Omega$

Next step is to define $B : RE_W^{\Omega \rightarrow D} \rightarrow (\Omega \rightharpoonup D \times W)$ which for a given formula ψ returns set of Ω characters that can be found as the first in any string of $V^{\Omega \rightarrow D}(\psi)$ 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 $\cdot : (\Omega \rightharpoonup D \times W) \times (D \times W) \rightarrow (\Omega \rightharpoonup D \times W)$ 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 a 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 $E : RE_W^{\Omega \rightarrow D} \rightarrow (\Omega \rightarrow D \times W)$, 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 $V^{\Omega \rightarrow D}(\psi)$. Given two functions $b, e : \Omega \rightarrow D \times W$ we define product $b \times e : \Omega \times \Omega \rightarrow D \times W$ such that for any $(\omega_0, (y_0, w_0)) \in b$ and $(\omega_1, (y_1, w_1)) \in e$ there is $((\omega_0, \omega_1), (y_0 y_1, w_0 + w_1)) \in b \times e$. Then define $L : RE_W^{\Omega \rightarrow D} \rightarrow (\Omega \times \Omega \rightarrow D \times W)$ 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 a 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

$$\delta : Q \times \Sigma \rightarrow (Q \rightarrow D \times W)$$

$$\tau : Q \rightarrow D \times W$$

$$Q = \{q_\omega : \omega \in \Omega\} \cup \{q_\epsilon\}$$

$$\tau = E(\psi)$$

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

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

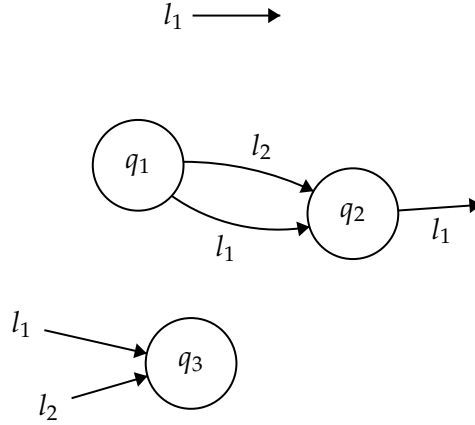
This concludes our extended version of Glushkov's construction. This formal definition does not include the plentiful optimisation techniques that we implemented. In particular, usually this algorithm treats $\Omega \times \Omega$ as 2-factor strings $\Omega\Omega$. It's possible to instead interpret this in a graph-theoretic manner, where

every element of Ω is a state and the product $\Omega \times \Omega$ defines an adjacency matrix. The function L produces an assignment $\Omega \rightarrow D \times W$ which to every edge associates its label. Graph theory has countless tools for optimising graph-based algorithms and data structures. While adjacency matrix works best for dense graphs, different data representations are usually used for sparse graphs. Finite state automata and transducers are the canonical example of sparse graphs, because the number of possible edges is bounded by the size of the alphabet, whereas the number of states is expected to be much larger. This graph-theoretic approach can be made more formally rigorous.

The formal definition of a graph is a tuple (Q, δ) where Q is the set of vertices and $\delta \subset Q \times Q$ is the set of edges. Most books by convention use letters V and E to refer to the set of vertices and edges respectively. Here the naming Q and δ has been used purposely to highlight the connection between graphs and automata. The formal definition of a directed graph (digraph for short) is the same as the previous one, with the additional assumption that δ is a set of ordered pairs, while undirected graphs assume the pairs to be unordered.

We define a labelled graph as (Q, δ) where the set of edges has the form $\delta \subset Q \times \mathbb{L} \times Q$ for some set of labels \mathbb{L} . Every multitape automaton can be viewed as a labelled graph. For example finite state automaton with single input tape is usually defined in terms of $\delta : Q \times \Sigma \rightarrow Q$, which could be rewritten as $\delta \subset Q \times \Sigma \times Q$. Mealy automata could alternatively be defined using $\delta \subset Q \times \Sigma \times \Gamma \times Q$. Sequential transducers are of the form $\delta \subset Q \times \Sigma \times \Gamma^* \times Q$. Nonsequential transducers could further be extended as $\delta \subset Q \times \Sigma^* \times \Gamma^* \times Q$. Interestingly, there is no difference between input tapes and output tapes. A multitape automaton with one input Σ and two outputs Γ, Δ can be formalized as $\delta \subset Q \times \Sigma \times \Gamma \times \Delta \times Q$ but the exact same formalization would be achieved if all Σ, Γ and Δ were input tapes. Hence the concept of “input” and “output” is an algorithmic distinction that appears in implementation of automaton but from the algebraic and graph-theoretic point of view the distinction is artificial.

We define partial graph as a labelled graph allowing partial edges. A full edge $(q_1, l, q_2) \in \delta$ has source vertex q_1 , label l and target vertex q_2 . A partial edge might be lacking source (\emptyset, l, q_2) , target (q_1, l, \emptyset) or both $(\emptyset, l, \emptyset)$ but it must have a label. An edge that neither has any target nor source is called an epsilon edge. Those with no source are called incoming and those with no target are outgoing. Hence partial graphs could be formalized as (Q, δ) with edges in the form of $\delta \subset (Q \cup \{\emptyset\}) \times \mathbb{L} \times (Q \cup \{\emptyset\})$. Below is an example of a partial graph state diagram.



The partial graphs have a lot in common with Glushkov's construction. Let $D \times W$ be the set of labels. Then Λ function returns a single label $\Lambda(\psi) = (y, w)$, which looks very much like an instance of the epsilon edge $(\emptyset, y, w, \emptyset)$. The B function is responsible for collecting initial atomic symbols $\omega \in \Omega$ and their labels $(y, w) \in D \times W$, which resembles partial edges with no source $(\emptyset, y, w, \omega)$. Similarly for $(\omega, y, w) \in E(\psi) \subset \Omega \rightarrow D \times W$, which we could interpret as partial edge with no target $(\omega, y, w, \emptyset)$. The set of 2-factors $L(\psi) \subset (\Omega \times \Omega \rightarrow D \times W)$ could be interpreted as a set of full labelled edges $\Omega \times D \times W \times \Omega$. As a result, instead of converting Λ, B, E and L to a subsequential transducer, we can directly obtain definition of a partial graph with no "post processing" required. This could be better seen if we combine all of the Λ, B, E and L functions into one large procedure G that returns the entire partial graph $G : RE_W^{\Omega \rightarrow D} \rightarrow (\Omega, \delta)$. We have several cases to consider.

Let's recall how each function behaves on the atomic symbol ω . We can see that G should return a partial graph with no full edges

$$L(\epsilon) = \emptyset$$

one outgoing edge

$$E(\omega) = \{(\omega, (\epsilon, 0))\}$$

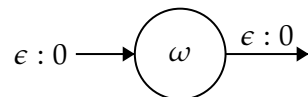
one incoming edge

$$B(\omega) = \{(\omega, (\epsilon, 0))\}$$

and no epsilon edge

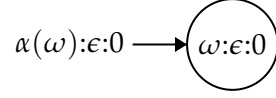
$$\Lambda(\omega) = \emptyset$$

Hence it could be presented as the following very simple single-state graph:



In order to use such a graph as an automaton it's enough to treat all the outgoing edges state's subsequential output and the input label of every full or incoming

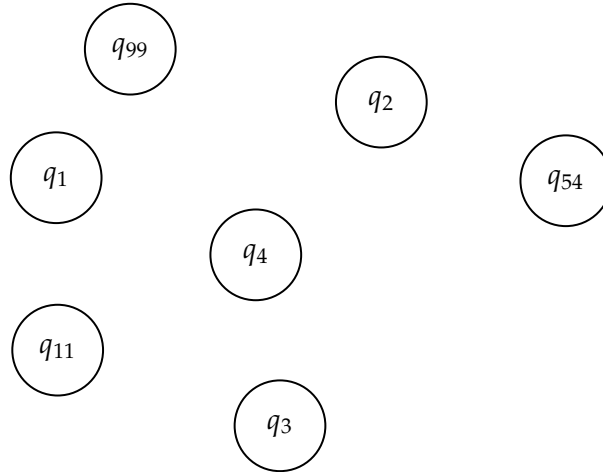
edge is directly determined by the target state. Hence this graph becomes the following automaton:



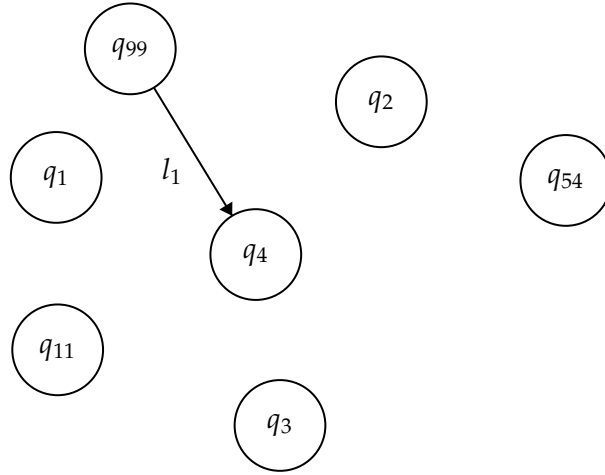
It is worth pointing out that there will always be at most only one outgoing edge per state and at most one global epsilon edge. This directly follows from the definition of E as function $E(\psi) : \Omega \rightarrow D \times W$ rather than relation $E(\psi) : \Omega \times D \times W$ and because Λ returns only a single element.

The set Ω is directly treated as the set of states Q . An input string $\sigma_1\sigma_2\ldots\sigma_n \in \Sigma^*$ is accepted by the partial graph if there exists a path $(\emptyset, y_0, w_0, \omega_1) \rightarrow (\omega_1, y_1, w_1, \omega_2) \rightarrow \ldots \rightarrow (\omega_n, y_n, w_n, \emptyset)$ starting in incoming edge $(\emptyset, y_0, w_0, \omega_1)$, ending in outgoing edge $(\omega_n, y_n, w_n, \emptyset)$ and such that for every i the input symbol σ_i equals to the symbol of target state $\alpha(\omega_i)$. The produced output is determined by $y_0y_1\ldots y_n$.

It's possible to omit the use of α mapping entirely during the implementation. There is no need to linearise ψ or build Ω . Instead we could use a special data structure of singly-linked graphs. Every vertex itself is an array holding its own outgoing (full) edges. Such representation of graphs has certain benefits. Unlike in most other graph data structures (adjacency matrices and lists) the singly-linked graphs don't have a well defined set of vertices. There could be millions of vertices allocated in computer memory and none of them connected to any other.



Each of those individual vertices could be thought of as a separate graph in and of itself but the distinction is blurry. As soon as we add some connection



the two graphs suddenly merge into one (or perhaps, one is a subgraph of the other, since it's possible to reach q_4 from q_{99} but not the other way around). We might think of one “infinite supergraph” containing all vertices that have ever been created or could potentially be created in the future. Hence it makes more sense to speak of connected components rather than the “supergraph” itself. Given some set of incoming partial transitions $B \subset \{\emptyset\} \times \mathbb{L} \times Q$, the connected component induced by B is the set \bar{B} defined as subset of Q containing all the reachable vertices, starting from some initial edge in B .

Every time we encounter some symbol σ , we allocate a new vertex q and initialize it as an empty array $q = []$. Next we create a set of incoming edges B containing only a single edge $B = \{(\emptyset, \epsilon, 0, q_1)\}$. Similarly we also need to keep track of outgoing edges $E = \{(q_1, \epsilon, 0, \emptyset)\}$. The $G(\sigma)$ function will return a partial graph $(\bar{B}, B \cup E)$ but for technical details (presented in upcoming paragraphs) the sets B and E should be kept in their own separate lists. There is no need to store the full edges in their own list, since they are stored in each corresponding vertex. The set of vertices \bar{B} is implied by B and also doesn't need to be stored in computer memory explicitly. Hence $(\bar{B}, B \cup E)$ is merely the formal presentation of a graph but in the actual implementation a data structure like (B, E) is used instead.

We have a full formal and algorithmic description of $G(\sigma)$. The next case to consider is the union $G(\psi_0 + \psi_1)$. We need to combine the full edges from both subgraphs

$$L(\psi_0 + \psi_1) = L(\psi_0) \cup L(\psi_1)$$

as well as the respective outgoing

$$E(\psi_0 + \psi_1) = E(\psi_0) \cup E(\psi_1)$$

and incoming edges

$$B(\psi_0 + \psi_1) = B(\psi_0) \cup B(\psi_1)$$

The epsilon edges cannot be combined in any way hence union might result in errors when both subgraphs simultaneously contain their own epsilon edges

$$\Lambda(\psi_0 + \psi_1) = \Lambda(\psi_0) \cup \Lambda(\psi_1)$$

We can combine all of those operations into a single operation G on partial graphs. We first perform $G(\psi_0)$ and then $G(\psi_1)$. The result of $G(\psi_0 + \psi_1)$ does not require to perform any additional operations. The graph $G(\psi_0 + \psi_1)$ is merely one "supergraph" consisting of two disconnected components $G(\psi_0)$ and $G(\psi_1)$. Algorithmic implementation can be achieved by concatenating the list of incoming edges of the first subgraph with the other (and analogically for outgoing edges). Because we use singly-linked graphs as the backing data structure, the set of reachable vertices is automatically implied and does not require to be updated. As a result union operation is $O(1)$ with respect to size of the graph and its memory footprint is kept low.

The next case to consider is the concatenation $G(\psi_0\psi_1)$. First we need to compute $G(\psi_0)$ to obtain the sets B_{ψ_0} , E_{ψ_0} , L_{ψ_0} and Λ_{ψ_0} of its incoming, outgoing, full and epsilon edges respectively. Analogically for $G(\psi_1)$. The L function performs product

$$\Lambda(\psi_0\psi_1) = \Lambda(\psi_0) \cdot \Lambda(\psi_1)$$

This operation can be redefined in terms of partial edges. If the set of labels \mathbb{L} is a monoid, then multiplication of edges becomes possible. Outgoing edge (q_1, l_1, \emptyset) multiplied together with incoming edge (\emptyset, l_2, q_2) will give us a full edge (q_1, l_1l_2, q_2) . Similarly, multiplication of epsilon edge $(\emptyset, l_3, \emptyset)$ with either outgoing edge (q_1, l_1, \emptyset) or incoming edge (\emptyset, l_2, q_2) yields (q_1, l_3l_1, \emptyset) or (\emptyset, l_3l_2, q_2) respectively. If we extend those operations to work on entire sets of edges

$$X \cdot Y = \{x \cdot y : x \in X \text{ and } y \in Y\}$$

then we can introduce multiplication (concatenation) of partial graphs.

$G(\psi_0) \cdot G(\psi_1) = (\overline{B_{\psi_0}}, L_{\psi_0} \cup L_{\psi_1} \cup E_{\psi_0} \cdot B_{\psi_1} \cup \Lambda_{\psi_0} \cdot B_{\psi_1} \cup E_{\psi_0} \cdot \Lambda_{\psi_1} \cup \Lambda_{\psi_0} \cdot \Lambda_{\psi_1})$ In the context of singly-linked graphs, the operation $E_{\psi_0} \cdot B_{\psi_1}$ for every $(q_0, y_0, w_0, \emptyset) \in E_{\psi_0}$ and $(\emptyset, y_1, w_1, q_1) \in B_{\psi_1}$ computes a full edge $(q_0, y_0y_1, w_0w_1, q_2)$ and inserts it to the list q_0 . Analogically partial edges are computed and inserted to the respective lists as well.

The case of Kleene closure $G(\psi_0^*)$ is similar to concatenation. When working with singly linked graphs, Kleene closure behaves like concatenation of graph $G(\psi_0) \cdot G(\psi_0)$ with itself, except that epsilon edges require special handling.

The remaining cases of $G(\psi : d)$, $G(\psi w)$ and $G(w\psi)$ are achieved by multiplying outgoing edges with $(d, 0)$, (ϵ, w) with incoming edges and outgoing edges with (ϵ, w) respectively. It should be noted that labels \mathbb{L} can act on partial graphs. Left action $l \cdot G$ multiplies $l \cdot b$ with all incoming edges of G and the

epsilon edge $l \cdot \lambda$. Similarly right action $G \cdot l$ multiplies $e \cdot l$ with all outgoing edges of G and the epsilon edge $\lambda \cdot l$.

We conclude this construction with a few notes about performance and possible extensions. As it can be noticed, by eliminating the need for computing linearised alphabet Ω , the algorithm became fully parallelizable. Any subexpression of the original regular expression can be compiled independently to the rest. Singly-linked graphs guarantee that no reallocations of states are necessary. The weights and outputs only act on partial edges, hence once a full edge is computed it is never mutated. As a result, the algorithm never has the need to revisit already compiled parts of the automaton. In such sense the construction is fully linear and all individual operations are of $O(1)$ complexity. There are exactly as many states as there are input symbols, hence memory consumption is also linear. The produced automata are epsilon-free, except for the only one global partial epsilon edge. The compiled singly-linked graph forms an automaton in and of itself, hence no conversion from 2-factors to automata is necessary, like it was in the case of “standard” Glushkov’s construction. Moreover, it’s straightforward to extend the compilation with custom “external” functions

$$G(F(\psi)) = F_{\text{custom_implementation}}(G(\psi))$$

This allows for adding non-standard operations like subtraction, composition, inversion, powerset and many others.

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 a set of finite sequences Ω_X of elements from a 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 $(n, f) : \mathbb{N} \times (0 \dots n \rightarrow X)$. Then in Java we could represent it as class

```

1 |         class Ω<X>{
2 |             int n();
3 |             X f(int i); //requires 0<=i<n
4 |         }
```

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

```

1 |         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 they 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 lead 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`

```

1 |         concat(a,b1);
```

```
2 | 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

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

The 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 a linear function without mutations. For instance

```
1 | static <X> void concat( $\Omega$ <X> left,  $\Omega$ <X> right)
```

is the mutating version of

```
1 | /**consumes left*/
2 | static <X>  $\Omega$ <X> concat( $\Omega$ <X> left,  $\Omega$ <X> right)
```

In object-oriented languages we have one more possible approach

```
1 | void concat( $\Omega$ <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 the 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

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

we obtain more concise notation

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

```
3 | default Y f2(X1 x1, X2 x2, X3 x3){...}  
4 | }
```

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.

CHAPTER 3

Build system

MARCIN JABŁOŃSKI

Build systems sometimes referred to as build automations, are widely understood solutions that automate processes related to software building, the most important of which are in a narrower sense compiling, packing, testing, or in a broader sense also processes included in continuous integrations and continuous deployment. The rest of the chapter will focus mainly on the first meaning of the concept.

3.1 ADVANTAGES OF USING BUILD SYSTEMS

Based on the research we conducted while designing this system, we concluded that the main reasons for using this type of tools are:

- time saving and resistance to mistakes

The most obvious advantage of automation is time savings. The complexity of the build processes could be not only excessively time-consuming while it is manually operated but also leads to numerous mistakes.

- consistency and portability

Depending on the technology stack used in the project, the result may differ depending on the specific environment. Build systems can mitigate this risk by standardizing some elements of the environment such as versions of compilers, runtime systems, dependencies, and also compilation parameters or environment variables.

- reproducible builds

This feature results from the previous point. Build reproducibility can be important to maintaining the credibility of open source projects and in

some cases, it may help detect compromised build chains which bring us to the next feature.

- security

Build systems can contribute to increased security by the fact that it facilitates the separation of the environment in which the application is developed and in which it is built.

- dependency resolving

Depending on the specific system, it can help to varying degrees in managing dependencies from local resources as well as remote ones.

- better audibility

The more descriptive nature of the configuration makes it easier to identify certain characteristics, mostly related to versions of environment components or dependencies.

- tasks parallelization

Build systems can perform multiple tasks in parallel as long as the order of dependencies is followed.

3.2 OVERVIEW OF SELECTED BUILD SYSTEMS

This section of the chapter covers some of the building automation tools. Due to the enormity of the possibilities of each of them, the analysis is focused on presenting the features that are particularly important for a given one, and at identifying the differences between them.

3.2.1 GNU Make

The first system described in this chapter is Make, actually its implementation from the GNU project. Make, as a utility included in the POSIX standard[18], is a fairly common build system. Its implementations differ in terms of specific functionalities but remain largely compatible. For simplicity, Make is hereinafter referred to as GNU Make.

Make is language-agnostic and is not limited to building only, but can be easily used to automate any related side tasks. It is facilitated by the simple

syntax of the Makefile file, which defines the rules in which the declared targets should be executed. Makefiles can be human-written or automatically generated by high-level build systems such as CMake, GNU Autotools, or Meson. Make comes with an extensive set of tools that can simplify the manual creation of makefiles.

Rules are the basics of makefiles. They have the following syntax.

```
target: dependencies
      system command(s)
```

The example below shows the contents of a simple makefile. The program built by Make in this case will be called `edit`, like the first target. The first target is the default one and will run if you run `make` with no additional parameters. A specific target can be called using `make <target>`.

Listing 3-1. Example of simple MakeFile [19]

```
1 | objects = main.o command.o display.o utils.o
2 |
3 | edit : $(objects)
4 |       cc -o edit $(objects)
5 | main.o : main.c defs.h
6 |       cc -c main.c
7 | command.o : command.c defs.h command.h
8 |       cc -c command.c
9 | display.o : display.c defs.h buffer.h
10 |      cc -c display.c
11 | utils.o : utils.c defs.h
12 |      cc -c utils.c
13 | clean :
14 |      rm edit $(objects)
```

The target can be a file like the listed object files and executable, or an action like the `clean` target. If the target is a file, by default `make` will check whether the source file has changed since the target file was created and decide on that basis if it should be rebuilt. Changing dependencies will also trigger a rebuild for the proper units.

Make's functionality includes:

- echoing
- variables (sometimes referred as macros)
- build in functions like `prefix`, `foreach`, `eval`, `shell`

- conditional expressions

and many more to make makefiles more convenient when read and written by users. These features make it a very versatile tool.

3.2.2 Ninja

Ninja is a relatively new software. It was created to speed up the building of very large projects consisting of many thousands of files. Work on this project started in 2010, when its creator Evan Martin was working on the Google Chrome browser. The browser code back then consisted of around 40,000 source files [20], which explains the desire to speed up the application building process.

The syntax of the `.ninja` files that define the build process is trivial and is limited almost only to rules that allow you to define short names for complex commands, variables, and expressions that work just like in Make. If the target on the left is older than the source on the left, the target will be rebuilt. The example 3-2 defines the CC rule with the command property. In this case, this rule specifies how to execute builds with GCC. It is then used in the build target.

Listing 3-2. Example of simple `.ninja` file [21]

```
1 | cflags = -Wall
2 |
3 | rule cc
4 |     command = gcc $cflags -c $in -o $out
5 |
6 | build foo.o: cc foo.c
```

There are several aspects related to the simplicity of this solution. It allowed creating a hand-written and very efficient lexer and parser which sped up significantly the parsing process. Removal of redundant functionalities not only helped to improve performance because Ninja does not have to call additional subroutines to handle these but even check whether they have been used as if only the simplest Make expression was used. It also makes the `.ninja` file for a large project inconvenient to write manually. Therefore, it is assumed that Ninja should be used in conjunction with another higher-level build system such as CMake.

The performance of the Ninja is demonstrated by the benchmark result by David Röthlisberger. The following fragments come from the article about it [22].

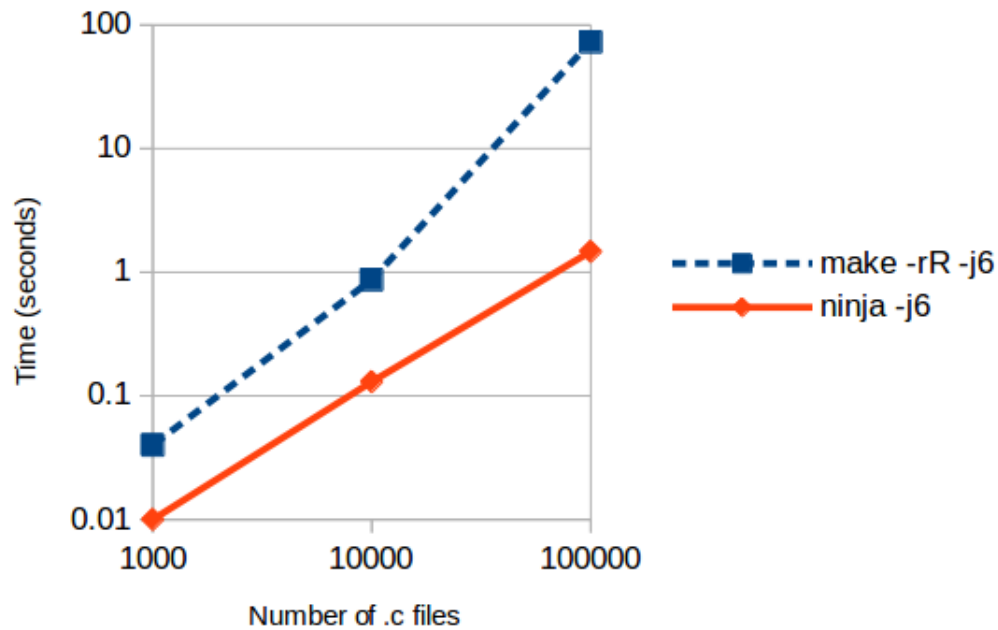


Figure 3.1. Benchmark results for no-op build [22]

For a project with 1,000 programs built from 10,000 C files and 10,000 header files, there is no significant difference in the duration of a fresh build. A no-op build takes less than a second with make (0.87s versus 0.13s with Ninja), probably not enough to really matter for interactive use.

For a much larger project (10,000 programs, 100,000 C files and 100,000 header files) there is a significant difference in a no-op build: 73s for Make versus 1.5s for Ninja. Make spends 98% of that time processing the 100,000 compiler-generated “.d” files that are used for tracking implicit dependencies on header files.

3.2.3 Maven

Maven is significantly different from previous systems. It is intended for managing Java-based projects. Projects are configured by pom.xml file, which, unlike the previous configuration files, is more declarative. In addition to meta information, it contains a list of dependencies, repositories from which the external packages can be downloaded, and a list of plugins with a configuration. The entire system is based on a plugin mechanism that can be attached from external

sources. Additional configuration or new functionality, e.g. how to build a package or how to deploy an application, can be added and configured using plugins. Below is an example pom.xml file. The example defines the name, version, unique identifier of the project, the name of the compiled file, dependencies, in this case — JUnit, version of the runtime environment, and plugins — maven-compiler-plugin which handles the compilation process.

Listing 3-3. Example of simple pom.xml file [23]

```
1 <project xmlns="http://maven.apache.org/POM/4.0.0"
2   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
3     xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
4       http://maven.apache.org/xsd/maven-4.0.0.xsd">
5   <modelVersion>4.0.0</modelVersion>
6
7   <groupId>com.mycompany.app</groupId>
8   <artifactId>my-app</artifactId>
9   <version>1.0-SNAPSHOT</version>
10
11   <name>my-app</name>
12   <url>http://www.example.com</url>
13
14   <properties>
15     <project.build.sourceEncoding>UTF-8</project.build.
16       sourceEncoding>
17     <maven.compiler.source>1.7</maven.compiler.source>
18     <maven.compiler.target>1.7</maven.compiler.target>
19   </properties>
20
21   <dependencies>
22     <dependency>
23       <groupId>junit</groupId>
24       <artifactId>junit</artifactId>
25       <version>4.11</version>
26       <scope>test</scope>
27     </dependency>
28   </dependencies>
29
30   <build>
31     <plugins>
32       <plugin>
33         <groupId>org.apache.maven.plugins</groupId>
34         <artifactId>maven-compiler-plugin</artifactId>
35         <version>3.3</version>
```

```

35         <configuration>
36             <source>1.5</source>
37             <target>1.5</target>
38         </configuration>
39     </plugin>
40 </plugins>
41 </build>
42 </project>

```

Maven uses its predefined project structure. It can be extended depending on the configuration. Below is an example structure.

```

my-app
|-- pom.xml
'-- src
    |-- main
    |   |-- java
    |   |   '-- com
    |   |       '-- mycompany
    |   |           '-- app
    |   |               '-- App.java
    |   '-- resources
    |       '-- META-INF
    |           '-- application.properties
    '-- test
        '-- java
            '-- com
                '-- mycompany
                    '-- app
                        '-- AppTest.java

```

The most important features of Maven, which distinguish it from previous systems, are that it is significantly easier to configure a complex process of building, if a project is well-defined, builds are reproducible. Also, Maven includes a built-in package manager, so any environment setup can be recreated

without additional solutions.

3.3 SOLOMONOFF

3.3.1 System description

Our build system does not resemble any of the previously mentioned but is closer to Maven in some of its concepts. As in the case of Maven, the build configuration is declarative. In this case, we are also dealing with a specialized build system, which is not suitable for other applications. Additionally, Solomonoff has a very simplified package manager.

3.3.2 Problem

The task of Solomonoff Build System was not only to create an automation system but also to extend the capabilities of the compiler itself, so that it could be practical.

The main problems it has to solve was:

- compiler related
 - resolve dependencies
 - parallel build process
- automation related
 - keep the intermediate transducers and builds only what is needed
 - import project configuration from file
 - download defined packages from remote repository
 - export project as package

Due to the architecture of the library, which provides compiler functionality, and its close nature to some of the problems the build system had to address, the only reasonable option seemed to be keeping both compiler and build system

within a single process.

3.3.3 Solutions

Simple approach

The easiest way would be not to build the system at all, but only to wrap the library with a simple file-loading interface and implement for some kind of “include” macro. Files could then be concatenated and parsed to extract the function definitions. Then compile them one by one. This approach would force users to take care of the proper order of function definitions in the source code. Also, in the case of our compiler, it would not be possible to compile in parallel. The tasks described in this model would be handled by a compiler in more traditional approaches, but due to the specifics of our compiler architecture, the aforementioned drawbacks and the potential associated with such a solution, made us decide to adopt the following architect.

System architecture

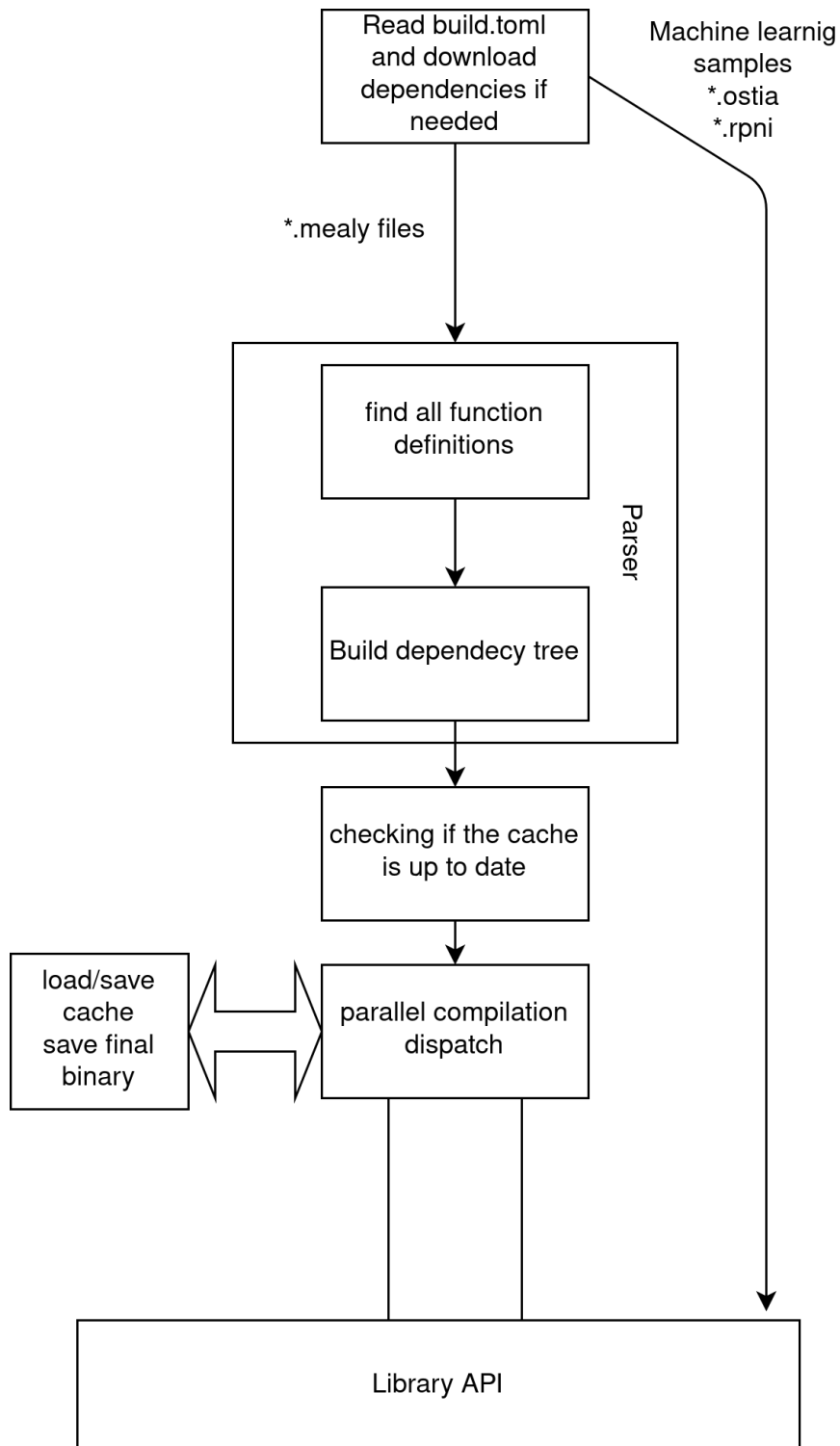
The following diagram 3.2 shows the build process.

Parsing sources

Before starting the compilation, it is necessary to find a list of all function definitions at the beginning, which will then allow the cache to be loaded and the dependency graph to be mapped to make the process convenient and efficient. To do this it is necessary to pre-parse all source files. For this purpose, we use a parser generated with ANTLR and more specifically ANTLR4. ANTLR is LL parser generator, which means that input is processed from left to right with leftmost derivation. The grammar is identical to that used in the library, however, the parser itself is very simplified. It works comes down to writing down the names of functions, their types, and their bodies.

Dependency resolving

The problem is that function definitions can be located in different files on any position and functions can be nested (one function can be used in the body of another one). Before starting the compilation, it is worth checking immediately whether it has a chance to succeed at all. To accomplish this, Solomonoff first loads code from repositories and user-supplied source code, and then parses it. During the process, a directed acyclic graph is generated. The

**Figure 3.2.** Build process

graph represents relations of being a dependency of another node that represent functions. The system uses JGraphT to This sorting algorithm guarantees that if we start compiling functions in order in which they were returned by the sorting function, the necessary dependencies for each subsequent element will already be compiled.

The topological sort is based on the Kahn's algorithm.

In the constructor fragment cited below, a list of source nodes is created.

For each vertex in a graph, we check how many incoming edges it has. If zero then it is a source vertex.

Listing 3-4. Implementation of the algorithm from the JGraphT library pt. 1 [24]

```

1  ...
2  this.inDegreeMap = new HashMap<>();
3  for (V v : graph.vertexSet()) {
4      int d = 0;
5      for (E e : graph.incomingEdgesOf(v)) {
6          V u = Graphs.getOppositeVertex(graph, e, v);
7          if (v.equals(u)) {
8              throw new IllegalArgumentException(GRAPH_IS_NOT_A_DAG);
9          }
10         d++;
11     }
12     inDegreeMap.put(v, new ModifiableInteger(d));
13     if (d == 0) {
14         queue.offer(v);
15     }
16 }
17
18 this.remainingVertices = graph.vertexSet().size();
19 ...

```

The advance method 3-5 called, for each iteration, will initially return the existing sources vertex and then the new ones which appear after removing the previous ones, and so on to the last vertex.

We find all adjacent vertices for each vertex taken from the queue. Then we decrease the counter of adjacent vertices for each of them. If a given vertex's counter equals zero, it is added to the queue of the source vertexes.

Listing 3-5. Implementation of the algorithm from the JGraphT library pt. 2 [24]

```

1  private V advance()
2  {

```

```

3      V result = queue.poll();
4
5      if (result != null) {
6          for (E e : graph.outgoingEdgesOf(result)) {
7              V other = Graphs.getOppositeVertex(graph, e, result);
8
9              ModifiableInteger inDegree = inDegreeMap.get(other);
10             if (inDegree.value > 0) {
11                 inDegree.value--;
12
13                 if (inDegree.value == 0) {
14                     queue.offer(other);
15                 }
16             }
17         }
18
19         --remainingVertices;
20     } else {
21         /*
22          * Still expecting some vertices, but no vertex has zero
23          * degree.
24          */
25         if (remainingVertices > 0) {
26             throw new IllegalArgumentException(GRAPH_IS_NOT_A_DAG);
27         }
28
29         return result;
30     }

```

Parallel building

The next step is a compilation. To achieve compilation efficiency, the functions are put in the mentioned order in the queue from which they will be then downloaded to the Thread pool, if the compilation of their dependencies have already finished. In this way, we are to use the full potential of the CPU that the operating system will make available to us.

The code below shows how the parallel compilation problem is solved. We take out successive sorted nodes of the dependency graph. For each subsequent node, we assign threads from the pool to the Future which run compilations of a given subgraph or load its copies from the cache.

```
1 | ...
```

```

2 // already sorted dependencies
3 while (dependencyOrder.hasNext()) {
4
5     final String id = dependencyOrder.next();
6     final VarDef<G> varDef = definitions.get(id);
7
8     if (varDef == null) {
9         assert compiler.specs.borrowVariable(id) != null : id;
10
11     } else if (varDef instanceof VarDefAST) {
12         VarDefAST<G> var = (VarDefAST<G>) varDef;
13         assert var.def != null : id;
14
15         compiled.put(id, pool.submit(() -> {
16             if (var.needsRecompilation) {
17                 final G compiledGraph =
18                     var.def.compile(compiler.specs, i -> {
19                         try {
20                             final Future<G> f = compiled.get(i);
21                             if (f == null) {
22                                 final LexUnicodeSpecification.Var<N, G> v =
23                                     compiler.specs.copyVariable(i);
24                                 assert v != null;
25                                 final G graph = compiler.specs.getGraph(v);
26                                 return graph;
27                             } else {
28                                 final G graph = compiler.specs.deepClone(f.get());
29                                 return graph;
30                             }
31                         } catch (InterruptedException
32                             | ExecutionException e) {
33                             throw new RuntimeException(e);
34                         }
35                     });
36
37                 System.err.println("Compiled " + id);
38
39                 if (buildBin) {
40                     try (FileOutputStream f =
41                         new FileOutputStream(var.cacheFilePath.toFile())) {
42                         compiler.specs.compressBinary(compiledGraph,
43                             new DataOutputStream(f));
44                     } catch (IOException e) {
45                         e.printStackTrace();

```

```

46         var.cacheFilePath.toFile().deleteOnExit();
47     }
48 }
49 return compiledGraph;
50 } else {
51     System.err.println("Loaded from cache " + id);
52     try (DataInputStream dis =
53         new DataInputStream(new FileInputStream(
54             var.cacheFilePath.toFile())) {
55         return compiler.specs.decompressBinary(Pos.NONE, dis);
56     }
57 }
58 }));
59 }
60 }
61 ...

```

Caching transducers

To achieve this goal, an analogous mechanism was used to that used in GNU Make. When parsing the code, we retrieve information about the modification date of a given source from the file system. If the user did not order otherwise before the compilation starts, the modification dates of previously compiled transducers saved on the disk will be read and then both will be compared. If the binary file date is not older than the source code date and the same situation repeats for all dependencies of the given function, it will not be compiled again at this point.

The code below shows how to check if the compiled transducer is up-to-date with the source code.

```

1  protected VarDef(String id, String sourceFile, Config config)
2  throws IOException {
3      ...
4      if (sourceFile == null) {
5          this.needsRecompilation = false;
6          return;
7      }
8      final Path sourceFilePath = Paths.get(sourceFile);
9      final FileTime sourceFileModificationTime =
10         (FileTime) Files.getAttribute(sourceFilePath, "
11             lastModifiedTime");
12     final boolean needsRecompilation;
13     if (!config.caching_read && config.caching_write) {

```

```

13     needsRecompilation = true;
14 } else if (Files.exists(cacheFilePath)) {
15     final FileTime cacheModificationTime =
16         (FileTime) Files.getAttribute(cacheFilePath, "
            lastModifiedTime");
17     final int diff =
18         cacheModificationTime.compareTo(sourceFileModificationTime)
19         ;
19     needsRecompilation = diff < 0;
20 } else {
21     needsRecompilation = true;
22 }
23 this.needsRecompilation = needsRecompilation;
24 }

```

Here we go through the dependency tree to check if any of the dependencies out of date will force a cascading recompilation. For each function under test, we find all incoming edges for the corresponding vertex in the dependency graph and then check to see if any of their dependencies need recompilation.

```

1  ...
2  //Resolve which dependencies need to be recompiled
3  final Stack<VarDef<G>> toCheckIfNeedsRecompilation = new Stack
4      <>();
5  for (VarDef<G> def : definitions.values()) {
6      if (!def.needsRecompilation) {
7          toCheckIfNeedsRecompilation.add(def);
8      }
9  }
10 while (!toCheckIfNeedsRecompilation.isEmpty()) {
11     final VarDef<G> definition = toCheckIfNeedsRecompilation.pop()
12     ;
13     assert !definition.needsRecompilation;
14     boolean actuallyShouldBeRecompiled = false;
15     for (Object dependencyEdge : dependencyOf.incomingEdgesOf(
16         definition.id)) {
17         final String dependencyOfVertex = dependencyOf.getEdgeSource
18             (dependencyEdge);
19         final VarDef<G> definitionOfDependency = definitions.get(
20             dependencyOfVertex);
21         if (definitionOfDependency != null
22             && definitionOfDependency.needsRecompilation) {
23             actuallyShouldBeRecompiled = true;
24             break;
25         }
26     }
27 }

```

```

21     }
22     if (actuallyShouldBeRecompiled) {
23         definition.needsRecompilation = true;
24         for (Object dependedEdge : dependencyOf.outgoingEdgesOf(
25             definition.id)) {
26             final String idToReconsider = dependencyOf.getEdgeTarget(
27                 dependedEdge);
28             final VarDef<G> definitionToReconsider = definitions.get(
29                 idToReconsider);
30             if (!toCheckIfNeedsRecompilation.contains(
31                 definitionToReconsider)) {
32                 toCheckIfNeedsRecompilation.add(definitionToReconsider);
33             }
34         }
35     }
36 }
37 ...

```

Setting up form a file

Build.toml file which delivers a configuration to the build system describes the project metadata, contains the program's configurations such as repository paths, and defines the sources from which the system loads the code for compilation. An example file is shown below.

```

1  project_name = "test" # project name (needed during export)
2  version = 1.0 # project version (needed during export)
3  private_key = "./private.der"
4  # private key (needed during export until sign_pkg equals false)
5  # RSA with SHA256 'der' format
6  sign_pkg = true
7
8  cache_location = "bin/" # bin is default
9  #local_repo = "" # custom location (default is $USER_HOME/.
10     Solomonoff)
11
12  [[source]]
13  path = "sample3.mealy"
14
15  [[source]]
16  path = "sample4.ostia"
17
18  [[pkg]]

```

```
18 | public_key = "pubkey.der" # publickey RSA with SHA256 'der'  
    |         format  
19 | name = "test"  
20 | version = "1.0"  
21 | remote_repo = "https://solomonoff.projektstudencki.pl/repo/"  
22 | verify_signature = true
```

The system uses TOML markup language as it gives us the greatest flexibility.

Package manager

Solomonoff comes with a very simplified package manager. To download a package, the build system, using HTTP or HTTPS, queries the remote repository for the requested package and downloads it to the local repository if it is available. Also, the signature is downloaded if the user has not requested otherwise. If the user tries to download the same package again in the same version, it will not be downloaded since it is already in the local repository. For this reason, the naming convention is very important. Our suggestion is to keep project names as <developer_email@example.com> <desired_project_name> in order to avoid a collision. Packages and signatures have names in the following format <project_name> - <version> .zip [.sig]. This very simple system allows you to use any web server as a remote repository without the need to install special software.

The package manager can also export a package. The package is a pair of zip archive and its signature. The archive only contains the code, because compilation does not take so long that it makes sense to distribute already compiled automata.

CHAPTER 4

Web technologies

BOGDAN BONDAR

4.1 EDITOR AND REPL CONSOLE

Creation of a 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. The 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 then be printed back to the user. For example in python's REPL, typing $2 + 2$ yields 4.

```
1 | >>> 2 + 2
2 | 4
```

In a more complicated case like running a regex, user obtains an object containing all matched groups

```
1 | >>> re.compile('a|b*').match('xxaxxbxxbbbx')
2 | <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

```
1 | 'a' ('b' | 'c' | 'ef' ) 'd'
```

would return a language consisting of strings

```
1 | 'abd', 'acd', 'aefd'
```

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


```
1 | 'a'*
```

would be evaluated as infinite set

```
1 | '', 'a', 'aa', 'aaa', ...
```

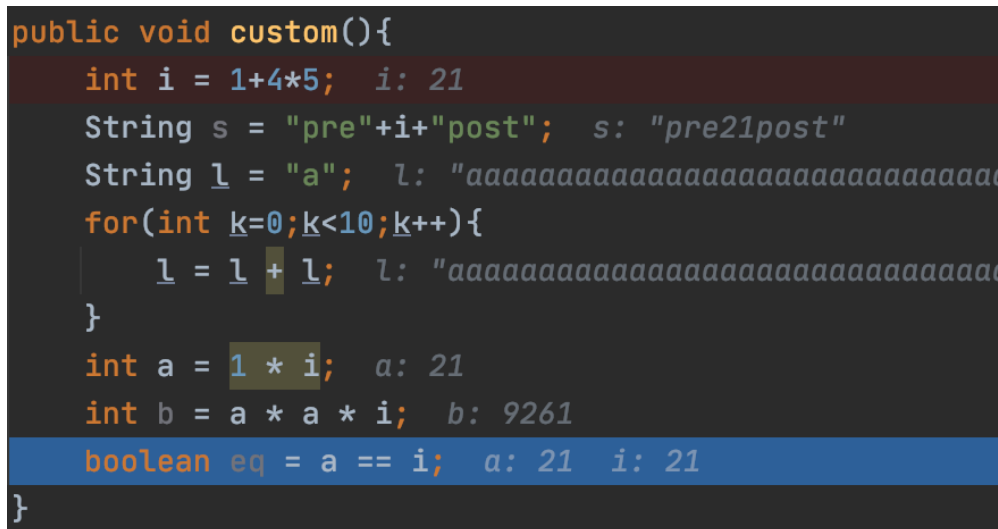
Some regexes, might be finite but of exponential size. For instance

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

yields a set of all bit-strings of length 4. Presenting the user with the result in the 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 the 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. Such an approach has been chosen by many debuggers, including the one presented in figure 4.1. For regular expressions such a user interface is not as spectacular.



```
public void custom(){
    int i = 1+4*5;    i: 21
    String s = "pre"+i+"post";    s: "pre21post"
    String l = "a";    l: "aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
    for(int k=0;k<10;k++){
        l = l + l;    l: "aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
    }
    int a = 1 * i;    a: 21
    int b = a * a * i;    b: 9261
    boolean eq = a == i;    a: 21    i: 21
}
```

Figure 4.1. Graphical interface of debugger used by IntelliJ. The results of Java expressions are displayed in the same line. The contents can change dynamically as the execution progresses.

It's not possible to evaluate a regex down into any particular value. The closest two operations that could be used for presentation purposes are as follows. It's either possible to evaluate a regular expression on a particular input (it can be

achieved with `:eval NAME 'input string'` command in REPL) or generate a sample set of accepted inputs (using `:rand_sample NAME of_size NUMBER`).

One last and perhaps the most engaging way of presenting the results to the user is by visually graphing them. Any automata can be interpreted as a directed graph. This property was used to further enhance user interface (automaton graph will be shown after typing `:vis NAME` or by clicking an appropriate button in the graphical interface)

Those and many other functionalities have been added to the browser-based version of REPL. Its 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

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

and hit the 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

```
1 | '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

```
1 | x = 'a'
2 | y = 'b'
```

could be written in a single line

```
1 | x = 'a' y = 'b'
```

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

```
1 | x = 'a' y
```

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

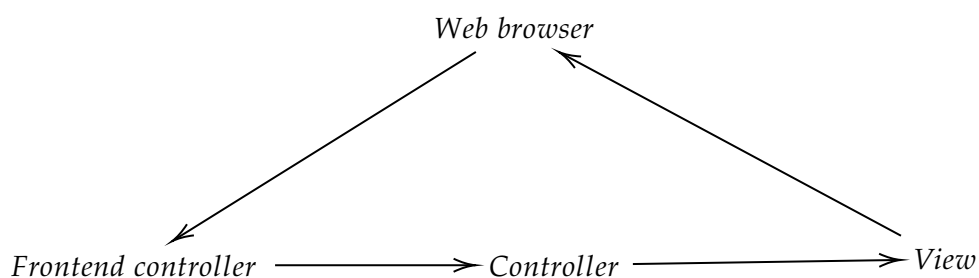
```
1 | = 'b'
```

in the following 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 the 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 a similar convention.

4.2 SPRING BACKEND

A Spring MVC is a Java framework which is used to build web applications. It follows the Model-View-Controller design pattern. It implements all the basic features of a core spring framework like Inversion of Control, Dependency Injection. A Spring MVC provides an elegant solution to use MVC in spring framework by the help of DispatcherServlet. Here, DispatcherServlet is a class that receives the incoming request and maps it to the right resource such as controllers, models, and views.



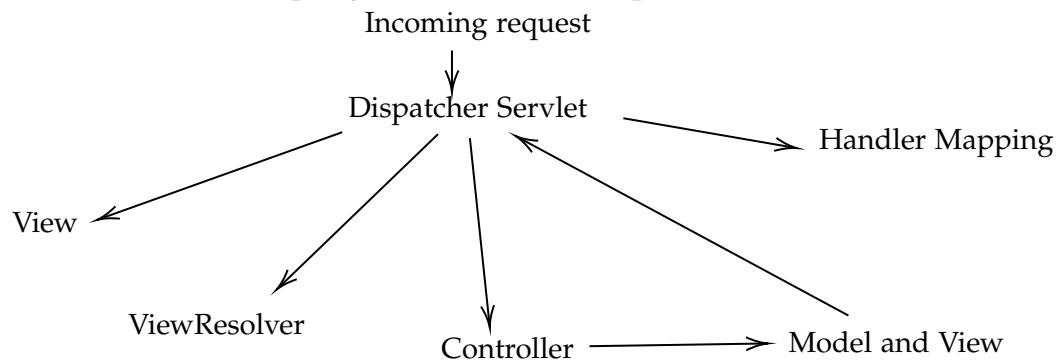
Model contains the data of the application. A data can be a single object or a collection of objects.

Controller contains the business logic of an application. Here, the @Controller annotation is used to mark the class as the controller.

View represents the provided information in a particular format. Generally, JSP+JSTL is used to create a view page. Although Spring also supports other view technologies such as Apache Velocity, Thymeleaf and FreeMarker.

Frontend controller is implemented in Spring Web MVC in the form of the DispatcherServlet class. It is responsible to manage the flow of the Spring MVC

application. The flow of Spring Web MVC could be presented as follows



As displayed in the figure, all the incoming requests are intercepted by the DispatcherServlet that works as the front controller. The DispatcherServlet gets an entry of handler mapping from the XML file and forwards the request to the controller. The controller returns an object of ModelAndView. The DispatcherServlet checks the entry of the view resolver in the XML file and invokes the specified view component.

Using Spring MVC comes with numerous advantages. It allows for separation of roles, where the model object, controller, command object, view resolver, DispatcherServlet, validator, etc. can be fulfilled by a specialized object. It uses a light-weight servlet container to develop and deploy your application. It provides a robust configuration for both framework and application classes that includes easy referencing across contexts, such as from web controllers to business objects and validators. The Spring MVC also facilitates fast and parallel development. It promotes reusable business code by using the existing business objects instead of creating new ones. Test of JavaBean classes enables injection of test data using the setter methods, which allows for easier testing. Flexible Mappings provide the specific annotations that easily redirect the page. Overall Spring is a powerful tool used by numerous companies and developers around the world.

In this project, Spring has been used to facilitate communication between frontend interface and compiler instances. The REPL is implemented on the server-side as a REST API endpoint.

```

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

Every user has their own instance of REPL

```

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

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

```

1 | public static class Repl {
2 |     private static class CmdMeta<Result> {
3 |         final ReplCommand<Result> cmd;
4 |         final String help;
5 |         final String template;
6 |
7 |         private CmdMeta(ReplCommand<Result> cmd,
8 |             String help, String template) {
9 |             this.cmd = cmd;
10 |            this.help = help;
11 |            this.template = template;
12 |        }
13 |    }
14 |
15 |    HashMap<String, Repl.CmdMeta<String>> commands;
16 |    OptimisedHashLexTransducer compiler;
17 | }

```

whenever user types some command on the REPL console

```
1 | :cmd arg1 arg2 arg3
```

it gets parsed as

```

1 | String firstWord = "cmd";
2 | String remaining = "arg1 arg2 arg3";

```

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

```

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

```

The rest controller contains implementations of many such commands

```

1 | public static final ReplCommand<String> REPL_LIST = ...
2 | public static final ReplCommand<String> REPL_EVAL = ...
3 | public static final ReplCommand<String> REPL_RUN = ...
4 | public static final ReplCommand<String> REPL_EXPORT = ..
5 | public static final ReplCommand<String> REPL_IS_DETERMINISTIC =
   |     ...
6 | public static final ReplCommand<String> REPL_LIST_PIPES = ...
7 | public static final ReplCommand<String> REPL_EQUAL = ...
8 | public static final ReplCommand<String> REPL_RAND_SAMPLE = ...
9 | public static final ReplCommand<String> REPL_CLEAR = ...

```

```

10 public static final ReplCommand<String> REPL_UNSET = ...
11 public static final ReplCommand<String> REPL_RESET = ...
12 public static final ReplCommand<String> REPL_LOAD = ...
13 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

```

1 public interface ReplCommand<Result> {
2     Result run(
3         HttpSession httpSession,
4         OptimisedHashLexTransducer compiler,
5         Consumer<String> log,
6         Consumer<String> debug,
7         String args) throws Exception;
8 }

```

As an example, consider the command

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

which evaluates transducer `f` for input string `'abc'`. On the frontend side, JavaScript will perform REST query as follows

```

1 const response = await fetch('repl', {
2     method: 'POST',
3     body: ":eval f 'abc'"
4 })

```

which will be received by server

```

1 @PostMapping("/repl")
2 public ReplResponse repl(HttpSession httpSession,
3 @RequestBody String line) {
4     Repl repl = (Repl) httpSession.getAttribute("repl");
5     final String result = repl.run(
6         httpSession,
7         line, // ":eval f 'abc'"
8         s -> out.append(s).append('\n'), // console output
9         s -> { } // debug logs are not displayed
10    );
11    ...
12 }

```

and the `repl.run` method will query the appropriate implementation to call

for the `:eval` command.

```

1 final String firstWord = "eval";
2 final String remaining = "f 'abc'";
3 final Repl.CmdMeta<String> cmd = commands.get(firstWord);
4 return cmd.cmd.run(httpSession, compiler, log, debug, remaining)
    ;

```

This in turn will trigger the following lambda function

```

1 ReplCommand<String> REPL_EVAL =
2 (httpSession, compiler, logs, debug, args) -> {
3     String[] parts = args.split("\\s+", 2); // f 'abc'
4     String name = parts[0]; // f
5     String input = parts[1]; // 'abc'
6     G transducer = compiler.getTransducer(name);
7     String output = compiler.specs.evaluate(transducer, input);
8     return output == null ? "No match!" : output;
9 };

```

The output is sent back to JavaScript in form of JSON

```

1 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 `:load` command. Its purpose is to emulate the process of loading source code from a file. Whenever a user types some code in the editor, it needs to be transported to the 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

```

1 class ReplInput{
2     String command;
3     String editorContent;
4 }
5 public ReplResponse repl(
6     HttpSession httpSession,
7     @RequestBody ReplInput)

```

and query it using

```

1 const response = await fetch('repl', {
2     method: 'POST',
3     body: {
4         command: replCommand,

```

```

5 |         editorContent: editor.getValue()
6 |     }
7 | })

```

The downside of such a solution is that the editor content could become large and sending it would require more internet bandwidth and time. Often the 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 the user has not modified it. Hence the process of uploading code to the server has been delegated to a separate REST call.

```

1 | const response = await fetch('upload_code', {
2 |     method: 'POST',
3 |     body: code
4 | })

```

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

```

1 | @PostMapping("/upload_code")
2 | public void uploadCode(HttpSession httpSession,
3 | @RequestBody String text) {
4 |     httpSession.setAttribute("code", text);
5 | }

```

Aside from the editor and REPL there is one more window on the webpage. It is dedicated to tutorials 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 the 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.

4.3 DESIGN

The frontend technologies used for designing our website are based on Bootstrap [25]. It is an HTML, CSS & JS Library that focuses on simplifying the development of informative web pages (as opposed to web apps). The primary purpose

of adding it to a web project is to apply Bootstrap's choices of color, size, font and layout. As such, the primary factor is whether the developers in charge find those choices to their liking. Once added to a project, Bootstrap provides basic style definitions for all HTML elements. The result is a uniform appearance for prose, tables and form elements across web browsers. In addition, developers can take advantage of CSS classes defined in Bootstrap to further customize the appearance of their contents. For example, Bootstrap has provisioned for light- and dark-colored tables, page headings, more prominent pull quotes, and text with a highlight. Bootstrap also comes with several JavaScript components in the form of jQuery plugins. They provide additional user interface elements such as dialog boxes, tooltips, and carousels. Each Bootstrap component consists of an HTML structure, CSS declarations, and in some cases accompanying JavaScript code. They also extend the functionality of some existing interface elements, including for example an auto-complete function for input fields. The most prominent components of Bootstrap are its layout components, as they affect an entire web page. The basic layout component is called "Container", as every other element in the page is placed in it. Developers can choose between a fixed-width container and a fluid-width container. While the latter always fills the width of the web page, the former uses one of the four predefined fixed widths, depending on the size of the screen showing the page:

- Smaller than 576 pixels
- 576–768 pixels
- 768–992 pixels
- 992–1200 pixels
- Larger than 1200 pixels

Once a container is in place, other Bootstrap layout components implement a CSS Flexbox layout through defining rows and columns. A precompiled version of Bootstrap is available in the form of one CSS file and three JavaScript files that can be readily added to any project. The raw form of Bootstrap, however, enables developers to implement further customization and size optimizations. This raw form is modular, meaning that the developer can remove unneeded components, apply a theme and modify the uncompiled Sass files.

Figure 4.2 presents an example of a navigation bar created with help of Bootstrap's nav classes. Figure 4.3 shows two responsive panes implemented using flex, row and column classes.

Our 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

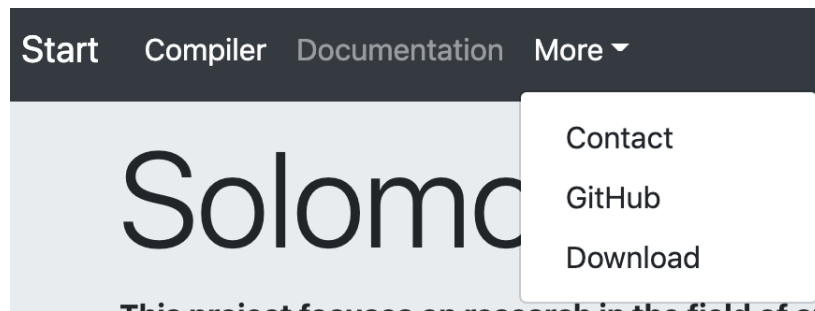


Figure 4.2. Navigation bar using dedicated Bootstrap classes from the nav family

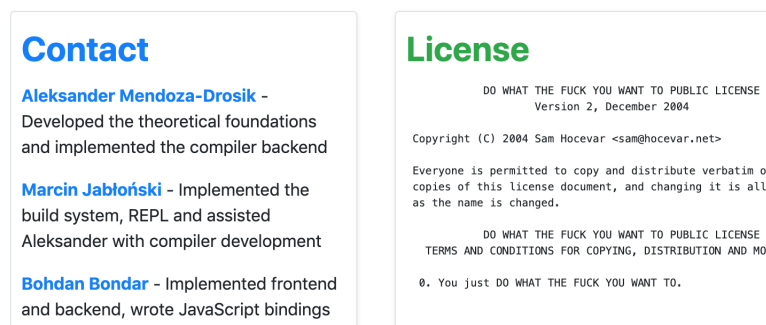


Figure 4.3. The two panes below will change their size and layout depending on dimensions and orientation of the screen.

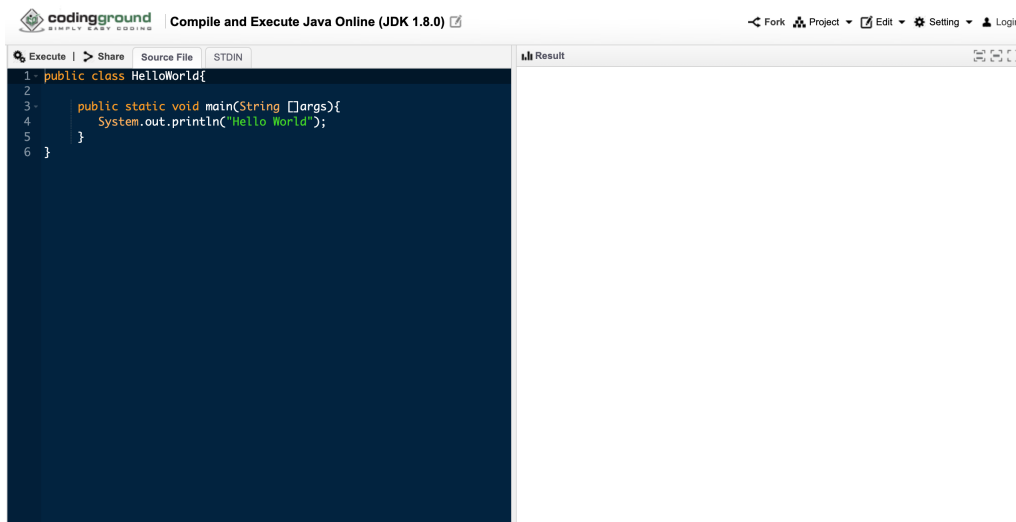


Figure 4.4. The CodingGround website provides an online compiler for several languages. This Java example has a dedicated window that emulates terminal output.

presenting it to the user was not so self-evident. There exist numerous different approaches that are highly dependent on the language. Figure 4.4 shows an example of an online Java compiler consisting only of two windows - one for code and one for compiler output. Java does not have REPL, hence there is no need to implement console input. A slightly different approach has been taken by Haskell mode for emacs presented in figure 4.5. There 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. Figure 4.6 presents the approach that we settled for in the final version of the online playground for Solomonoff. Before reaching such a state we have experimented with another approach that seemed more natural for regular expressions. Figure 4.7 shows an example of a similar website that evaluates UNIX regexes. Initially we tried to mimic such an 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. Figure 4.8 shows how our website looked at this stage. 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

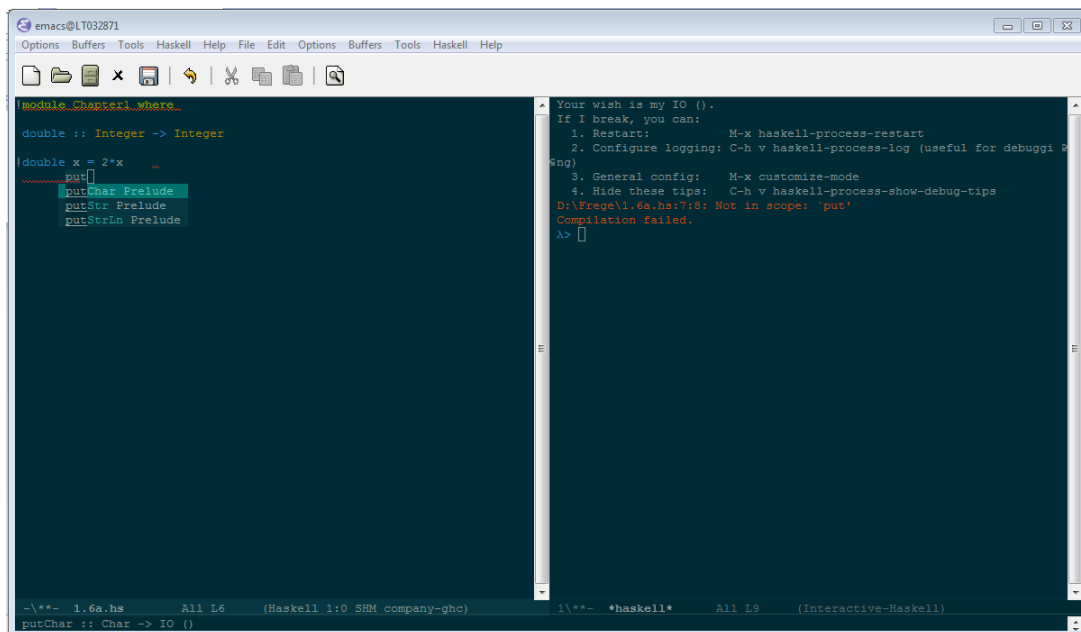


Figure 4.5. A dedicated Emacs mode for Haskell allows for interactive testing and evaluation of arbitrary expression defined in the source file.

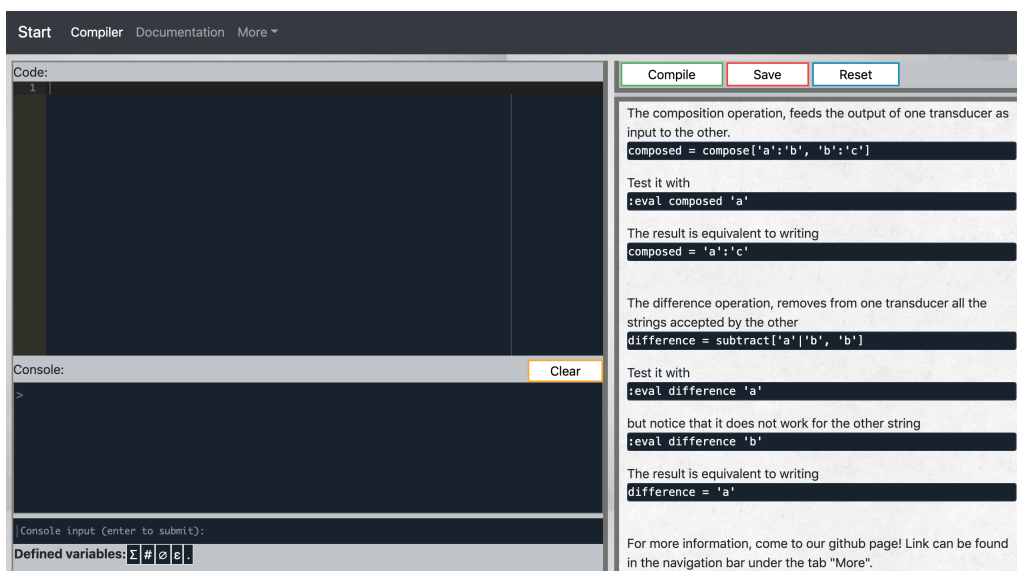


Figure 4.6. The final version of Solomonoff online playground. The layout inspired by several other websites and tools such as Emacs mode for Haskell, Alt-Ergo online demo and other editors

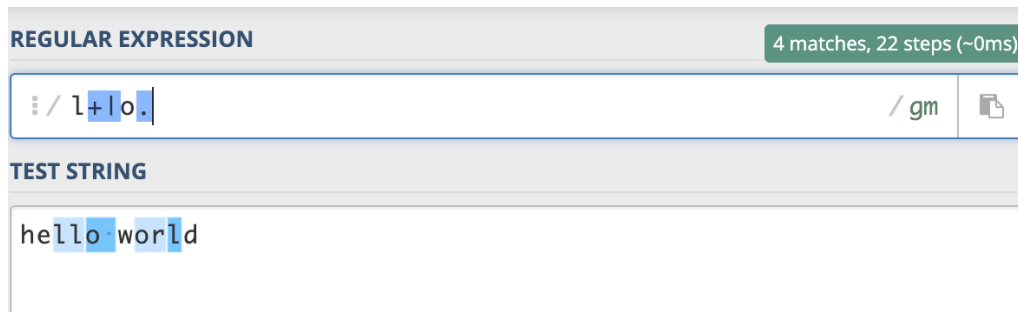


Figure 4.7. On this website a user can closely examine, which parts of regex correspond to a particular substring in the input.

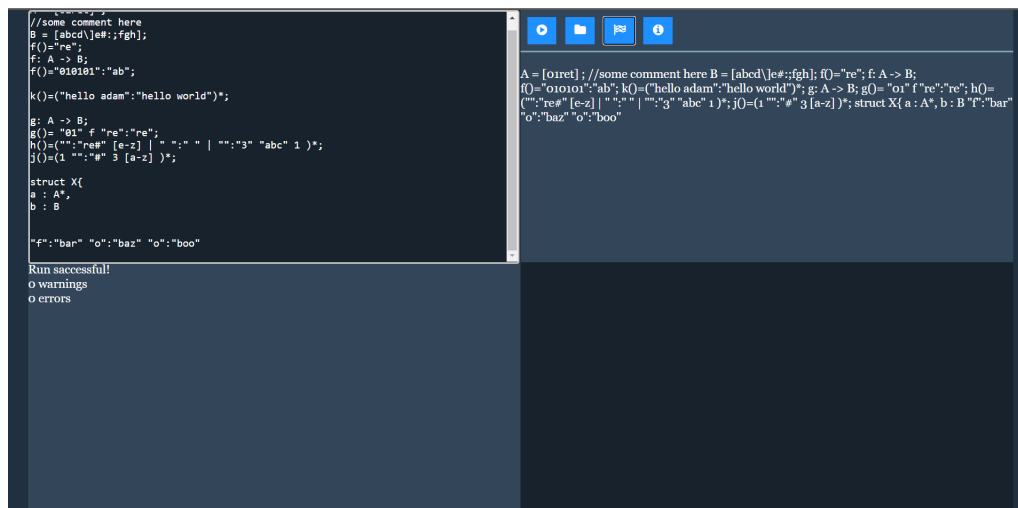


Figure 4.8. This early version of our website was primarily inspired by simple tools for regex investigation and debugging.

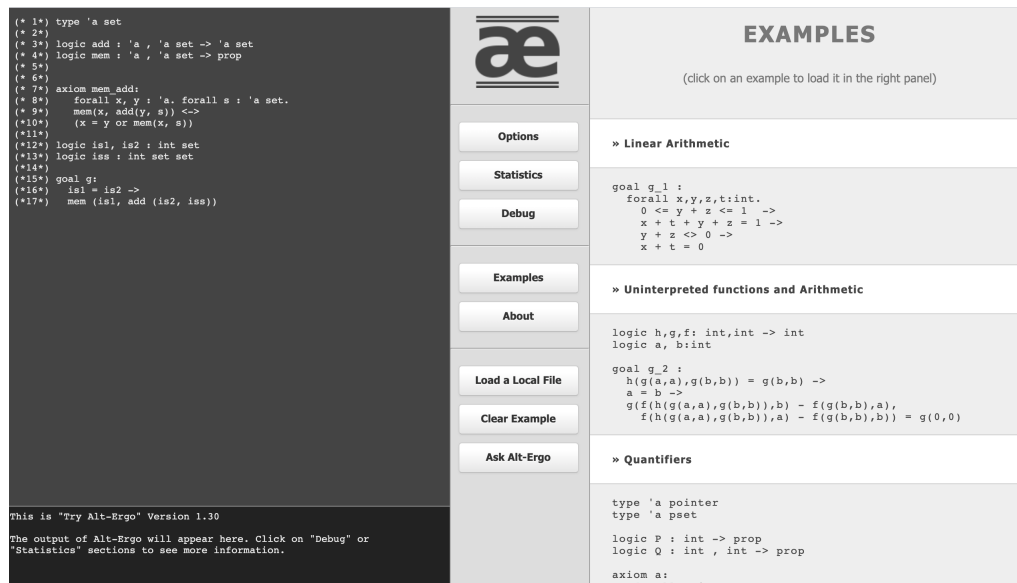


Figure 4.9. The Alt-Ergo is an SMT solver, which is unrelated to regular expressions but it has one thing in common with Solomonoff. They both are niche and complex tools that have a steep learning curve inaccessible to mass audiences. Alt-Ergo solved this issue by providing an online demo with interactive examples. We strived to replicate such an approach in Solomonoff.

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 the compiler. Hence, we decided to scrap the idea with two input-output windows and tried to emulate console-like REPL instead. It was at this point when we first tried to add a window for documentation like in figure 4.6. A great source of inspiration was the Alt-Ergo online demo presented in figure 4.9. It comes with many helpful examples, which can be automatically copied to the editor upon clicking on them. Later we extended this idea to a full interactive tutorial with explanations, rather than a simple list of copyable examples.

4.4 ACE EDITOR

At the same time as the layout of the 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 addi-

tional 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 own 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, the programmer may 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

```

1
2  styles [] {
3
4      .comment : style {
5          color          = #688557
6          italic         = true
7          ace_scope      = comment
8          textmate_scope = comment
9          pygments_scope = Comment
10     }
11
12 }
13
14 main : context {
15
16     : pattern {
17         regex      \= (//.*)
18         styles []  = .comment;
19     }
20
21 }
```

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 an optional dark and light theme. Depending on the user's choice, our colouring might be overridden. 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 the developer to manipulate the stack using the push and pop command.

```

1  : inline_push {
2      regex          \= (/\*)
3      styles []      = .comment;
4      default_style  = .comment
5      : pop {
6          regex      \= (\*/)
7          styles []  = .comment;
8      }
9  }
```

Solomonoff's syntax highlighter uses stack to correctly recognize comments and string literals enclosed in quotes and angle brackets. The rest of the 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 in 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 the website together with remaining JavaScript files. The Ace editor is then initialized using the following lines

```

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

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

```

1  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 [26]. This point is especially important, because Solomonoff is targeted 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 of available functions. Below is a list presenting some of the more important options.

```
1 editor.setOptions({
2   enableBasicAutocompletion: [{
3     getCompletions: (editor, session, pos,
4     prefix, callback) => {
5       callback(null, [{
6         name: 'subtract(',
7         value: 'subtract()',
8         score: 1,
9         meta: 'difference of two languages'
10      },
11      {
12        name: 'rpni(',
13        value: 'rpni()',
14        score: 1,
15        meta: 'RPNI inference algorithm'
16      },
17      {
18        name: 'rpni_mealy(',
19        value: 'rpni_mealy()',
20        score: 1,
21        meta: 'RPNI for Mealy machines'
22      },
23      {
24        name: 'ostia(',
25        value: 'ostia()',
26        score: 1,
27        meta: 'OSTIA inference for transducers'
28      },
29      {
30        name: 'compose(',
31        value: 'compose()',
32        score: 1,
33        meta: 'transducer composition'
34      },
35      {
```

```

36         name: 'inverse[' ,
37         value: 'inverse[]',
38         score: 1,
39         meta: 'transducer inversion'
40     }
41     ]);
42 },
43 ]],
44 enableSnippets: true,
45 enableLiveAutocompletion: true
46 });

```

The value field is the text that autocompletion will produce when selected. The meta argument provides a short explanation that will be shown to the user. We decided to set `enableLiveAutocompletion` 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 CTRL+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 the 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 an 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 result in syntax errors. For instance

```
1 | :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 in the main editor would be very misleading

```

1 | editor.setOptions({
2 |     enableBasicAutocompletion: [{
3 |         getCompletions: (editor, session, pos,

```

```

4      prefix, callback) => {
5          callback(null, [
6              ...
7              {
8                  name: ':eval',
9                  value: ':eval',
10                 score: 1,
11                 meta: 'evaluate transducer'
12             }
13             ...
14         ]);
15     },
16     ]],
17     enableSnippets: true,
18     enableLiveAutocompletion: true
19 });

```

On the other hand, not showing any hints related to REPL commands would seem like a major shortcoming of the online playground. To address this issue it was later decided that the REPL input line should also use Ace. As a result the website ended up with two instances of Ace editor. Both are 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 `:eval` would take the transducer name and then some input string. The visualization only needs transducer name

```
1 | :vis f
```

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

```
1 | :rand_sample f of_size 10
```

and

```
1 | :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 `of_size` and `of_length` argument decides which of the two modes of generation to use. The initial idea to address this issue was to add user help displayed by the `:?` command.

```
1 | > :?
```

```

2  :vis [ID]
3  Shows graph diagram of automaton
4  :rand_sample [ID] [of_size/of_length] [NUM]
5  Generates random sample of input:output pairs produced
6  by ths transducer
7  :ls
8  Lists all currently defined transducers
9  :clear
10 Clears REPL console
11 :run [ID] [STRING]
12 Runs pipeline for the given input
13 :unset [ID]
14 Deletes a variable
15 :is_det [ID]
16 Tests whether transducer is deterministic
17 :eval [ID] [STRING]
18 Evaluates transducer on requested input
19 ...

```

The user could then consult this cheat sheet to determine the 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.

```

1  getCompletions: (editor, session, pos, prefix, callback) => {
2      callback(null, [
3          {
4              name: ':eval',
5              value: ':eval',
6              snippet: ':eval ${1:transducer_name} \'${2:input_string}\'',
7              score: 1,
8              meta: 'evaluate transducer'
9          },
10         {
11             name: ':rand_sample of_size',
12             value: ':rand_sample',
13             snippet: ':rand_sample ${1:transducer_name} of_size ${2:number}',
14             score: 1,
15             meta: 'randomly generate sample'
16         },
17         {
18             name: ':rand_sample of_length',
19             value: ':rand_sample',

```

```

20         snippet: ':rand_sample ${1:transducer_name} of_length $
           {2:number} ',
21         score: 1,
22         meta: 'randomly generate sample'
23     },
24 ];
25 }

```

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 `${}` braces.

4.5 WEBASSEMBLY

WebAssembly [27] is a technology that provides a new type of code that can be run in modern web browsers, providing new functionality and significant performance improvements. The code for WebAssembly is not meant to be written by hand, rather it is designed to compile efficiently from low-level source languages such as C, C++, Rust, and so on.

WebAssembly allows code written in different languages to run in web applications at near natural speed [28, 29, 30]. This is of great importance for the web platform, as it previously could not be done.

Moreover, it's not necessary to know how to create WebAssembly code in order to use it. WebAssembly modules can be imported into a web application (or Node.js), and functions can be exported from them for use via JavaScript. JavaScript frameworks can use WebAssembly modules to gain tremendous performance benefits and new features, while making their functionality readily available to web developers.

The early versions of Solomonoff 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

```

1 function compile(){
2     console.log(input.value)
3     Module.ccall('compile',
4         'void',
5         ['string'],
6         [input.value])

```

```
7 | }  
8 | function runMealy(){  
9 |     console.log(output.value)  
10 |     var c = Module.ccall('run_global',  
11 |         'string',  
12 |         ['string'],  
13 |         [output.value])  
14 |     console.log(c);  
15 |     output.value = c;  
16 | }
```

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

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

that smoothened the process of WebAssembly integration. At that stage, the compiler was simple and such a 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 numerous errors.

The next library we attempted was CheerpJ. It was the most promising option. The project compiled successfully and could run in the browser. The problem was that support for WASM is still at an 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 the 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 a user interface in Java Swing technology.

The last available option was to use TeaVM [31]. It suffered similar problems to JWebAssembly. Parts of the Java standard library were not supported. Fragments of the compiler code had to be rewritten to use workaround functions.

For instance the

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

had to be rewritten as

```
1 | if(hashMap.containsKey(k)){  
2 |     x = hashMap.get(k);  
3 | }else{  
4 |     x = new X();  
5 |     hashMap.put(k,x);  
6 | }
```

Another problem was with porting all the libraries and dependencies. Parts of ANTLR [32] 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 the browser in any form and had to be removed. After enough modifications our code compiled successfully. When ran in the 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 a long and extensive history of being used as a server-side language with a plethora of tools and frameworks available - Spring being one of the most popular.

Bibliography

- [1] A. Mendoza-Drosik, "Multitape automata and finite state transducers with lexicographic weights," *ArXiv*, vol. abs/2007.12940, 2020. [Online]. Available: <https://arxiv.org/abs/2007.12940>
- [2] M. Sipser, *Introduction to the Theory of Computation 3rd Edition*.
- [3] S. Eilenberg, *Automata, Languages and Machines Vol. A*. Academic Press, 1974.
- [4] —, *Automata, Languages and Machines Vol. B*. Academic Press, 1976.
- [5] "A method for synthesizing sequential circuits," *Bell System Technical Journal*. pp. 1045–1079.
- [6] C. de la Higuera, *Grammatical Inference: Learning Automata and Grammars*. Cambridge University Press, 2010.
- [7] "Learning register automata: from languages to program structures." *Mach Learn*.
- [8] K. U. S. Stoyan Mihov, *Finite-State Techniques: Automata, Transducers and Bimachines*, 2019.
- [9] F. P. Mehryar Mohri and M. Riley, "Weighted finite-state transducers in speech recognition," *AT&T Labs – Research*, 2008.
- [10] M. Mohri, *Weighted Finite-State Transducer Algorithms. An Overview*. Springer, 2004.
- [11] —, "Weighted finite-state transducer algorithms an overview," *AT&T Labs*, 2004.

- [12] M.-P. Béal, O. Carton, C. Prieur, and J. Sakarovitch, "Squaring transducers: An efficient procedure for deciding functionality and sequentiality of transducers," in *LATIN 2000: Theoretical Informatics*, G. H. Gonnet and A. Viola, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2000, pp. 397–406.
- [13] V. M. Glushkov, *The abstract theory of automata*. Russian Mathematics Surveys, 1961.
- [14] P. W. Tsunehiko Kameda, "On the state minimization of nondeterministic finite automata," *IEEE Transactions on Computers*, 1970.
- [15] R. J. S. A. Bonchi F., Bonsangue M.M., "Brzozowski's algorithm (co)algebraically," *Lecture Notes in Computer Science*, 2012.
- [16] L. D'Antoni and M. Veanes, "The power of symbolic automata and transducers," *University of Wisconsin, Madison*.
- [17] J. A. Bondy, *Graph Theory With Applications*. GBR: Elsevier Science Ltd., 1976.
- [18] "The open group base specifications issue 7, 2018 edition." [Online]. Available: <https://pubs.opengroup.org/onlinepubs/9699919799/utilities/make.html>
- [19] *GNU Make*. [Online]. Available: <https://www.gnu.org/software/make/>
- [20] E. Martin, "Ninja." [Online]. Available: <http://www.aosabook.org/en/posa/ninja.html>
- [21] *The Ninja build system, v1.10.2*. [Online]. Available: <https://ninja-build.org/manual.html>
- [22] D. Röthlisberger, "Benchmarking the ninja build system." [Online]. Available: <https://david.rothlis.net/ninja-benchmark/>
- [23] *Maven Getting Started Guide*. [Online]. Available: <https://maven.apache.org/guides/getting-started/index.html>
- [24] "jgrapht." [Online]. Available: <https://github.com/jgrapht/jgrapht/blob/678f21f3f154bcec86974e43d949152e034fd8da/jgrapht-core/src/main/java/org/jgrapht/traverse/TopologicalOrderIterator.java>
- [25] J. Spurlock, *Bootstrap: Responsive Web Development*. O'Reilly Media, 2013. [Online]. Available: <https://books.google.pl/books?id=LZm7Cxgi3aQC>

- [26] T. R. Beelders and J.-P. L. du Plessis, "Syntax highlighting as an influencing factor when reading and comprehending source code," *Journal of Eye Movement Research*, vol. 9, no. 1, Dec. 2015. [Online]. Available: <https://bop.unibe.ch/JEMR/article/view/2429>
- [27] A. Haas, A. Rossberg, D. L. Schuff, B. L. Titzer, M. Holman, D. Gohman, L. Wagner, A. Zakai, and J. Bastien, "Bringing the web up to speed with webassembly," *SIGPLAN Not.*, vol. 52, no. 6, p. 185–200, Jun. 2017. [Online]. Available: <https://doi.org/10.1145/3140587.3062363>
- [28] A. Jangda, B. Powers, E. D. Berger, and A. Guha, "Not so fast: Analyzing the performance of webassembly vs. native code," in *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. Renton, WA: USENIX Association, Jul. 2019, pp. 107–120. [Online]. Available: <https://www.usenix.org/conference/atc19/presentation/jangda>
- [29] A. Rossberg, B. L. Titzer, A. Haas, D. L. Schuff, D. Gohman, L. Wagner, A. Zakai, J. F. Bastien, and M. Holman, "Bringing the web up to speed with webassembly," *Commun. ACM*, vol. 61, no. 12, p. 107–115, Nov. 2018. [Online]. Available: <https://doi.org/10.1145/3282510>
- [30] A. Haas, A. Rossberg, D. L. Schuff, B. L. Titzer, M. Holman, D. Gohman, L. Wagner, A. Zakai, and J. Bastien, "Bringing the web up to speed with webassembly," in *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2017. New York, NY, USA: Association for Computing Machinery, 2017, p. 185–200. [Online]. Available: <https://doi.org/10.1145/3062341.3062363>
- [31] Y. Asano and K. Kagawa, "Development of a web-based support system for object oriented programming exercises with graphics programming," in *2019 18th International Conference on Information Technology Based Higher Education and Training (ITHET)*, 2019, pp. 1–4.
- [32] T. Parr, *The Definitive ANTLR 4 Reference*. Pragmatic Bookshelf, 2013. [Online]. Available: <https://books.google.pl/books?id=gA9QDwAAQBAJ>