



Ordered Finite Action Transducers for High-Performance Stream Processing

Kristoffer Aalund Søholm
Sebastian Paaske Tørholm

July 17, 2015

Abstract

Efficient methods for regular string matching are well known, while the problem of regular string transduction (rewriting) is less explored. We introduce ordered finite action transducers (OFAT), building on the theory of streaming string transducers (SST) to produce an efficient two-phase transducer with good streaming behavior, that allows for execution of arbitrarily complex actions along the parsed path. We describe Kleenex, a programming language for expressing string transductions, and introduce a formalization to an OFAT with a limited subset of actions for which we can prove a worst case linear transduction time for fixed size automata. We also describe repg, an implementation of the Kleenex language, and its compilation process. In use cases we achieve good performance characteristics compared to both similar tools such as DReX and Ragel, as well as related tools such as RE2, PCRE and other regular expression libraries.

Thesis supervisor: Fritz Henglein

Acknowledgements

We would like to thank our thesis supervisor Fritz Henglein, as well as Niels Bjørn Bugge Grathwohl and Ulrik Rasmussen, Ph.D. students associated with the KMC project, for their excellent supervision and support during the entire project. We would also like to thank Mathias Bundgaard Svensson, Rasmus Wriedt Larsen, and René Løwe Jacobsen for their help with proofreading our thesis.

Contents

Contents	3
List of Figures	5
List of Tables	6
List of Theorems	7
1 Introduction	8
2 Preliminaries	11
2.1 Regular expressions	11
2.2 Finite automata	12
2.3 Ordered finite transducers	13
2.4 Path trees	16
2.5 Streaming string transducers	19
2.6 Symbolic automata	23
2.7 Action automata	24
3 The Kleenex Language	27
3.1 Overview	27
3.2 Core language	29
3.3 The Kleenex language	32
3.4 Expressivity of Kleenex	35
3.5 Time complexity bounds	36
4 Implementation	38
4.1 Compilation	38
4.2 Runtime	43
4.3 Pipeline implementation	45
4.4 Action implementation	45
4.5 Optimizations	45
4.6 Correctness	47
5 Evaluation	49
5.1 Benchmarks	49
5.2 Use cases	61
5.3 Comparison	68
6 Conclusion	72

<i>CONTENTS</i>	4
6.1 Related work	72
6.2 Future work	72
6.3 Closing remarks	73
7 References	75
A Appendix	79

List of Figures

1.1	Example of an INI-file.	8
1.2	An example of a simple INI parser written in Kleenex.	9
1.3	An example of a more complex INI transformation written in Kleenex.	9
2.1	Example of an NFST	14
2.2	Example of an NFST whose transduction is not single-valued	14
2.3	Example of an OFT	15
2.4	The path tree resulting from simulating the OFT in Figure 2.3 on the input <u>aaaa</u>	16
2.5	The path tree from Figure 2.4, with valuations annotated on each node.	17
2.6	The update that takes place on Figure 2.4 if another <u>a</u> is consumed	18
2.7	Continuing from Figure 2.6, we consume two more <u>as</u>	18
2.8	Demonstration of removal of deterministic states from a path tree	19
2.9	Example of an OFT to SST conversion	23
2.10	Example of an SST that performs a swapping operation	24
2.11	Example of an OFAT that replicates the functionality of Figure 2.10	24
2.12	Example of an oracle FST and action automaton	26
3.1	A Kleenex program, and its desugared version	32
3.2	The desugaring function for converting Kleenex actions into core actions.	33
3.3	The desugaring function for converting Kleenex terms into core terms.	34
3.4	The desugaring function for converting regular expressions into Kleenex terms.	34
3.5	The desugaring function for desugaring suppressed core terms into core terms.	35
3.6	The oracle and action automata corresponding to the grammar from Figure 3.1.	36
4.1	Kleenex compilation flowchart	39
4.2	The data type representing the Kleenex AST	39
4.3	Data type for recursive μ -terms	40
4.4	Recursive conversion of μ -terms to a OFT	41
4.5	Visualizations of <u>odd_even</u> from Figure 3.1	42
4.6	Example code generated by <u>repg</u>	43
4.7	Diagram of the Kleenex runtime	44
4.8	Common pattern of Kleenex programs	47
5.1	<u>flip_ab</u> run on a file with an average line length of 1000 characters.	52
5.2	<u>patho2</u> run on a file with an average line length of 1000 characters.	53

5.3	The program text for the <code>rot13</code> program	53
5.4	Benchmark of <code>rot13</code>	54
5.5	Benchmark of <code>thousand_sep</code>	55
5.6	The program text for the <code>csv_project3</code> program.	55
5.7	Benchmarks for <code>csv_project3</code>	56
5.8	The program text for the <code>iso_datetime_to_json</code> program.	56
5.9	Throughput for the program <code>iso_datetime_to_json</code>	57
5.10	<code>csv_project3</code> with and without suppressed bitcode optimization.	58
5.11	Bitcode vs. bytecode implementation of <code>csv_project3</code>	59
5.12	Bitcode vs. bytecode implementation of <code>rot13</code>	59
5.13	Bitcode vs. bytecode implementation of <code>thousand_sep</code>	60
5.14	Program text for the program <code>ini2json</code>	61
5.15	Throughput of the program <code>ini2json</code>	62
5.16	Man-in-the-middle attack on HTML pages with forms	62
5.17	The <code>apache_log</code> program, which parses and annotates apache log files.	63
5.18	Throughput comparison on the <code>apache_log</code> program.	64
5.19	Lexer for the instructional programming language <code>PL/0</code>	64
5.20	Syntax highlighter for Kleenex	65
5.21	The program <code>irc</code> , which parses the IRC protocol.	66
5.22	Throughput of the program <code>irc</code>	67
5.23	Kleenex versions of the programs presented in the DReX paper from POPL 2015 [1]	68
5.24	Throughput of the program <code>drex_swap-bibtex</code> on 2 MB of input data.	69
5.25	Execution time of the program <code>drex_swap-bibtex</code> on 2 MB of input data.	69
5.26	The Ragel implementation of <code>flip_ab</code>	71
A.1	The intermediate language definition	80
A.2	The code used for syntax highlighting all Kleenex code in this paper.	81

List of Tables

2.1	The operations on regular expressions and their standard interpretation on sets of strings. Here E^n means E concatenated with itself n times.	11
2.2	The extended regular expression syntactical elements, and their translation, T , to basic regular expressions.	12
5.1	Version numbers of tools used for benchmarking	51

List of Theorems

1	Definition (Regular expression[3])	11
2	Definition (Nondeterministic finite automaton)	12
3	Definition (Deterministic finite automaton[3])	13
4	Definition (Nondeterministic finite state transducer)	13
5	Definition (Ordered finite transducer)	15
6	Definition (Functional semantics of OFTs)	16
7	Definition (Path tree)	16
1	Theorem	20
1	Corollary	20
8	Definition (Streaming String Transducer[10])	21
9	Definition (Functional semantics of SSTs)	22
10	Definition (OFT to SST determinization)	22
2	Theorem	22
11	Definition (Ordered finite action transducer)	24
12	Definition (Functional semantics of OFATs)	25
13	Definition (Kleenex computational environment)	29
14	Definition (Action)	29
15	Definition (Core action)	29
16	Definition (Kleenex core terms and syntax)	30
17	Definition (Well-formedness of Kleenex programs)	30
18	Definition (Grammar)	31
19	Definition (Right-regular grammar with actions)	31
3	Theorem	31
20	Definition (Kleenex core semantics)	32
21	Definition (Kleenex actions)	33
22	Definition (Kleenex terms)	33
23	Definition (Kleenex desugaring)	33
4	Theorem	33
24	Definition (Kleenex semantics)	35
5	Theorem	35
1	Conjecture	36
6	Theorem	36
7	Theorem	37

Chapter 1

Introduction

Regular expressions are widely used for practical programming: they provide a convenient and succinct way of extracting data from a wide variety of structured (or even less structured) text. Compilation methods that allow for regular expressions to be compiled down to efficient deterministic finite automata are well known, and theoretically well founded. In spite of this implementations often make use of less efficient methods like backtracking, giving a worst case exponential running time.

Why are less efficient methods used in practice? One possible answer is that pure regular expressions simply are not expressive enough for the problems people want to use them for in practice. Instead of rethinking them from the ground up, many implementations choose to add new functionality. Features like backreferences increase the expressiveness, but leaves the theory behind. These features are popular nonetheless, as they increase the expressiveness, allowing people to solve more of their problems.

```
[haskell]
compiler=ghc
flags=
[c]
flags=-Wall -O3
compiler=gcc
```

Figure 1.1: Example of an INI-file.

Even so, these implementations still hit limitations that are not theoretically necessary. Often problems are tackled on a line-by-line basis, even if the problem could be tackled in a single expression. Take the problem of parsing an INI file. For those unfamiliar with the format, an example can be seen in Figure 1.1. The INI file format is inherently regular: it has a number of sections, each containing a header, followed by a number of key-value pairs. In essence, this format should be able to be parsed by the regular expression $([. + ?] \backslash n (. * ? = . * ? \backslash n) *) *$. In practice, the problem would not be tackled that way. Instead, one would go line by line, essentially programming a state machine that works on lines, making use of regular expressions to parse each line.¹

¹This is the point at which the actor in the infomercial would say: “There has got to be a better way!”

<pre> main := (header keyvalue*)* header := ~/\[/ "Section: " /[^\n]+/ ~/] / \n/ keyvalue := key ~=/ value ~/\n/ key := " Key: " /[^\n]+/ "\n" value := " Value: " /[^\n]+/ "\n" </pre>	<pre> Section: haskell Key: compiler Value: ghc Key: flags Value: Section: c Key: flags Value: -Wall -03 Key: compiler Value: gcc </pre>
---	--

(a) Kleenex program for parsing INI files (b) Example output

Figure 1.2: An example of a simple INI parser written in Kleenex.

```

main := (header keyvalue* output)*

header  := ~/\[/ header@[^\n]+/ ~/] / ~\n/
keyvalue := ~/compiler=/ compiler@[^\n]+/ ~/\n/
          | ~/flags=/ flags@[^\n]+/ ~/\n/

output := "To compile " !header " run: "
          !compiler " " !flags "\n"

```

(a) Processing compiler configurations with Kleenex.

```

To compile haskell run: ghc
To compile c run: gcc -Wall -03

```

(b) The output on our example INI.

Figure 1.3: An example of a more complex INI transformation written in Kleenex.

This thesis is about the new programming language Kleenex, a domain specific programming language that is

Fast : Kleenex is built on solid theory that allows us to do efficient (i.e. linear time in the size of the input) regular rewriting of text.

Expressive : Going beyond pure regular expressions, it allows for complex rewriting that would be hard to express even in extended regular expression engines.

Pleasant to use : This is rather subjective, but we have found it quite easy and pleasant to express some rather complex text transformations compared to our experience with other tools.

To demonstrate, Figure 1.2 shows a program that parses simple INI files, and the output produced from evaluating it on Figure 1.1.

While short and elegant, the example can get more interesting. The program shown preserves the order and structure of the input, but we can do more than that. Through what we call “actions”, we allow for data to be stored in registers as the matching is going on, allowing for more complex rewriting as shown in Figure 1.3

In this thesis we will give the theoretical background for Kleenex, as well as an understanding of the implementation, possibilities and limitations the language has to offer.

Overview

In Chapter 2 we present an introduction to transducers, focusing on the compilation of ordered finite state transducers (OFT) into deterministic streaming string transducers (SST), guaranteeing linear time transductions for fixed size automata. We also introduce the notion of an ordered finite action transducer (OFAT), a model that lets us extend the transductions to non-rational relations.

In Chapter 3 we present Kleenex, a declarative language and expressive language for string rewriting, with clear semantics for the handling of ambiguity. Furthermore, we define the semantics of the language based on the OFAT computational model, and prove linear worst case time complexity.

In Chapter 4 we present `repg`, a compiler for the Kleenex language that compiles the language to either an SST or a pair of sibling transducers derived from an OFAT. Additionally, we present several optimizations in the compilation process.

In Chapter 5 we present benchmarks that document consistently high performance in many different use cases. In addition we present a number of more complex programs, demonstrating the expressivity of the language.

Our contributions

This project was done as a part of, and in collaboration with, the Kleene Meets Church (KMC) research project at the University of Copenhagen. As a result of this, our results build heavily on their theory of efficient transductions using streaming string transducers, and their implementation of `repg`.

Our contributions to the project are the design and implementation of pipelines and actions in the language, an alternative formalization of Kleenex in terms of our OFAT model, a time bound for the full Kleenex language, the exploration of use cases for Kleenex, and benchmarks against regular expression matching tools. We also contributed optimizations targeting both the compile time and code generation, bug fixes, testing and tool-building for the project.

During the project we have also co-authored a paper[2], currently undergoing review for the 2016 Symposium on Principles of Programming Languages (POPL), covering the same topic as our thesis. The paper is attached as an appendix to the thesis.

Chapter 2

Preliminaries

2.1 Regular expressions

At the foundation of our thesis is the concept of a *regular expression*. A regular expression can be seen as a succinct way of describing a (potentially infinite) set of strings.

An example is the set of strings over $\{\underline{a}, \underline{b}, \underline{c}\}$ that start with an $\underline{a}$ and is followed either by 2 $\underline{b}$ s or by any number of $\underline{c}$ s. $\underline{abb}$, $\underline{accccc}$ and $\underline{a}$ all belong to this set, but $\underline{abbb}$ and $\underline{abc}$ do not. To describe this set, we would use the regular expression $\underline{a}(\underline{bb}|\underline{c}^*)$. Here $\underline{c}^*$ uses the Kleene star, which represents 0 or more repetitions of the $\underline{c}$ contained within.

Definition 1 (Regular expression[3]). We say that E belongs to $RE(\Sigma)$, the *regular expressions* over an alphabet Σ if E has one of the following forms:

1. $E = 1$,
2. $E = \underline{a}$, for $\underline{a} \in \Sigma$,
3. $E = E_1E_2$, for $E_1, E_2 \in RE(\Sigma)$,
4. $E = E_1|E_2$, for $E_1, E_2 \in RE(\Sigma)$,
5. $E = E_1^*$, for $E_1 \in RE(\Sigma)$.

The standard interpretation of a regular expression $E \in RE(\Sigma)$ is a set of strings, $L(E) \subseteq \Sigma^*$, called the *language* of the regular expression. The definition of L is shown in Table 2.1.

Name	E	$L(E)$
One	1	$\{\varepsilon\}$
Literal	$\underline{a}$	$\{\underline{a}\}$
Concatenation	E_1E_2	$\{s_1s_2 \mid s_1 \in L(E_1), s_2 \in L(E_2)\}$
Alternation	$E_1 E_2$	$L(E_1) \cup L(E_2)$
Kleene star	E^*	$\{\varepsilon\} \cup \bigcup_{n=1}^{\infty} L(E^n)$

Table 2.1: The operations on regular expressions and their standard interpretation on sets of strings. Here E^n means E concatenated with itself n times.

Commonly an extended shorthand syntax is used, which we call the extended regular expression syntax. The translation of these extended operations can be seen in Table 2.2.

E_{Ext}	$T(E_{Ext})$
E	E , where $E \in RE(\Sigma)$
E^+	$T(E)T(E)^*$
$E^?$	$T(E) \mid 1$
$E\{n\}$	$\underbrace{T(E)T(E) \dots T(E)}_{n \text{ times}}$
$E\{, m\}$	$\underbrace{T(E^?)T(E^?) \dots T(E^?)}_{m \text{ times}}$
$E\{n, \}$	$T(E\{n\})T(E)^*$
$E\{n, m\}$	$T(E\{n\})T(E\{, k\})$, where $k = m - n$
$[abc]$	$(a \mid b \mid c)$
$[a - z]$	$(a \mid b \mid c \mid \dots \mid y \mid z)$
$\cdot$ (dot)	$(x_0 \mid x_1 \mid \dots \mid x_n)$, where $\{x_0, \dots, x_n\} = \Sigma$
$[\wedge abc]$	$(x_0 \mid x_1 \mid \dots \mid x_n)$, where $\{x_0, \dots, x_n\} = \Sigma \setminus \{a, b, c\}$

Table 2.2: The extended regular expression syntactical elements, and their translation, T , to basic regular expressions.

We say that the language of a regular expression is *regular*. Note that not all subsets of Σ^* are regular. An example of this is the set $\{\underline{a}^n \underline{b}^n \mid n \in \mathbb{N}\}$ which is not regular, but *context-free*, a larger class.[3]

2.2 Finite automata

A common problem one wishes to solve in regards to regular expressions is the acceptance problem, i.e. determining whether a string belongs to the language of a given expression. The theory behind this problem is well-explored, and the problem is commonly solved by compiling the regular expression into a *non-deterministic finite automaton* (NFA) using Thompson's Construction.[3, 4, 5]

An NFA is simply a state machine: you have a number of *states*, each of which has a number of *transitions* to other states. Each of these transitions is marked with a symbol, representing a symbol you read from the input, or an ϵ representing the empty string. Starting in the *initial state*, the machine *accepts* a string if there exists a path you can follow to the *final state*, such that the concatenation of symbols along the edges match the input string.

Definition 2 (Nondeterministic finite automaton). Let Σ be a finite input alphabet.

We then define a *nondeterministic finite automaton* (NFA) over Σ to be a structure, $\mathcal{N} = (Q, q^-, q^f, E)$, where

- Q is a finite set of *states*,
- $q^- \in Q$ is the *initial state*,
- $q^f \in Q$ is the *final state*,
- $E \subseteq Q \times (\Sigma \cup \{\epsilon\}) \times Q$ is the transition relation

We understand (q, s, q') to be in E if you can transition from q to q' by reading symbol s . Furthermore, we say that $\text{Sim}_{\mathcal{N}} : \Sigma^* \rightarrow \mathbb{2}$ is a *simulation* of $\mathcal{N}$, if $\text{Sim}_{\mathcal{N}}(s) = 1$ precisely when $\mathcal{N}$ accepts s .

The non-determinism of this machine lies in the E -relation: it is possible that there are two transitions for the next symbol or a transition along an ε -edge which consumes no input. This nondeterminism can make simulation of an NFA slow, and it can therefore be favorable to determinize the machine into a deterministic finite automaton (DFA).

Definition 3 (Deterministic finite automaton[3]). Let Σ be a finite input alphabet.

We then define a *deterministic finite automaton* (DFA) over Σ to be a structure, $\mathcal{N} = (Q, q^-, F, E)$, where

- Q is a finite set of states,
- $q^- \in Q$ is the initial state,
- $F \subseteq Q$ is the set of final states,
- $E : Q \times \Sigma \rightarrow Q$ is the transition function

A DFA is an NFA where each state has precisely one outgoing transition for each symbol, and none for ε .

It is possible to determinize any NFA into an equivalent DFA using a method known as the subset construction[6]. In essence, a DFA is built where each state in the resulting DFA corresponds to the set of NFA states that are simultaneously possible.

Such an automaton can perform acceptance testing for a regular language in linear time by consuming a symbol on each transition. This efficiency in time complexity comes at a cost, as the resulting machines can grow exponentially in their number of states.

2.3 Ordered finite transducers

In practice, we are often interested in more than acceptance testing, namely to perform text rewriting (so called *transductions*). In order to accomplish this, we need the concept of a *nondeterministic finite state transducer* (NFST)[7].

To provide some intuition, we can liken an NFST to an NFA. The only difference between the two types of automaton is that, on each transition, an NFST is able to both read and *output* a symbol, where an NFA can only read symbols.

Definition 4 (Nondeterministic finite state transducer). Let Σ and Γ be two given finite input and output alphabets.

We then define a *nondeterministic finite state transducer* (NFST) over Σ, Γ to be a structure, $\mathcal{T} = (Q, q^-, q^f, E)$, where

- Q is a finite set of states,
- $q^- \in Q$ is the initial state,
- $q^f \in Q$ is the final state,
- $E \subseteq Q \times (\Sigma \cup \{\varepsilon\}) \times (\Gamma \cup \{\varepsilon\}) \times Q$ is the transition relation.

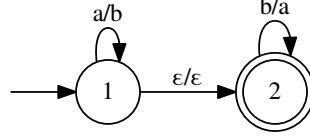


Figure 2.1: Example of an NFST which takes a sequence of 0 or more as, followed by 0 or more bs, and swaps each a for a b and vice versa.

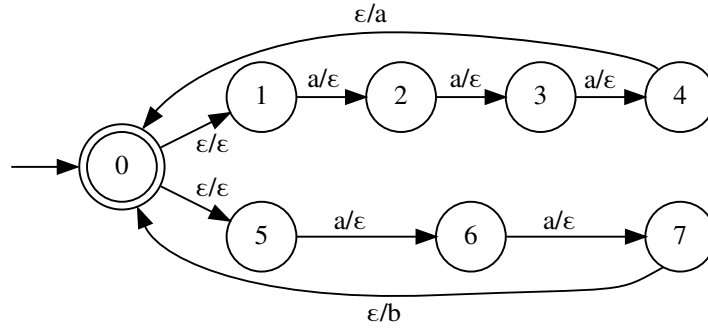


Figure 2.2: Example of an NFST whose transduction is not single-valued. For the input aaaaa it may produce an output of either ab or ba.

We understand (q, s, t, q') to be in E if you can transition from q to q' by reading symbol s and outputting t .

Figure 2.1 shows an example of such a transducer.

The addition of an output to an NFA has an important consequence: the result is no longer guaranteed to be *single-valued*, meaning that an NFST may have several valid transductions for any one input. An example illustrating this is given in Figure 2.2. The intuition is that while multiple paths may lead to an accepting state in an NFA, the precise path does not affect the end result. For NFSTs the path is key, since it determines the transduced output.

Similarly to the correspondence between NFAs and regular languages described above, NFSTs have a correspondence to the *rational relations*. The rational relations are a natural extension of regularity, and both notions can be defined in terms of *rational subsets*. A rational subset of a set S is a subset that can be constructed from finite applications of union, product, and Kleene star on the singleton subsets of S . The regular languages on Σ are precisely the rational subsets of Σ^* , and similarly the rational relations from Σ to Γ are the rational subsets of $\Sigma^* \times \Gamma^*$ [8].

In order to disambiguate the resulting transductions, we will extend the NFST model by introducing an index on each edge leaving a state, a construction stemming from a rejected version of [2]. This index indicates a preference for which edge to take, and allows us to pick the successful transduction that is lexicographically least as the canonical one.

This gives us an *ordered finite transducer* (OFT), which is defined as follows:

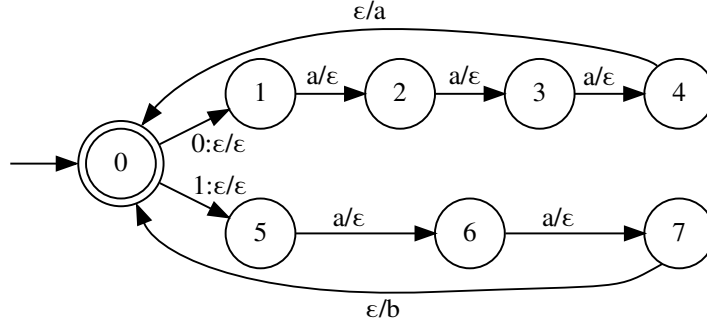


Figure 2.3: Example of an OFT. Indices are omitted on deterministic edges.

Definition 5 (Ordered finite transducer). Let Σ and Γ be two finite input and output alphabets.

We then define an *ordered finite transducer* (OFT) over Σ, Γ to be a structure, $\mathcal{T} = (Q, q^-, q^f, E, n)$, where

- Q is a finite set of states,
- $q^- \in Q$ is the initial state,
- $q^f \in Q$ is the final state,
- $n \in \mathbb{N}_0$ is an upper bound for the indices,
- $E \in Q \times \{0, \dots, n\} \times (\Sigma \cup \{\varepsilon\}) \rightarrow (\Gamma \cup \{\varepsilon\}) \times Q$ is the *indexed transition relation*, such that if $(q, i, \varepsilon) \in \text{dom}(E)$, then $\nexists a \in \Sigma : (q, i, a) \in \text{dom}(E)$.

For brevity we will usually omit indices on edges that are deterministically taken, since any index could be attributed to such an edge without affecting the lexicographical order of possible candidate paths.

An OFT based on the ambiguous NFST from Figure 2.2 can be seen in Figure 2.3. For this example we have chosen to prioritize the path transducing aaa into a over the path transducing aa into b.

We can now define what we mean by lexicographical ordering of paths. By a *path* in $\mathcal{T}$, we mean a sequence of consecutive transitions in $\mathcal{T}$:

$$q_0 \xrightarrow{i_1:x_1/y_1} q_1 \xrightarrow{i_2:x_2/y_2} \dots \xrightarrow{i_n:x_n/y_n} q_n$$

Here we use $q \xrightarrow{i:x/y} q'$ to denote that $E(q, i, x) = (y, q')$.

We say that the path has *index label* $i_1 i_2 \dots i_n$, *input label* $x_1 x_2 \dots x_n$ and *output label* $y_1 y_2 \dots y_n$.

We say that a state q is *active* on input s , if there exists a path from q^- to q with input label s , and it contains no outgoing ε transitions. We say that a path *can transduce* $s \in \Sigma^*$ if it starts in q^- , ends in q^f , and has input label s . We say that a path is *problematic* if it contains a non-empty loop with an empty input label. We say that $\mathcal{T}$ *transduces* s to $t \in \Gamma^*$ (via path p) if

- p can transduce s ,

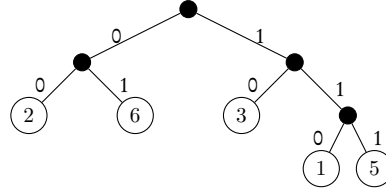


Figure 2.4: The path tree resulting from simulating the OFT in Figure 2.3 on the input aaaa.

- p is not problematic,
- there exists no other non-problematic path p' that can transduce s , such that the index label of p' is lexicographically smaller than that of p , and
- the output label of p is t .

Definition 6 (Functional semantics of OFTs). An OFT denotes a partial function $\llbracket \mathcal{T} \rrbracket : \Sigma^* \rightarrow \Gamma^*$, such that $(s, t) \in \llbracket \mathcal{T} \rrbracket$ precisely if $\mathcal{T}$ transduces s to t .

We say that a $\text{Sim}_{\mathcal{T}}$ simulates $\mathcal{T}$, if $\text{Sim}_{\mathcal{T}} = \llbracket \mathcal{T} \rrbracket$.

This provides a canonical transduction for any input that has multiple possible paths through the transducer.

2.4 Path trees

OFTs resolve the problem of how to perform deterministic transductions of regular languages. Where one problem is solved, another arises: how can we efficiently simulate a nondeterministic OFT on deterministic hardware?

To address this problem, we will now describe a path tree [9]. A path tree is a tree structure that captures the possible branching paths we can have followed while evaluating an OFT on a given input. Our description differs from the cited one, in that we do not restrict the trees to be binary.

Definition 7 (Path tree). Let $\mathcal{T} = (Q, q^-, q^f, E, n)$ be an OFT with input alphabet Σ , and let $s \in \Sigma^*$.

A *path tree*, $\mathcal{PT}(\mathcal{T}, s)$ is a directed, ordered tree with nodes labeled by Q , edges labeled by $\{0, \dots, n\}$, such that:

- The leaves are precisely the active states in $\mathcal{T}$ on input s .
- The path from the root to a leaf, q , corresponds to the lexicographically least path, p , from q^- to q , with each inner node in the tree corresponding to a state at which a non-deterministic choice was made, and the edge following it being labeled by the index label chosen by p .

As an example of this, consider the path tree in Figure 2.4, which results from evaluating the OFT from Figure 2.3 on the string aaaa. Every branching node in this tree corresponds to a nondeterministic choice in the evaluation of the OFT. The number in a given leaf corresponds to the state of the OFT after making the nondeterministic choices indicated by the path from the root to the given leaf. I.e.,

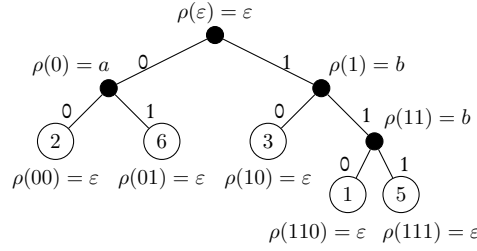


Figure 2.5: The pathtree from Figure 2.4, with valuations annotated on each node.

one possible path is picking the lower path (labelled 1), consuming $\underline{aa}$, then picking the upper path (labelled 0), consuming $\underline{aa}$ and ending up in state 3. This corresponds to the state found by starting in the root of the path tree, picking 1, followed by picking 0.

We define the *index label* of a node to be the concatenation of the edge labels along the path from the root to the node.

When drawing a path tree we will always order the edges in ascending index order. This allows us to transfer the lexicographical preference into the structure of our tree: a leaf l_1 has a lexicographically smaller index label than a leaf l_2 if l_1 is to the left of l_2 , i.e. l_1 comes before l_2 in an in-order traversal of the tree.

To allow us to keep track of the transduced output produced along each of the paths, we associate with each node a *valuation*, $\rho : \mathcal{PT} \rightarrow \Gamma^*$, where, if t' is a node in state s' , and t is its parent in state s , then $\rho(t')$ is the output label of the path from s to s' . The reasoning behind keeping the valuation separate from the path tree structure itself is explained in Section 2.5.

In Figure 2.5 an example of such a valuation is shown.

Simulating an OFT using path trees

A path tree captures the possible branching paths of execution when evaluating an OFT on a specific input. This makes it a natural choice as a data structure for simulating their execution. We will now outline how it can be used for this purpose.

The simulation can be broken into three steps:

1. Create an initial path tree for the input ε .
2. For each consumed input character, update the current path tree based on the character.
3. When all characters have been consumed, determine if the transduction was successful.

The initial path tree is created by evaluating the ε -closure of q^- .

To update the tree with a new input character we take each leaf in the path tree, simulate the OFT from the associated state on the character, and replace the old leaf with the resulting state. If a non-deterministic choice has to be made we introduce it as a branch. Figure 2.6 shows such an update. We will call this *stepping* the tree with the given symbol.

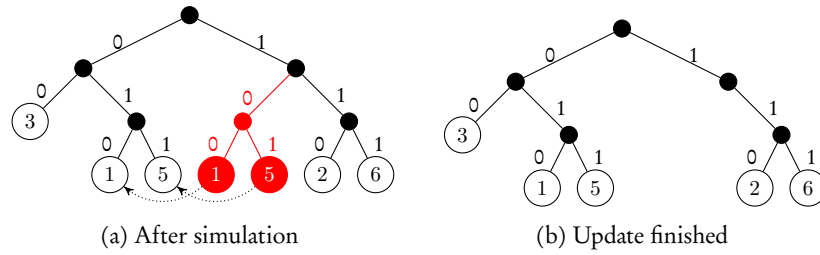


Figure 2.6: The update that takes place on Figure 2.4 if another $\underline{a}$ is consumed. (a) Each leaf is stepped with an $\underline{a}$, duplicate states are highlighted. (b) Duplicate states are removed, keeping the leftmost one.

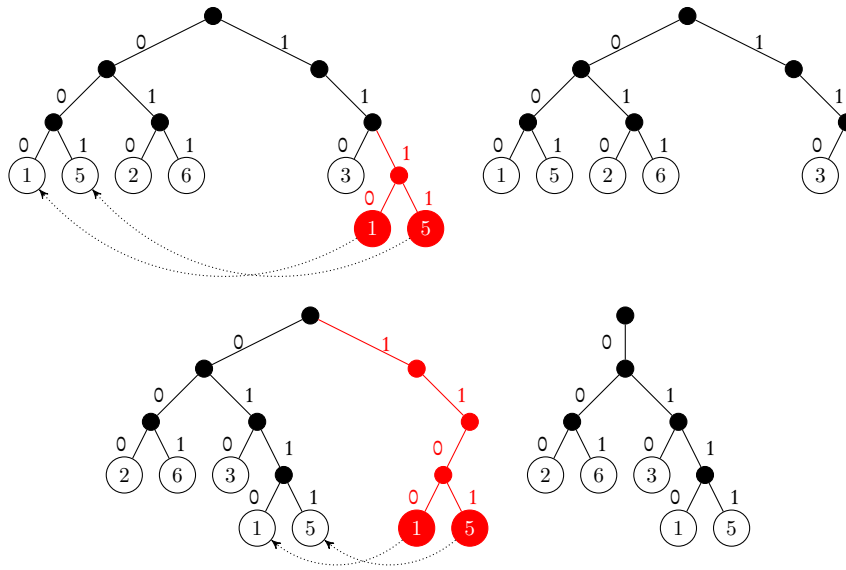


Figure 2.7: Continuing from Figure 2.6, we consume two more as. Observe how, after 7 as, we know that a matching path must start by making a choice of 0.

In performing this stepping, we keep track of whether we encounter a final state in the OFT. If we do, we annotate that the node is final, and associate with it the output label of the path up till the final state.

If there are duplicate leaves after an update all but the leftmost one can be eliminated, since this leaf has the lexicographically least index label. In addition to limiting the number of leaves in the tree to at most one for each state in the OFT, this also allows us to extract which nondeterministic choices have to be made as soon as we know for certain.

In the path tree we built with the input aaaaaaa in Figure 2.7, we see that all candidate paths start with a choice of 0. Thus we know with certainty that the OFT must follow this particular branch up until the next point of nondeterminism. This allows us to perform our transduction in a *streaming* fashion, understood in the sense that an output action on the OFT is performed as soon as we are certain that the edge with the output action will be visited.

It is worth noting, that we in our described algorithm follow ε -transitions after

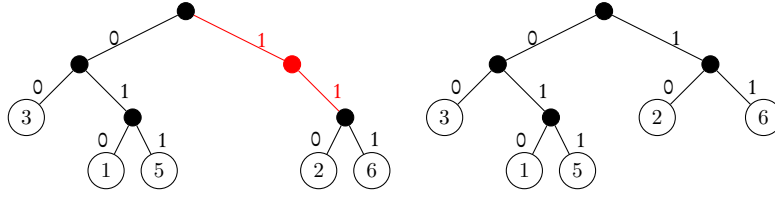


Figure 2.8: Continuing from Figure 2.6, we demonstrate how deterministic states can be removed, to make the resulting path tree smaller.

reading the input symbol. In the general case, ε -transitions should be consumed prior to consuming the input symbol, as it otherwise could result in a path tree where continuing the transduction from an inner node is required, rather than just from the leaves. For our uses, we ignore this problem, as we will only ever construct OFTs where each node has only ε -transitions, or only symbol transitions, and never a mixture of the two.

Note that the pruning method above is not optimal in the sense of resolving the choices at the earliest possible point. For instance, in Figure 2.6 both of the leaves along the rightmost path can be pruned: 2 since the path 110 will always be inferior to the 011 chosen in the 5 node, and 6 since 111 will always be inferior to the 00 chosen in the 3 node. This is because the input labels of the paths overlap in both cases.

There exists an *optimal pruning method* [9], which allows for *optimally streaming* simulation of an augmented NFA (aNFA), a model very similar to the OFT. The key difference between the two methods is that of coverage, i.e. which states can act as possible replacements for a given state in the path tree. For instance, in Figure 2.3 states 3 and 6 both cover each other, since either state goes to state 0 if given an $\bar{a}$. Combining such a coverage relation with the pruning method based on index labels results in optimally streaming behavior. Unfortunately, computing this coverage relation for an aNFA is PSPACE-hard. The simpler pruning still produces good streaming behavior, and is tractable to compute.

In order to minimize the size of the path tree, we can remove its *deterministic states*, which are non-leaf nodes with only one child. In doing so, the valuation of its sole child node becomes the concatenation of the two valuations. Such a removal is illustrated in Figure 2.8. The index retained is the one from the parent, to ensure preservation of the tree order. We will call such a tree a *reduced path tree*.

2.5 Streaming string transducers

Now that we have a method we can use to simulate an OFT, an obvious question becomes whether we can perform this simulation more efficiently. Currently we have to simulate each non-deterministic branch for each symbol we read. We would like to avoid this if at all possible, and so we will look at the reasoning which leads us to *streaming string transducers* as a better solution[2].

The first observation we make is that if we have two path trees with identical branching structure and identical leaves, the path trees created from stepping an additional input character on both will be identical as well. Therefore, we can hope to “recycle” or precompute these branching structures in advance, thus determinizing the OFT.

The main problem is that the path trees, apart from their tree structure, also carry the function ρ which associates a valuation with each node. We might have two path trees with identical structure, but different valuations associated with the nodes. Luckily, it turns out that the valuation of an updated path tree, ρ' , can be described as a function of ρ . Furthermore, as the following theorem shows, we can prove the update to be both *copyless*, such that no $\rho(t)$ is used twice in defining ρ' , and *hierachical*, such that $\rho'(t)$ only depends on the values of $\rho(t')$ where t' are in the subtree rooted in t .

Theorem 1. *Let $\mathcal{T}$ be a path tree with valuation function ρ . Furthermore, let $\mathcal{T}'$ and ρ' be the path tree and valuation function resulting from updating $\mathcal{T}$ with the consumption of $a \in \Sigma$.*

Then it holds that

1. *For each node t in $\mathcal{T}$, $\rho'(t) = X_1 X_2 \dots X_n c$, with each X_i being $\rho(t_i)$ for some node t_i belonging to the subtree rooted in t_{i-1} , with $t_0 = t$, and c being a constant.*
2. *For each node t in $\mathcal{T}$, the value of $\rho(t)$ is used in the definition of $\rho'(t')$ for at most one t' .*

Sketch of proof. In an update, we have two phases that occur: First we do a simulation of each leaf on the next input character. Secondly, we remove deterministic nodes, collapsing their input and output edges. For each of these phases, we will consider what change would take place in the valuation function.

For the first phase, observe first that any internal nodes remain unchanged. Let us now look at a leaf, t , stepping from state s to s' . If this path has output label y , we get that $\rho'(t) = \rho(t)y$. Finally, any newly introduced states will have their valuation be the output label of the path from the parent to itself.

Based on this, we can convince ourselves that ρ' will satisfy both of the above-mentioned items.

Let us now consider the second phase. If no nodes are removed, no change is made to the valuation. We will therefore consider the case where one node is removed, and show that the resulting valuation retains the desired properties. If more than one node is removed, we can consider it as a sequence of single removals, and therefore only need to consider this one case.

Let ρ'' be a valuation for which the desired properties holds, and t be the deterministic node we wish to remove, and t' be its only child. In removing t , we set $\rho'(t') = \rho''(t)\rho''(t')$, setting $\rho'(t) = \varepsilon$. No other valuation is changed. Since we don't introduce any new uses on ρ , and we know that the desired properties hold for ρ'' , we can convince ourselves that they also hold for ρ' . $\square$

Corollary 1. *For the root node, t_ε , $\rho(t_\varepsilon)$ is only ever appended to.*

Proof. Since no other node has the root node in its subtree, any update must either discard the value stored in $\rho(t_\varepsilon)$, or append to it.

The only time a value is discarded, is when its node is pruned for not being on any optimal paths. However, the root will always be on an optimal path, and is therefore never pruned. $\square$

With this in mind, we can now consider how to construct an automaton that allows us solve the problem in a more efficient manner.

Consider the idea of constructing a DFA with reduced path trees as the nodes, and the edges reflecting how a path tree would transform on a given input. Having such an automaton, it would be possible to simulate the underlying OFT in linear time if we ignore the outputs.

Naturally we would first have to argue that the number of reachable path trees is finite. However, this follows easily from the observations that:

- Each path tree only can contain a given state once as a leaf node.
- Each inner node in a reduced tree has to be branching.
- There are a finite number of labels that can be associated with each edge in the tree.

This method of DFA construction is similar to the subset construction for NFAs. There is one major difference, however: in preserving the relationship between the active states, we can make use of a key piece of information from Theorem 1 to expand the automaton to also include outputs: The valuation after an update can be computed solely based on the current valuation.

Consider an augmentation of our machine model with a finite number of *registers*, each capable of holding a string in the output alphabet. In order to allow for updates to these registers, we furthermore associate with each edge a set of register updates, which compute the new register values based on the former ones.

If we, for each possible index label in all the possible reduced path trees, associate a corresponding register, we can then use this register to keep track of the ρ -value of the node at that index. With each edge, we associate an appropriate register update, based on Theorem 1.

The final states are the reduced path trees containing a final state.

Such a machine has already been described in the literature as a *streaming string transducer*, and we will now provide a formal definition of it.

Definition 8 (Streaming String Transducer[10]). Let Σ and Γ be two given finite input and output alphabets.

We then define a deterministic *streaming string transducer* (SST) over Σ and Γ to be a structure, $\mathcal{S} = (X, Q, q^-, F, \delta_1, \delta_2)$, where

- X is a finite set of *register variables*,
- Q is a finite set of *states*,
- $q^- \in Q$ is the *initial state*
- F is a partial function, $Q \rightarrow (\Gamma \cup X)^*$ mapping each *final state* $q \in \text{dom}(F)$ to a word $F(q) \in (\Gamma \cup X)^*$, such that each $x \in X$ occurs at most once in $F(q)$,
- δ_1 is a transition function $Q \times \Sigma \rightarrow Q$,
- δ_2 is a *register update* function $Q \times \Sigma \times X \rightarrow (\Gamma \cup X)^*$, such that for each $q \in Q, a \in \Sigma$ and $x \in X$, there is at most one $y \in X$, such that x occurs in $\delta_2(q, a, y)$.

We say that an SST *transduces* s to t , if by starting in q^- and following the unique path through the SST with index label s , you arrive at a state for which F is defined, and where the concatenation of the registers given by F produces t .

Definition 9 (Functional semantics of SSTs). An SST denotes a partial function $\llbracket S \rrbracket : \Sigma^* \rightarrow \Gamma^*$, such that $(s, t) \in \llbracket S \rrbracket$ precisely if S transduces s to t .

We can now define the constituent parts formally, building on the above intuition.

Consider an OFT, $\mathcal{T}$. We will now construct an equivalent SST as follows:

Let q^- be $\mathcal{PT}(\mathcal{T}, \varepsilon)$. Starting from q^- , let Q' be the set of all reachable reduced path trees. Let $Q = Q' \cup \{q^\perp\}$, where $q^\perp$ is the *reject state*.

Let X be the set of index labels used in any tree in Q' .

Let δ_1 be defined by $(q, a, q') \in \delta_1$ if and only if the tree obtained from stepping q with symbol a reduces to q' , or q doesn't step with symbol a and $q' = q^\perp$.

Let δ'_2 be defined by $(q, a, x, x_1 \dots x_n c)$ if and only if stepping q on symbol a produces the valuation update $\rho'(x) = \rho(x_1) \dots \rho(x_n)c$. Let δ_2 be δ'_2 , as well as (q, a, x, ε) for any state-transitions that aren't covered by δ'_2 .

Let F be the smallest set, such that for any $q \in Q'$ containing a node marked final, $(q, x_0 x_1 \dots x_n c) \in F$, where c is the annotated value on the final marking, and $x_0 \dots x_n$ are the index labels of the nodes along the path from the root node down to q , with $x_n = q$.

Definition 10 (OFT to SST determinization). Let $\mathcal{T}$ be an OFT, and $q^-, Q, X, \delta_1, \delta_2$ and F be defined as above.

We will call the SST $S = (X, Q, q^-, F, \delta_1, \delta_2)$ the *determinization* of $\mathcal{T}$.

We will call the register associated with the root node the *output register*, since any data written to this register is guaranteed to be output.

Figure 2.9 shows an example of an OFT, and how the SST constructed from the path trees on that OFT would look.

An SST constructed from an OFT in this fashion has some nice properties. Firstly, since the output register is only ever concatenated to, it is possible to output what is written to it in a streaming fashion.

Secondly, it's possible to implement an SST, such that it takes linear time to simulate, as shown in the following theorem.

Theorem 2. *Let S be an SST constructed via our method from an OFT $\mathcal{T}$.*

S can be implemented such that the time complexity of evaluating S on a string s is $O(mn)$, where m the number of states in $\mathcal{T}$, and n is the length of the string.

Proof. We can simulate the state machine itself in $O(n)$, as it corresponds to a DFA, so we will focus on the register updates.

If dynamic arrays with linear time concatenation are used for register operations, and identity updates are omitted, the overall transduction will be $O(nm)$ by the following argument: if an output character is moved, it is always concatenated to a register further up in the path tree. Since the reduced path trees of a given OFT have a maximal height of m , it will at most take a m moves before the character is either discarded or ends up in the output register. This means that all register updates in total will take a maximum of $O(mn)$ time, concluding the proof. $\square$

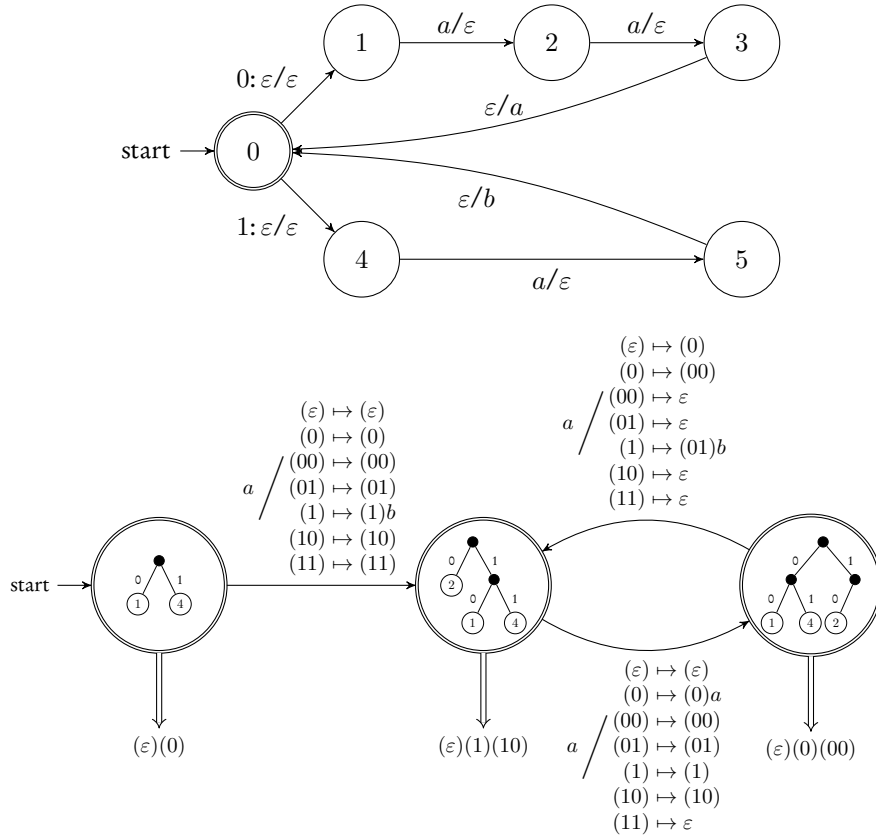


Figure 2.9: The figure at the top shows an OFT. The figure at the bottom shows the SST that results from applying our determinization algorithm to it, with the δ_2 register update shown on each edge, and F , the final output, emerging from the bottom of each final state.

2.6 Symbolic automata

For practical uses in string matching, automata produced in the fashion mentioned above will often end up containing bundles of similar transitions between states, as the transducers we have described require a separate transition for each possible input symbol. Thus, a transducer converting $. * \underline{a}$ to $. * \underline{b}$ in an 8-bit alphabet would require 256 different transitions, one for each possible value of “.”.

To avoid this, both the OFT and SST definitions above can be augmented to allow edges to have *predicates* as input labels, and *terms* indexed by the input symbols as output labels, an idea first explored by Watson [11].

For our uses, a predicate will be a set of character ranges, allowing the “.” mentioned above to be represented in a single transition.

We will not go into theoretical detail on this subject, as we have not been involved in this part of the project. For further theoretical details we refer to [12, 13].

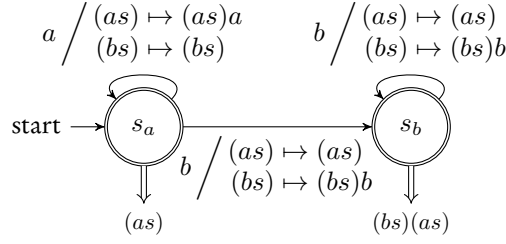


Figure 2.10: An SST that takes a string of the form $\underline{a}^* \underline{b}^*$ as input, and outputs the $\underline{bs}$ followed by the $\underline{as}$.

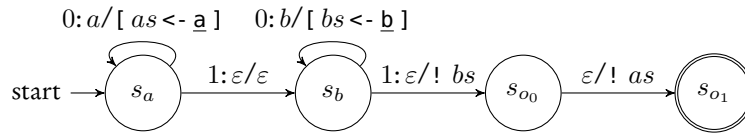


Figure 2.11: An OFAT that uses the register operations $[\cdot <- \cdot]$ and $! \cdot$ to replicate the functionality of Figure 2.10.

2.7 Action automata

So far we have considered the determinization of OFTs into SSTs, allowing for their corresponding single-valued rational relations to be realized in linear time. We will now consider how to take this further, delving into a world beyond rationality.

Our chief motivation for extending the model further was an observation, namely that we often ran into problems we could almost solve using our model, but were unable to due to a fundamental limitation in rational transduction: it is impossible to reorder non-trivial inputs for output. I.e., it is impossible to take an input consisting of any number of $\underline{as}$, followed by any number of $\underline{bs}$, and transduce that into the $\underline{bs}$ followed by the $\underline{as}$, turning $\underline{aabbba}$ into $\underline{bbbbaa}$.

While SSTs in general do allow for this sort of swapping, as shown in Figure 2.10, our approach is different, making use of two machines, one feeding its output into the other: a disambiguation oracle, and a so called action automaton. This construction was inspired by the Kleenex implementation, and will be used to define the semantics of Kleenex in Section 3.2.

To facilitate this we define a type of automaton, inspired by OFTs and SSTs, which we choose to call an *ordered finite action transducers* (OFAT). The idea is, rather than labeling edges with an output symbol, we instead label them with a function that performs a computation and potentially outputs an output string. An example of such an automaton can be seen in Figure 2.11. In this example we use the actions $[R <- \cdot]$, which appends a value to a register, and $! \cdot$, which outputs the contents of a register, to implement a transducer performing the aforementioned $\underline{a-b}$ swap.

We can formalize this into the following definition:

Definition 11 (Ordered finite action transducer). Let Σ and Γ be two given finite input and output alphabets.

We then define an *ordered finite action transducer* (OFAT) over Σ, Γ to be a structure, $\mathcal{T} = (Q, q^-, q^f, n, C, c_0, \delta_0, \delta_1)$, where

- Q is a finite set of states,
- $q^- \in Q$ is the initial state,
- $q^f \in Q$ is the final state,
- $n \in \mathbb{N}_0$ is an upper bound for the indices,
- C is the (possibly infinite) set of possible *computational environments*,
- $c_0 \in C$ is the initial environment,
- $\delta_1 : Q \times (\Sigma \cup \{\varepsilon\}) \times \{0, \dots, n\} \rightarrow Q$ is the *indexed transition relation*, such that if $(q, i, a) \in \text{dom}(E)$, then $\nexists a' \in \{\Sigma \cup \varepsilon\} : (q, i, a') \in \text{dom}(E)$,
- $\delta_2 : Q \times \{0, \dots, n\} \times C \rightarrow \Gamma^* \times C$ is the *environment update* function.

In a similar fashion to both OFTs and SSTs, the OFATs can also be generalized to allow for predicate input labels, giving a symbolic OFAT.

We introduce the notion of a computational environment, which initially starts as c_0 and can be updated on each transition. This concept is intentionally very abstract, in order to allow for arbitrary stateful computation in the model. For instance, an SST could be directly mapped into this model by letting C be $(\Gamma^*)^{|X|}$, the set of possible valuations of the registers, c_0 be $\varepsilon^{|X|}$, and assigning distinct indices to each outgoing edge of a state.

Using the natural extension of $\mathcal{T}$ transducing s to t from OFTs, we can define a functional semantics for this model.

Definition 12 (Functional semantics of OFATs). An OFAT denotes a partial function $\llbracket \mathcal{T} \rrbracket : \Sigma^* \rightarrow \Gamma^*$, such that $(s, t) \in \llbracket \mathcal{T} \rrbracket$ precisely if $\mathcal{T}$ transduces s to t .

The requirements put on the indices are strengthened for the very reason we alluded to prior to defining the OFAT model: We wish to split this *mother automaton* up, giving two *child automata*: One which acts as a *disambiguation oracle*, and one that acts as the *action automaton*, acting on the guidance of the oracle.

Observe how the strengthened requirements on the OFAT indices makes it possible to know the exact path used for a simulation of the machine, without ambiguity, simply based on the index label of the path. The idea is therefore to construct an SST that produces this *index code*, which the action automaton can act on.

With this in mind, consider the machine that arises from taking an OFAT, letting $C = \{\emptyset\}$, and letting $\delta_2(q, i, \emptyset) = (e(i), \emptyset)$, where $e(i)$ is i if there are multiple outgoing edges in q , and ε otherwise. The resulting automaton can be directly mapped into a $\Sigma, \{0, \dots, n\}$ OFT, which can be further determinized into an SST. The output produced by the resulting transducer is exactly the index code produced by simulating the mother automaton on the same input.

We now wish to construct the action automaton, a machine that rigidly follows the index code provided by the disambiguation oracle. We can construct this machine by taking the same mother automaton from before, and changing its input alphabet from Σ to $\{0, \dots, n\}$. This is done by setting the input label on each edge to its index label if the state it comes from has more than one outgoing edge, and ε otherwise.

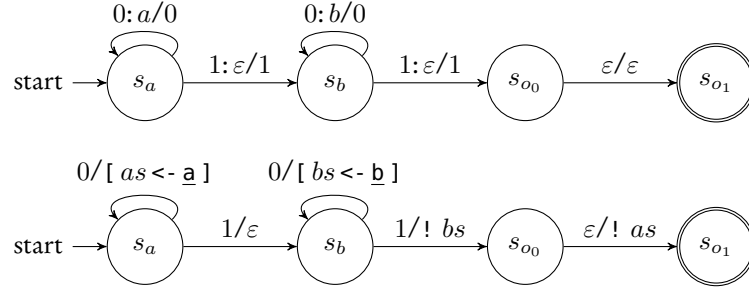


Figure 2.12: An illustration of the oracle FST and action automaton resulting from the OFAT in Figure 2.11.

An example of this separation of an OFAT into two child automata can be seen in Figure 2.12.

It is easy to convince oneself, that the combination of these two child automata is equivalent to the original mother automaton, since the exact transduction path is completely determined by its index label. Furthermore, since the disambiguation oracle can be compiled down to a linear-time SST, we can still do linear time transduction for fixed size automata. In Section 3.5 we explore this further, taking a look at a useful set of actions that provide such guarantees.

Chapter 3

The Kleenex Language

3.1 Overview

Kleenex[2] is a domain specific language for expressing string transformations. Parsing and extracting regular data is usually done with regular expressions, and they are well suited for it: they are succinct, efficient and are available in almost all languages. Kleenex is a superset of regular expressions, so the simplest Kleenex program you can write is simply a regular expression and some boilerplate code. For example:

```
main := /a*b/
```

This just runs the regular expression a^*b on its input and echoes it back if it matches. Below is an example with a few more features:

```
main := two_bs "d" main
      | /a/
```

```
two_bs := ~/bb/
```

In the example above we have two terms, which consists of two cases. The first case matches `two_bs`, i.e. `two bs`, but it does not output anything because of the `~` operator in front, which suppresses all output from that term. Notice that the Kleene star is outside the regular expression, as Kleenex supports some of the normal regex operators on the Kleenex term level. Next, it outputs the constant string `d`, as text in quotes denote constant output. It is also *recursive*, so if this case is chosen it starts from the same term all over again. The second case matches a single `a`, and since it is not recursive the program will stop if this alternative is chosen.

Running this Kleenex program, it matches an even number of `bs`, outputs a `d` for each, and ends by matching and outputting an `a`. Thus, if given `bbbba` it will output `dda`, while `bbba` will fail with a match error on the third `b`.

Kleenex is based on the theory in Section 2.3, which means we can define an ordering such that the path with the lexicographically least index label will be taken. We use this to ensure that Kleenex will always prefer the left-most matching alternative. It also guarantees that no output or actions from a case will be performed unless the case matches completely. As an example, consider the following program:

```
main := /foo/ "baz" /bar/
      | ~/fooba./
```

If given the input `foobam` it will not output anything, since the case ended up not matching. On the input `foobar` it will output `foobazbar`, as it prefers the first alternative when they both match.

A more comprehensive example that demonstrates the available language elements is given below.

```
// A Kleenex program starts with what we call a pipeline declaration.
// This one can be understood: First remove the comments,
// then gather the numbers at the bottom.
start: remComments >> gatherNumbers

// If no pipeline is specified, "main" is picked
// as the starting point.

// The most basic Kleenex term is matching. It matches
// the input against a regular expression, outputting it directly.
line := /[^\n]*\n/

// Often you don't want all the input turned into output.
// The ~ operator lets suppress the output otherwise produced
// by a term, in this case removing lines that start with "#",
// and preserving ones that don't.
// When there's ambiguity, the leftmost choice is always chosen.
commentLine := ~(#/ line) | line

// Recursion is allowed, but only in tail position. Here we
// terminate the recursion with "1", which consumes nothing and
// always succeeds.
remComments := commentLine remComments | 1

// We also allow regex operators like *, + and ? on terms:
thousandSepLines := (thousandSep /\n/ | line)*

// It's possible to output text without matching by using "...".
// In this case, we use it to insert thousands separators into a number.
thousandSep := digit{1,3} ("," digit{3})* /\n/
digit := /[0-9]/

// We also allow for more complicated operations. We call these 'actions'.
// reg@term runs the term as normal, but all output it would produce is
// stored in the register named reg.
// [ ... += ... ] allows you to append things to a register, both contents
// of other registers, as well as string constants.
// !reg outputs the contents of a register.
gatherNumbers :=
  (num@thousandSep [ numbers += num ] | line)*
  !numbers
```

3.2 Core language

In order to define the semantics of the language, we will first introduce a simpler core language. This language contains the bare minimum required to make Kleenex work.

In Section 3.3 we will cover the full language, and its desugaring into the core language.

We will define the Kleenex core semantics in terms of the OFAT model introduced in Section 2.7. First we need some auxiliary definitions.

Definition 13 (Kleenex computational environment). The *Kleenex computational environment* $C_K = X^* \times (\Gamma^*)^{|X|} \times X$, where X is the set of *register variables*.

For a $c = (S, Y, o) \in C_K$,

- S is a stack of pushed registers,
- Y is the current contents of the registers,
- o is the current *output register*.

We say that $c_0 = (\text{nil}, \varepsilon^{|X|}, r_0)$ is the *initial Kleenex environment*.

The purposes of these values will become clear once we introduce the actions of Kleenex.

Definition 14 (Action). An *action* is a function $b \in C_K \rightarrow \Gamma^* \times C_K$, where Σ and Γ are the sets of input and output symbols.

We define ε_A to be the *empty action*, $c \mapsto (\varepsilon, c)$.

Definition 15 (Core action). In the core language, we have the following actions defined:

$$\begin{aligned} \text{output } c &= (S, \{\dots, o, \dots\}, o) \mapsto (S, \{\dots, o ++ c, \dots\}, o) \\ \text{setreg } R \ Z &= (S, Y, o) \mapsto (S, \text{update}(Y, R, Z), o) \\ \text{push } R &= (S, Y, o) \mapsto (o : S, Y, r) \\ \text{pop} &= (r : S, Y, o) \mapsto (S, Y, r) \end{aligned}$$

where $c \in \Gamma$ is an output character, $R \in X$ is a register, $Z \in (X \cup \Gamma^*)^*$ is a string consisting of distinct register variables and output strings, and $\text{update}(Y, R, Z)$ is a register update function that sets the R register to the concatenation of Z , and clears all registers $r' \in Z$ where $r' \neq R$. The operators $(++)$ and $(:)$ are used to denote the list operations append and cons, respectively.

We will call these actions the *core actions*.

The simplest action is the output c action, which outputs the constant c . The next action is a $\text{setreg } R \ Z$ action, mirroring the register updates seen in SSTs. Note that the clearing semantics of this operation prevents copying of data.

The final two actions $\text{push } R$ and pop change which register is used as the output register in output c .

With this in mind, we will now give the definition of the core syntax for Kleenex.

Definition 16 (Kleenex core terms and syntax). Let Σ, Γ be input and output alphabets, respectively.

A *Kleenex core term* is a *term* generated by the grammar

$$t_i ::= 1 \mid N \mid \langle s \rangle \mid a \mid t_i \mid t_i \mid t_j \cdot t_i$$

where $i, j \in \mathbb{N}$, $i > j$, $s \in \Sigma$, a is a core action, and N is a *nonterminal* of order at most i .

We say that i is the *order* of the term t_i .

A Kleenex program is a non-empty list of *declarations*, $p = d_1 \dots d_n$, each of the form $N_i := t_k$, where t_k is a k -order term.

We call the nonterminal N_i the *identifier* of the definition, and say that it has *order* k , the order of its associated term.

This defines a number of families of terms and declarations, grouped by their order. While slightly cumbersome, the order serves to limit the recursion of terms, thus enforcing that the grammars they produce are regular. This limitation will be crucial in the proof of Theorem 3 below. The restrictions allow mutual tail-recursion between definitions of the same order, but requires that any other use of nonterminals are of a lower order.

The core language contains a small number of constructs:

- 1, which does nothing,
- $\langle s \rangle$, which matches string s from input without giving any output,
- a , any action defined previously,
- $t_1 \mid t_2$, alternation that chooses the left branch if possible,
- $t_1 \cdot t_2$, concatenation; first evaluate t_1 , then t_2 .

To complete our definition we introduce a notion of well-formedness on Kleenex programs.

Definition 17 (Well-formedness of Kleenex programs). We say that a list of Kleenex declarations is *well-formed* if

- Identifiers are declared at most once
- No undeclared identifiers appear in the declarations
- The program contains no recursion without input consumption (i.e. *problematic paths*)

From this point we will assume that all Kleenex programs are well-formed.

We will now define right-regular grammars with actions, an extension of right-regular grammars introduced in a rejected version of [2], and show that Kleenex corresponds to such a grammar.

Definition 18 (Grammar). We define a *grammar with actions* to be an ordered list of *production rules* of the form

$$N \longrightarrow x_1 x_2 \dots x_n$$

where we call N a *nonterminal*, and where each x_i either is either a nonterminal from the grammar, or c/a , with $c \in \Sigma \cup \{\varepsilon\}$ and a being an action.

We will give c/a the intuitive meaning "read c , and perform action a ".

We use $G_1 ++ G_2$ to denote ordered concatenation of two grammars, producing a new grammar.

Definition 19 (Right-regular grammar with actions). We say that a grammar with actions is *right-regular* if all the rules take one of the following forms:

$$\begin{aligned} N &\longrightarrow c/a \\ N &\longrightarrow c/a N' \end{aligned}$$

where $c \in \Sigma \cup \{\varepsilon\}$ is an input symbol, b is an action and N, N' are nonterminals.

Theorem 3. *Any well-formed Kleenex core program has an equivalent right-regular grammar with actions.*

Proof. We will rewrite each declaration in the program to a right-regular grammar, only referencing nonterminals of lower or equal order. We do so inductively on the order of the declarations.

As the base case, consider declarations of order 1. Using structural induction on $N := t_1$, we have 6 possible cases:

$N := \mathbf{1}$ can be directly rewritten to $N \longrightarrow \varepsilon/\varepsilon_A$

$N := \langle c \rangle$ can be directly rewritten to $N \longrightarrow c/\varepsilon_A$

$N := a$ can be directly rewritten to $N \longrightarrow \varepsilon/a$

$N := t \mid t'$:

$N := t$ and $N := t'$ can be rewritten by inner induction, producing G_1 and G_2 respectively. The grammar $G_1 ++ G_2$ is equivalent and preserves ordering of terms¹.

$N := t \cdot t'$ is not possible, since no t of lower order exists.

$N := N'$ can be directly rewritten to $N \longrightarrow \varepsilon/\varepsilon_A N'$, since we know that N' must have order 1.

This concludes the inner structural induction, and the constructed grammar is clearly right-regular, thus also concluding the base case of the outer induction.

For the induction step, consider declarations of order $k > 1$. Again we use structural induction on $N := t_k$ to construct the rewritten grammar. All cases except for $N := t \cdot t'$ are unchanged, and therefore omitted.

¹In the sense, that the left to right order on Kleenex core terms is converted into the same top to bottom order on their corresponding productions.

<pre> main := odd ~/a/ even ~/a/ odd := ~/aa/ "bb" odd "c" even := ~/a/ "c" even "b" </pre>	<pre> $N_{\text{main}} := N_{\text{odd}} \langle a \rangle \mid N_{\text{even}} \langle a \rangle$ $N_{\text{odd}} := \langle a \rangle \langle a \rangle (\text{output } b) (\text{output } b) N_{\text{odd}}$ output c $N_{\text{even}} := \langle a \rangle (\text{output } c) N_{\text{even}} \mid (\text{output } b) 1$ </pre>
---	--

Figure 3.1: To the left is a Kleenex program in the surface syntax and on the right is the desugared version.

Consider $N := t.t'$. By outer induction we can construct a right-regular grammar, G_1 , for $N := t$, using fresh names for all nonterminals except for N . By inner induction, we can construct a right-regular grammar, G_2 , for $N' := t'$. We now define a new grammar, G'_1 , which is G_1 with all rules of the form $M \rightarrow c/a$ replaced by $M \rightarrow c/a N'$. $G'_1 \uparrow\uparrow G_2$ will now constitute a right-regular grammar for $N := t.t'$.

This concludes both inductions, proving the theorem. $\square$

Important to note about this transformation is, that it preserves order in the sense that the rule resulting from a left-hand branch in an alternation comes earlier in the grammar than the rule resulting from a right-hand branch. This is important in ensuring the correct disambiguation.

We will now construct the OFAT that defines the semantics of the language.

Let p be a Kleenex core program, and assume WLOG that it is in right-regular form. We let the set of states, Q , be the set of nonterminals of p , as well as q^f , a final state. The transition relation, δ_1 , and environment update function, δ_2 , are defined based on the rules in the program.

- For each rule $N \rightarrow a/b N'$, $(N, \hat{a}, i, N') \in \delta_1$ and $\forall c \in C_K (N, i, c, b(c)) \in \delta_2$
- For each rule $N \rightarrow a/b$, $(N, \hat{a}, i, q^f) \in \delta_1$ and $\forall c \in C_K (N, i, c, b(c)) \in \delta_2$

where i is the number of previous rules we have seen for N , and $\hat{a}$ is the natural extension of the action to produce output.

We can now define an OFAT, $\mathcal{T}_p = (Q, N_0, q^f, n, C^K, c_0^K, \delta_0, \delta_1)$, where N_0 is the starting production, and n the maximal number of rules for any nonterminal.

Definition 20 (Kleenex core semantics). Let p be a Kleenex core program with associated OFAT, $\mathcal{T}_p$ as described above.

The program p denotes a partial function $\llbracket p \rrbracket : \Sigma^* \rightarrow \Gamma^*$ given by

$$\llbracket p \rrbracket = \llbracket \mathcal{T}_p \rrbracket$$

3.3 The Kleenex language

Using the semantics above, we can now give the rest of the Kleenex language as syntactic sugar on top of the core language. First we will introduce the full Kleenex actions.

$$\begin{aligned}
\mathcal{D}_a \llbracket "s_1 \dots s_n" \rrbracket &= (\text{output } s_1) \cdots (\text{output } s_n) \\
\mathcal{D}_a \llbracket R @ t \rrbracket &= \text{push } R \cdot \mathcal{D}_t \llbracket t \rrbracket \cdot \text{pop} \\
\mathcal{D}_a \llbracket !R \rrbracket &= \text{setreg } o \text{ } (o ++ R) \quad \text{where } o \text{ is the current output register} \\
\mathcal{D}_a \llbracket [R <- Z] \rrbracket &= \text{setreg } R \text{ } Z \\
\mathcal{D}_a \llbracket [R += Z] \rrbracket &= \text{setreg } R \text{ } (R Z)
\end{aligned}$$

Figure 3.2: The desugaring function for converting Kleenex actions into core actions.

Definition 21 (Kleenex actions). Let Σ, Γ be input and output alphabets. A *Kleenex action* of order i , is a term generated by the grammar

$$a_i ::= "s" \mid R @ t_j \mid !R \mid [R <- (R \mid "s")^*] \mid [R += (R \mid "s")^*]$$

where R is a register and $s \in \Sigma$, and $j < i$.

" s " outputs a constant string, $R @ t$ captures the result of t into register R , $[R <- Z]$ sets register R to the concatenation of the registers and constants in Z , and $[R += Z]$ appends the contents of the registers and constants in Z to R .

The desugaring function that translates these actions to the core language can be seen in Figure 3.2.

Definition 22 (Kleenex terms). Let Σ, Γ be input and output alphabets. A *Kleenex term* of order i is a term generated by the grammar

$$\begin{aligned}
t_i ::= & 1 \mid a_i \mid t_i \mid t_j \cdot t_i \mid /E/ \mid \sim t_j \mid t_j^* \mid t_j^+ \mid t_i? \\
& \mid t_j \{n\} \mid t_j \{n, \} \mid t_j \{, m\} \mid t_j \{n, m\}
\end{aligned}$$

where $s \in \Gamma^*$, $E \in RE_{\text{Ext}}(\Sigma)$, $j < i$.

Most of the new constructs added in the full language are based on the regular expression operators, and their semantics match as well. $/E/$ is the match operator, which matches a regular expression E , and outputs the matching value. $\sim t$ suppresses the output of a given term.

The desugaring function for terms can be seen in Figure 3.3, which makes use of the functions in Figure 3.4 to desugar regular expressions, and Figure 3.5 to desugar suppressed terms.

Definition 23 (Kleenex desugaring). Let $p = d_1 d_2 \dots d_n$ be a Kleenex program, and let d'_i the declaration d_i with $\mathcal{D}_t \llbracket \cdot \rrbracket$ applied to its term.

We define the desugaring function on programs,

$$\mathcal{D}_p \llbracket p \rrbracket = d'_1 d'_2 \dots d'_n$$

Theorem 4. For all p , $\mathcal{D}_p \llbracket p \rrbracket$ is well-defined.

$$\begin{aligned}
\mathcal{D}_t[1] &= 1 \\
\mathcal{D}_t[N] &= N \\
\mathcal{D}_t[/E/] &= \mathcal{D}_t[\mathcal{D}_e[E]] \\
\mathcal{D}_t[\sim t] &= \mathcal{D}_t[\mathcal{D}_t[t]] \\
\mathcal{D}_t[t*] &= N' \quad \text{where } N' := \mathcal{D}_t[t] \cdot N' \mid 1 \\
\mathcal{D}_t[t+] &= \mathcal{D}_t[t] \cdot \mathcal{D}_t[t*] \\
\mathcal{D}_t[t?] &= \mathcal{D}_t[t] \mid 1 \\
\mathcal{D}_t[t_1 \cdot t_2] &= \mathcal{D}_t[t_1] \cdot \mathcal{D}_t[t_2] \\
\mathcal{D}_t[t_1 \mid t_2] &= \mathcal{D}_t[t_1] \mid \mathcal{D}_t[t_2] \\
\mathcal{D}_t[t\{n\}] &= \underbrace{\mathcal{D}_t[t] \cdot \dots \cdot \mathcal{D}_t[t]}_n \\
\mathcal{D}_t[t\{n, \}] &= \mathcal{D}_t[t\{n\}] \cdot \mathcal{D}_t[t*] \\
\mathcal{D}_t[t\{, m\}] &= \mathcal{D}_t[(t?)\{m\}] \\
\mathcal{D}_t[t\{n, m\}] &= \mathcal{D}_t[t\{n\}] \cdot \mathcal{D}_t[t\{, k\}] \quad \text{where } k = m - n \\
\mathcal{D}_t[a] &= \mathcal{D}_a[a]
\end{aligned}$$

Figure 3.3: The desugaring function for converting Kleenex terms into core terms.

$$\begin{aligned}
\mathcal{D}_e[1] &= 1 \\
\mathcal{D}_e[a] &= \langle a \rangle "a" \quad \text{for } a \in \Sigma \\
\mathcal{D}_e[E_1 E_2] &= \mathcal{D}_e[E_1] \cdot \mathcal{D}_e[E_2] \\
\mathcal{D}_e[E_1 \mid E_2] &= \mathcal{D}_e[E_1] \mid \mathcal{D}_e[E_2] \\
\mathcal{D}_e[E^*] &= \mathcal{D}_e[E]^* \\
\mathcal{D}_e[E^+] &= \mathcal{D}_e[E]^+ \\
\mathcal{D}_e[E^?] &= \mathcal{D}_e[E]^? \\
\mathcal{D}_e[E\{n\}] &= \mathcal{D}_e[E]\{n\} \\
\mathcal{D}_e[E\{, m\}] &= \mathcal{D}_e[E]\{, m\} \\
\mathcal{D}_e[E\{n, \}] &= \mathcal{D}_e[E]\{n, \} \\
\mathcal{D}_e[E\{n, m\}] &= \mathcal{D}_e[E]\{n, m\} \\
\mathcal{D}_e[[r]] &= \langle c_1 \rangle "c_1" \mid \langle c_2 \rangle "c_2" \mid \dots \quad \text{for } c_i \in r \\
\mathcal{D}_e[[^r r]] &= \langle c_1 \rangle "c_1" \mid \langle c_2 \rangle "c_2" \mid \dots \quad \text{for } c_i \in \Sigma - r
\end{aligned}$$

Figure 3.4: The desugaring function for converting regular expressions into Kleenex terms.

$$\begin{aligned}
\mathcal{D}_\sim[\mathbf{1}] &= 1 \\
\mathcal{D}_\sim[N] &= N_\sim && \text{where } N_\sim := \mathcal{D}_\sim[t] \\
\mathcal{D}_\sim[\langle s \rangle] &= \langle s \rangle \\
\mathcal{D}_\sim[t_1 \mid t_2] &= \mathcal{D}_\sim[t_1] \mid \mathcal{D}_\sim[t_2] \\
\mathcal{D}_\sim[t_1 \cdot t_2] &= \mathcal{D}_\sim[t_1] \cdot \mathcal{D}_\sim[t_2] \\
\mathcal{D}_\sim[\text{push } x \cdot t \cdot \text{pop}] &= \text{push } x \cdot t \cdot \text{pop} \\
\mathcal{D}_\sim[\text{output } c] &= 1 \\
\mathcal{D}_\sim[a] &= a && \text{for other actions}
\end{aligned}$$

Figure 3.5: The desugaring function for desugaring suppressed core terms into core terms.

Proof. Since we have specified $\mathcal{D}_p[p]$ using a number of re-write rules that cover all syntactical cases, we will argue for progress, i.e., that we can never reach the same state again after the rewrite.

The $\mathcal{D}_e[\cdot]$ function is only used when the match operator is encountered, and rewrites it to a term that does not use the match operator.

The one recursive case in $\mathcal{D}_a[\cdot]$ can at most be reached a finite number of times, since the order of t is lower than $R @ t$ by definition.

In the $\mathcal{D}_\sim[\cdot]$ function, the nonterminal only needs to be rewritten once, allowing future desugaring to skip that step. Pushes and pops will always come in this pattern, since they can only be generated by desugaring an $R @ t$. All other cases must clearly progress.

All cases in $\mathcal{D}_t[\cdot]$ must progress, since they reduce the problem to a structurally simpler one, or have been covered by the previous arguments. $\square$

Definition 24 (Kleenex semantics). Let p be a Kleenex program.

The program p denotes a partial function $\llbracket p \rrbracket : \Sigma^* \rightharpoonup \Gamma^*$ given by

$$\llbracket p \rrbracket = \llbracket \mathcal{D}_p[p] \rrbracket$$

where $\mathcal{D}_p[\cdot]$ is the function that applies $\mathcal{D}_t[\cdot]$ on the terms of all definitions in a program.

3.4 Expressivity of Kleenex

Alur and Deshmukh [14] describe a nondeterministic version of streaming string transducers that both with ε transitions (ε -NSST) and without (NSST). These models are the obvious extension of the SST model we have described, to allow for nondeterminism.

Theorem 5. Let p be a Kleenex program.

There exists an NSST S , such that

$$\llbracket S \rrbracket = \llbracket p \rrbracket$$

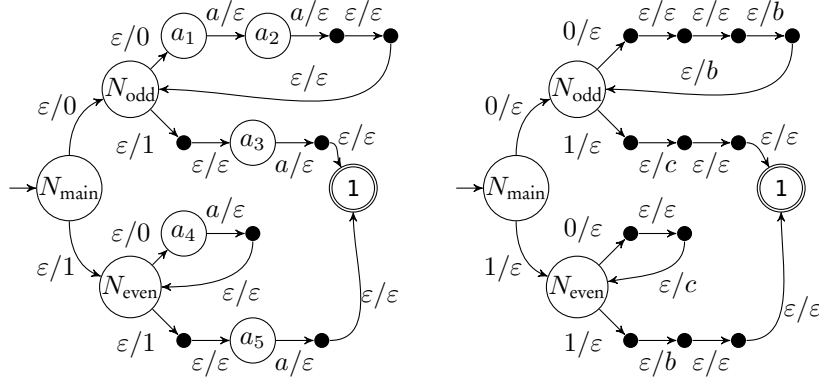


Figure 3.6: The oracle and action automata corresponding to the grammar from Figure 3.1.

Proof. We can define an alternate semantics for Kleenex by omitting the stack actions push R and pop from the core language. Due to $R @ t_j$ requiring the inner term to be of lower order, we can still recursively desugar the complete Kleenex language to the core language.

Doing so would allow us to reduce the required computational environment to only require the register contents. Due to the clearing semantics of the register updates, and the limited actions allowed, the resulting OFAT maps directly onto an ε -NSST that outputs the output register in the final state.

General ε -NSST are more expressive than NSST, however an ε -loop-free ε -NSST can be transformed into an NSST [14]. The well-formedness requirement that the program contain no problematic paths ensures that this will be the case, giving us an NSST. $\square$

Conjecture 1. *Let p be a Kleenex program.*

There exists an SST S , such that

$$\llbracket S \rrbracket = \llbracket p \rrbracket$$

3.5 Time complexity bounds

If all actions in a program are constants, we can prove a very nice time bound on the program.

Theorem 6. *If p is a Kleenex program containing only actions of the form " a ", $a \in \Gamma^*$ and $s \in \Sigma^*$, $\llbracket p \rrbracket(s)$ can be computed in $O(nm)$ time, where n is the size of s and m is the number of states of the OFAT generated by p .*

Proof. If we construct an OFAT from p , the resulting machine will contain only actions that do not manipulate state and output single characters. Because of this, the machine can be directly translated into an OFT with the output actions split up across multiple edges as needed. This machine can then be determinized to a SST, giving the time bound by Theorem 2. $\square$

We can go one step further, and prove a time bound for arbitrary Kleenex actions.

Theorem 7. *Let p be a Kleenex program using r registers, which generates an OFAT with m states. Furthermore, let $s \in \Sigma^*$ be an input string of length n .*

$\llbracket p \rrbracket(s)$ can be computed in $O(nm^2(1+r))$ time.

Proof. Take the OFAT constructed for p and split it into oracle and action automata. By Theorem 2, the oracle SST can be simulated in $O(nm)$. Furthermore, the SST can produce an index label of at most $O(m)$ symbols for each character consumed, giving the action automaton potentially $O(nm)$ data to process.

For each character the action automaton receives, it will potentially have to do $O(m)$ actions. The push and pop actions can be done in $O(1)$ by maintaining a stack of pointers, and output literals are by definition $O(1)$. The register updates can touch at most r registers. Since the semantics clear the registers from which the data is taken, registers can be implemented as doubly-linked lists of strings, making it possible to concatenate two registers in constant time. This allows us to do a register update in $O(r)$ time.

Since at most $O(m)$ new data can be introduced on each input symbol, final output can be done in $O(nm^2)$.

This gives the action automaton a $O(nm^2(1+r))$ time complexity, proving the bound. $\square$

This proves a linear transduction time for fixed automata with any Kleenex actions. For programs using reasonably small numbers of registers, this essentially gives an nm^2 bound on the time complexity. Furthermore, for most programs it seems likely to us, that the $m^2(1+r)$ factor will be significantly lower, as it reflects the absolute worst case.

Chapter 4

Implementation

In this section we will introduce our work on `repg`¹, a compiler for the Kleenex language implemented by the KMC group[2]. This thesis contributed the following features to `repg` (and the Kleenex language):

- The pipeline construct and its runtime implementation
- The Kleenex action extension
- Regex operators on the Kleenex term level
- A number of optimizations to the compilation process and runtime

In this chapter we will describe part of the `repg` implementation, focusing on the features we have contributed.

4.1 Compilation

The original compilation path creates fused SSTs without actions, and was contributed by other project members. The section below will focus on our contribution, namely the compilation path which includes actions. We note that the action compilation path builds upon the infrastructure from the original compilation path, especially type classes, the SST construction step, optimizations and C code generation.

The compilation process is split into multiple phases, which are illustrated in Figure 4.1, and each phase is described in detail below.

Conversion to μ -terms

Lexical analysis and parsing is implemented using Parsec, a parser combinator library for Haskell[15]. The regular expression parsing is handled by the library `regexps-syntax`². This produces an AST of the form seen in Figure 4.2.

¹<https://github.com/diku-kmc/repg>

²<http://github.com/diku-kmc/regexps-syntax>

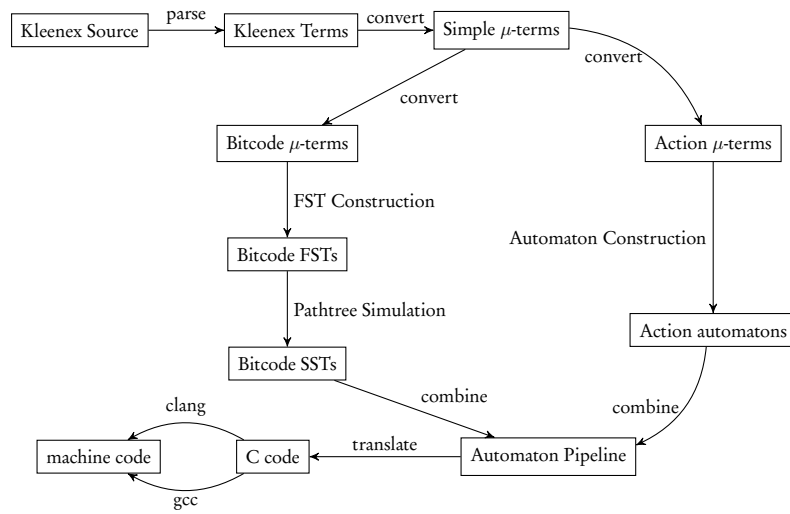


Figure 4.1: Kleenex compilation flowchart

```

-- | A Kleenex program is a list of assignments.
data Kleenex = Kleenex [Identifier] [KleenexAssignment]

-- | Assigns the term to the name.
data KleenexAssignment = HA (Identifier, KleenexTerm)

-- | The terms describe how regexps are mapped to strings.
data KleenexTerm = Constant ByteString
  | RE Regex
  | Var Identifier
  | Seq KleenexTerm KleenexTerm
  | Sum KleenexTerm KleenexTerm
  | Star KleenexTerm
  | Plus KleenexTerm
  | Question KleenexTerm
  | Range (Maybe Int) (Maybe Int) KleenexTerm
  | Ignore KleenexTerm
  | Action KleenexAction KleenexTerm
  | One

```

Figure 4.2: The data type representing the Kleenex AST

While it would be possible to directly construct the two automata from here, it becomes much easier after some preprocessing. Thus we transform the Kleenex type into μ -terms in two steps.³

First the terms are converted into `SimpleMu`. In this step the Kleenex terms used by the program are inlined as much as possible, and when recursive variables are encountered an explicit loop constructor is created. The variables are encoded as de Bruijn indices.

In the second step they are converted from the `SimpleMu` representation to two type-specialized instances of the polymorphic `Mu` data type, namely `BitcodeMu`

³ μ -terms, also known as μ -recursive expressions, is an alternative abstraction used to fill the role of the right-regular grammars from Section 3.2.

```

data Mu pred func a =
  Var a -- ^ Recursion variable.
  | Loop (a -> Mu pred func a) -- ^ Fixed point.
  | Alt (Mu pred func a) (Mu pred func a) -- ^ Alternation.
  | Action func (Mu pred func a) -- ^ Perform a manual register action
  | RW pred func (Mu pred func a) -- ^ Read a symbol matching the given
                                   -- predicate and write an output
                                   -- indexed by the concrete input
                                   -- symbol.
  | W (Rng func) (Mu pred func a) -- ^ Write a constant.
  | Seq (Mu pred func a) (Mu pred func a) -- ^ Sequence
  | Accept -- ^ Accept

```

Figure 4.3: Data type for recursive μ -terms

and ActionMu. This is necessary because the two automata need to handle certain constructs differently, e.g. when outputting a constant string the oracle transducer should do nothing because the action is unambiguous, while the action automaton needs to actually output the string. The Mu data type is given in Figure 4.3.

Two things happen during this transformation. First, the program is converted to a HOAS representation by converting the de Bruijn indexed variables to Haskell variables, and by converting the μ -terms as Haskell functions. Second, the regular expressions are converted to the same Mu data type and inlined, erasing the difference between constructs on the term level and on the regular expression level.

Automaton constructions

The automata are all implemented as the following two data types:

```

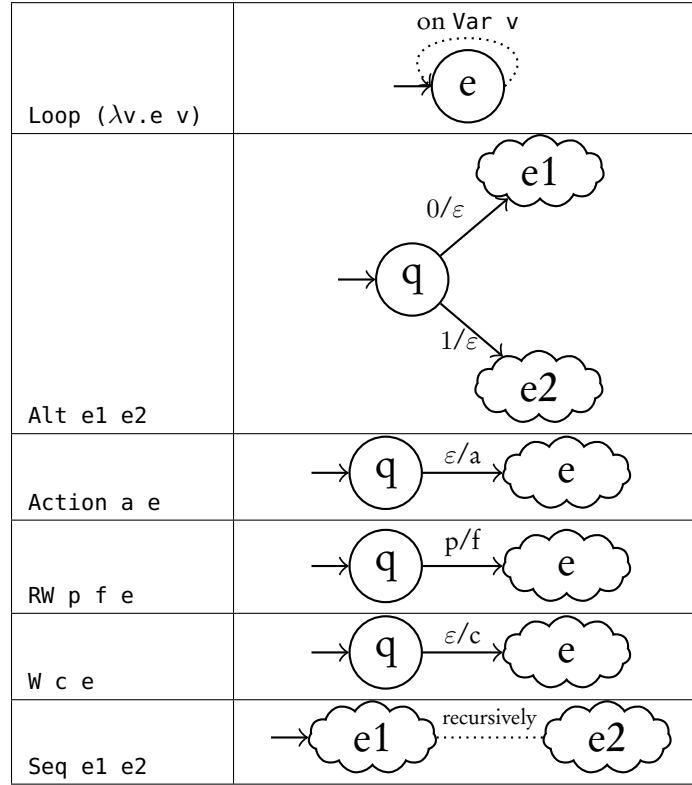
data FST st pred func =
  FST
  { fstS :: S.Set st -- ^ State set
  , fstE :: OrderedEdgeSet st pred func -- ^ Symbolic transition relation
  , fstI :: st -- ^ Initial state
  , fstF :: S.Set st -- ^ Final states with final output
  }

data SST st pred func var =
  SST
  { sstS :: S.Set st -- ^ State set
  , sstE :: EdgeSet st pred func var -- ^ Symbolic transition relation
  , sstI :: st -- ^ Initial state
  , sstF :: M.Map st (UpdateString var (Rng func)) -- ^ Final states with final output
  }

```

Here st is the type of states, pred is the type of predicates used for matching input, func needs to be an instance of the type class Function and is used for edges that needs to copy their input to output, and finally var are the type of register variables. This use of parametric polymorphism allows these two data types to model all the automata used in the compilation.

The FST data type models OFTs, but it is not called OFT because the indices are not explicit in its type. The SST type actually models both SSTs and action automata, as its edges has the following type:

Figure 4.4: Recursive conversion of μ -terms to a OFT

```

type EdgeSet st pred func var = M.Map st [([pred], EdgeAction var func, st)]
type EdgeAction var func = RegisterUpdate var func
                             :+: ActionExpr var (Dom func) (Rng func)

```

Note that `:+:` is an infix type operator which is isomorphic to the `Either` type. The SSTs are simply restricted to not use `EdgeActions` of the `ActionExpr` type, as they model the Kleenex actions from Section 3.2.

Both types of automata are symbolic in the sense described in Section 2.6, allowing for a character class to be represented as a single edge. They are represented by a data type called `RangeSet`, which is a sorted list of pairs representing character ranges.

The bytecode μ -terms are converted into a non-deterministic OFT using a simple method analogous to Thompson’s construction for transforming regular expressions into NFAs [4]. A graphical representation of this conversion can be seen in Figure 4.4.

The oracle OFT and action automata generated from the odd-even example from Figure 3.1 can be seen in Figure 4.5.

The bytecode SSTs are generated from the OFTs by simulation with path trees, as described in Section 2.5. This is done by defining a `Closure` monad, which models concurrent programs with shared state executed in round-robin. We will not go into details about this, as it is outside the scope of this thesis.

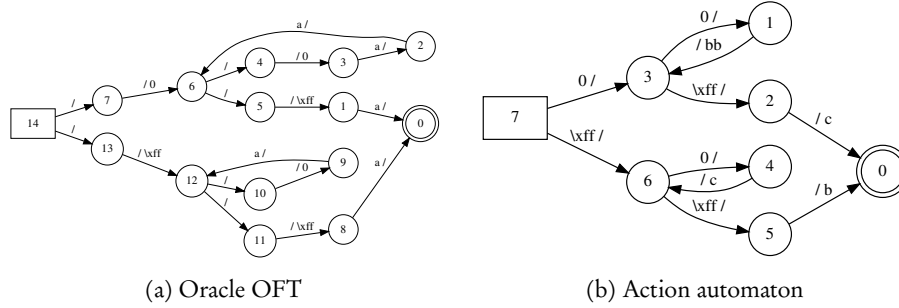


Figure 4.5: Visualizations of `odd_even` from Figure 3.1, created with `repg`'s visualization mode. The rectangular node is the start state, and the node with double circle is the final state. Blank spots denote ε and ε_A as appropriate.

As can also be seen with NFA determinization, the determinization of an OFT into the corresponding SST can cause a blowup in the number of states. This leads to large compile times, with some programs with large amounts of overlap being unable to compile.

Since there is no nondeterminism in the action automata, they can be constructed directly from the action μ -terms, completely analogously to the oracle OFT construction above.

Code generation

While the compiler is structured to allow multiple backends for code generation, currently only a C backend exists. This section will focus on the translation to the intermediate language, as the translation from this language to C is trivial (and tedious).

The OFAT pipeline is converted to a list of programs in this intermediate language, which consists of `goto`, conditionals, simple expressions, buffer operations and tables. The full definition can be seen in Figure A.1. Each program is generated from the OFAT by a simple recursive algorithm, which traverses each state in the SST exactly once.

A state is translated into a code block, i.e. a sequence of instructions. This block consists of the code blocks generated by all transitions from the state, and possibly a finalizing code block if it is a final state.

A transition is translated into two code blocks, namely a test block and an action block. The test block is simply an `if`-statement with the conditions needed for the transition to be taken, while the action block contains code for consuming the right amount of input, performing the actions and buffer operations associated with the transition, and then transferring control to the transition's end state.

Symbolic transitions are tabulated, i.e. a table is generated which maps an input symbol to a corresponding output character. This is used when converting a bytecode to the character it represents, among other places.

An important detail is that the SST model assumes that all buffer updates happen simultaneously, but this is not true in the runtime. To see why this can be a problem, consider a transition which updates two registers a and b , such that a is appended to b and a is cleared. If the clearing happens first we get the wrong result, since the buffers are mutable.

The solution is to do a topological sort of the buffer operations, based on the dependencies of the operations. More precisely, an update u_0 should happen before an update u_1 if u_0 uses the register that u_1 updates. This only works as long as there are no cyclic dependencies, but luckily that is guaranteed by the SST construction.

```

ll_2: if (!readnext(1, 1))
{
    goto fail1;
}
if (((avail >= 1) && (((0 <= next[0]) && (next[0] <= 9))
|| ((11 <= next[0]) && (next[0] <= 255))) && 1))
{
    outputarray(const_1_0,8);
    outputconst(tbl1[0][next[0]],8);
    consume(1);
    goto ll_2;
}
if (((avail >= 1) && ((next[0] == 10) && 1)))
{
    outputarray(const_1_4,8);
    consume(1);
    goto ll_4;
}
goto fail1;

```

Figure 4.6: Example of some code produced by the repg compiler. `next` is a pointer to the next byte in the input stream. `tbl1` contains a tabulated function, and `const_1_4` contains a constant output string.

4.2 Runtime

In order for the generated code to be run, it needs to interface with a runtime that can do input, output, and has dynamically-sized registers. While these requirements are pretty simple, the runtime needs to be carefully designed and optimized to get good performance.

The runtime is written in C. This is both for performance reasons, and for simplicity when interfacing with the C code generated from the SST. The downside is that it is harder to write correct C than Haskell. There can also be security implications if the streaming data is controlled by a malicious user and a memory corruption vulnerability is present.

The registers are implemented as dynamic arrays in the following struct:

```

typedef struct {
    buffer_unit_t *data;
    size_t size;
    size_t bitpos;
} buffer_t;

```

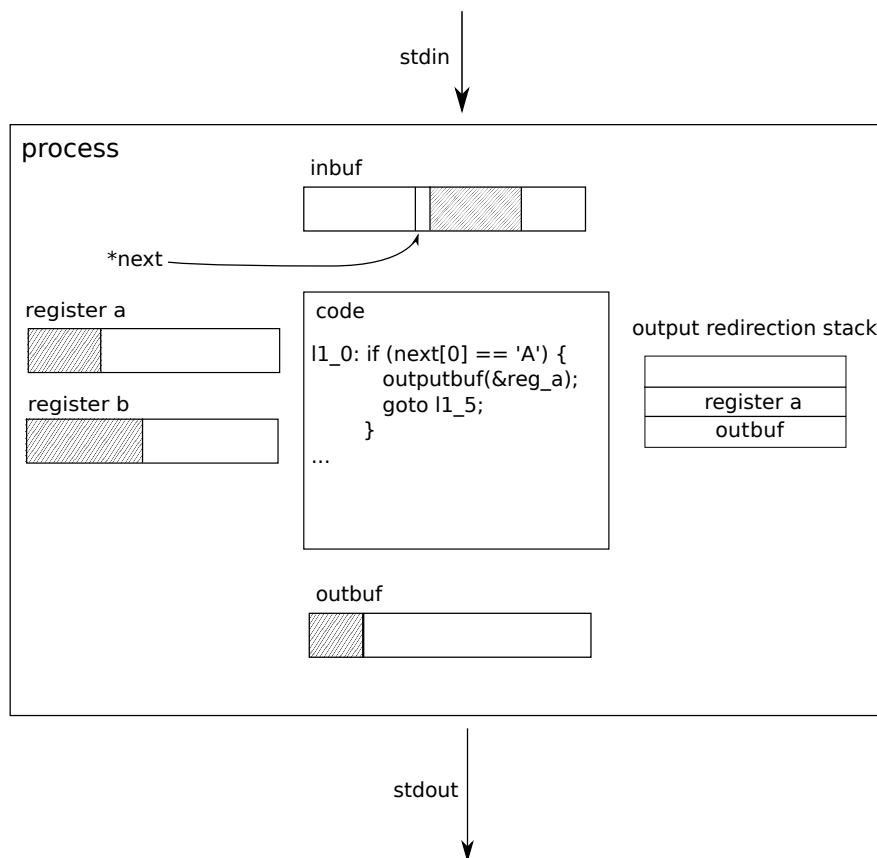


Figure 4.7: Diagram of the Kleenex runtime. The hatched areas are data waiting to be processed. `outputbuf` will always write to the pointer on top of the output redirection stack. When running the split automata, two runtimes are instantiated and connected with Unix pipes, and when using the pipeline extension any number of runtimes can be used.

Here, `data` is a pointer to a heap-allocated buffer, `size` is the current size of the buffer, and `bitpos` is the amount of bits the buffer currently contains.

An alternative design would be to use linked lists as described in Section 3.5. This has not been investigated, but it might not give much performance in practice despite having a better asymptotic running time, because the registers are usually cleared often in the programs we have examined.

The buffers support the following operations:

reset(buf)

Clear `buf`. This is implemented by setting `bitpos` to zero and clearing the first byte of data. In an earlier version of the runtime it cleared the entire register, but that ended up being a big performance bottleneck.

resize(buf, shift)

Double the size of `buf`, and do it `shift` times. `shift` is usually 1, but in edge

cases it might need to be more, e.g. when appending a large register to a small one.

concat(dst, src)

Appends `src` to `dst`. This is done by resizing the destination buffer if required, and then copying all the data from `src` to `dst`. Whole machine words are copied first and the remaining 0 – 3 bytes are copied individually. If the data is not byte aligned, individual bits are copied last.

writeconst(buf, const, numbits)

Appends `const` to `buf`. Since `const` might be unaligned it needs to know how many bits to copy from `const`, which is given in `numbits`.

4.3 Pipeline implementation

The implementation of the transduction pipeline is inspired by our early efforts to separate a more complex program into separate sub tasks: we would compile each part separately, and write a wrapper script that piped data between the separate programs using Unix pipes. Our implementation uses an approach close to this.

In the compiler, each of the phases of the pipeline is compiled separately into their corresponding automata. The code produced by all automata are compiled into a single C file, which has a number of separate match functions.

When the generated program is called, it forks off a process for each transducer, setting up Unix pipes between them, taking the place of the last transducer in the pipeline.

4.4 Action implementation

The register actions are implemented using the same register implementation as above. This makes appending very cheap, while prepending is linear in the buffer size. It might be worthwhile to investigate other data types for these registers.

While the semantics says that all buffers that are output must also be cleared, this is not currently enforced by the implementation. This allowed us to experiment with programs that could copy data. In future versions this should be enforced, and an explicit copy action can be added to the register actions. This makes it explicit that the theoretical guarantees does not hold when arbitrary copying can be used.

4.5 Optimizations

In this section we will present various optimizations performed during the compilation and to the runtime. We implemented the optimizations discussed in the sections *I/O buffering*, *Bitcodes*, and *Bitcodes of suppressed terms* below. The other optimizations are from other contributors[2], but are described here for completeness.

Byte-alignment

In the case of string to string transductions we know that our buffers are byte-aligned, so they don't need the overhead of performing bit operations. The runtime can be compiled to take advantage of this with `FLAG_WORDALIGNED`. In the current implementation this flag is always enabled, except for in the benchmark of bitcodes.

I/O buffering

In order to avoid syscall and context switch overhead, we perform buffering on both input and output. This drastically improved the performance of simple Kleenex programs, and gave a noticeable boost to the more complex ones. We did not include this optimization in our benchmarks.

Bitcodes

An interesting question is whether to pack the bitcode in actual bits, or to use a fixed size of a byte instead. While a bit encoding uses less memory bandwidth, it also takes extra instructions to access the bits, since most modern CPU architectures operate with bytes as the smallest addressable unit. The bandwidth savings are also limited when large ranges are matched, and in the extreme case of matching $\cdot$ the bitcode takes up a byte anyway, just without the benefit of being byte aligned.

Streaming bitcodes are also much more difficult to implement. Apart from updating the model and runtime to support matching on variable length data, there is also the problem of detecting the end of the stream. As an example, the stream might end being unaligned by four bits, and no matter what the sending process packs in the last four it might be interpreted as a valid code! We could not think of an obviously satisfactory solution to this. You could send a sentinel value to mark the end of the stream, but it would cause overhead to check for it, and it might accidentally collide with a valid bitcode as well, which would cause the program to stop before time. Another solution would be to use out-of-band signals, but that could cause delays and would complicate the runtime even more.

Our bitcode implementation does not solve the problem. Instead we use the following heuristic: if we are in a final state and there is one byte or less data left, and the rest of the bits are zero, then we halt. This is not guaranteed to be correct, but it is good enough to benchmark as there can only be errors in the last couple of bytes of data. We did not encounter such errors during our benchmarks.

In Section 5.1 we benchmark the two approaches. The bytecode implementation tends to perform a little better on throughput, while the bitcode implementation requires less data to be transferred. If bandwidth is a concern the the bitcode approach might be useful, while in other situations the bytecode approach wins on simplicity, as it is never much worse than the bitcode approach.

Bitcodes of suppressed terms

While benchmarking the program `csv_project3`, seen in Figure 5.6, we observed a severe performance degradation of the action-enabled automaton, as compared to the single SST implementation. We further observed, that the program contained a lot of suppressed terms, and realized that the bytecode output produced by the oracle for these suppressed terms have no practical effect on the result, and might as well be omitted.

To determine which terms can be omitted, we first decide which subterms are *action-free*. By action-free we mean that their subtree contains no non-output action. This is a necessary requirement, as the action automaton needs to know if actions are to be done, and thus the bitcodes instructing it to do so cannot be optimized out.

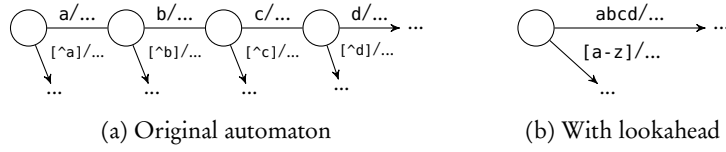


Figure 4.8: Common pattern of Kleenex programs

This analysis can be done by assuming all non-terminals action-free and performing a recursive analysis of the syntax-tree to determine which subterms are action-free under that assumption. If this changes the non-terminals we know not to be action-free, repeat the analysis using the new knowledge until we reach a fixed point.

Determining which subterms are suppressed is a simple recursive syntactical check.

When constructing the oracle OFT, no outputs are added to the edges of known action-free suppressed terms, and when constructing the action automaton, action-free suppressed terms are rewritten to 1.

This optimization cannot possibly degrade performance of the final automaton, as it only removes outputs and states from the automata.

In Section 5.1 we show benchmark of this optimization on the `csv_project3` program that inspired it.

Constant propagation

The generated SSTs contain a lot of trivial register updates, such as registers that are set to themselves, are concatenated with recently cleared registers, or are simply always constant. Many of them can be removed by constant propagation, which is a standard compiler optimization.

Finite lookahead

When Kleenex is used in practice, programs often have the form

```
token := ~/abcd/ commonCase | ~/[a-z]+/ fallback
```

In this case the automaton will match the string one character at a time, and have a fallback edge for each, as seen in Figure 4.8(a). Instead, we would like to test the entire string at once, with only a single fallback edge, like in Figure 4.8(b).

The performance of this optimization is discussed in Section 5.1.

4.6 Correctness

The Kleenex compiler and runtime are relatively large and complex pieces of software, which inevitably means that there will be bugs in the implementation. The following points helped us convince ourselves that the implementation is close to correct, since a full proof is outside the scope of this thesis.

Static typing Our work benefited hugely from being performed in Haskell, as most of our contributions involved refactoring of existing code. Purity, static

typing and equational reasoning were all important tools for us during the implementation work.

Unit and regression testing The project has a small unit test suite, and regression tests were added each time a parser bug was found. Some parts of the model, e.g. the bitcoding, also have property based testing. This was part of the project infrastructure, and thus not our own contribution.

Benchmarks For the benchmarks we built a small library of relatively complex Kleenex programs, and at least one parallel implementation in a different language for most of them. This allowed the benchmarks to function as advanced unit tests during the implementation work.

Chapter 5

Evaluation

5.1 Benchmarks

In this section we will cover the benchmarks we have conducted to compare Kleenex' performance with that of other tools, as well as to determine the effectiveness of our various optimizations.

The benchmarks were conducted in cooperation with the rest of KMC, and most of the graphs are also in the POPL article. Apart from the section on the experimental setup, all descriptions and interpretations are our own.

Our contribution consists of inventing and implementing the use cases in Section 5.2, creating reference implementations for many of the benchmarks and use cases in other languages, and creating data generator programs for the benchmark.

All the benchmark programs can be found in the `repg` GitHub repository under the `bench` folder¹.

Compilation times

The benchmark only measures runtime performance and ignores compile times. This favors Kleenex immensely, as all the disambiguation is done at compile time with the risk of a blowup in the number of states, which in turn would prevent compilation in practice. This makes the benchmark biased towards Kleenex, as such programs are not included in the benchmark.

We argue that it is still a meaningful benchmark. Since we are interested in streaming behavior, it is more interesting to optimize for high throughput than for fast compile or startup times. The implementations we benchmark against have been pre-compiled as much as they allow. Also, the compile times are not that bad for the benchmark programs, as most of them compile in less than 10 seconds on a modern laptop. Those that do not compile at all are too complicated to realistically implement as regular expressions.

The compile times of Kleenex might be improved by using a dual approach, where compilation is aborted after a certain amount of time, at which point the automata are simulated instead.

¹<https://github.com/diku-kmc/repg/tree/master/bench>

Correctness testing

For the comparisons to be fair, it is necessary (but not sufficient) that the programs we compare all give the same output on the same inputs. Our build tool has a mode that checks this, and all programs in the benchmarks passed before the benchmark was run.

Experimental setup

We have run comparisons with different combinations of the following tools:

RE2, Google’s automata-based regular expression C++ library [16].

RE2J, a recent re-implementation of RE2 in Java [17].

GNU AWK, GNU `grep`, and GNU `sed`, programming languages and tools for text processing and extraction [18].

Oniglib, a regular expression library written in C++ with support for different character encodings [19].

Ragel, a finite state machine compiler with multiple language backends [20].

DReX, a declarative language for expressing regular string transductions [21, 1].

In addition, we implemented test programs using the standard regular expression libraries in the scripting languages Perl [22], Python [23], and Tcl [24].

Meaning of plot labels

Kleenex plot labels follow the format [`<0|3>[-la] | woACT`] [`clang|gcc`], which indicates the compilation path and enabled optimizations.

0/3 indicates whether constant propagation was disabled/enabled.

la indicates that lookahead was enabled.

clang/gcc indicates which C compiler was used.

woACT indicates that the implementation uses a direct SST implementation without the oracle-action separation. This is only run with constant propagation and lookahead enabled.

Hardware and software parameters

The machine used for the benchmarks runs Linux, has 32 GB RAM and an eight-core Intel Xeon E3-1276 3.6 GHz CPU with 256 KB L2 cache and 8 MB L3 cache. Each benchmark program was run 15 times, after first doing two warm-up rounds. All the libraries and tools that the benchmarks were conducted against can be seen in Table 5.1. All C and C++ files have been compiled with `-O3`.

Name	Version
gcc	4.8.4
clang	3.5.0
Perl	5.20.1
Python	2.7.9
Tcl	8.6.3
GNU AWK	4.0.2
GNU grep	2.21
GNU sed	4.2.1
GNU coreutils	8.21
Oniguruma	5.9.6
RE2	GitHub: bdb5058
RE2J	1.0
Ragel	6.9
DRex	20150114

Table 5.1: Version numbers of tools used for benchmarking

Difference between Kleenex and the other implementations

The implementations used for comparison in other languages do not necessarily follow the same structure as the Kleenex program. Instead, the programs are written in the style that comes most natural in that given language. This means that many tools work on line-based approaches, that process one line at a time, and certain implementations use optimized libraries that are included with the language.

Ragel optimization levels

Ragel is compiled with three different optimization levels: T1, F1, and G2. “T1” and “F1” means that the generated C code should be based on a lookup-table, and “G2” means that it should be based on C `goto` statements.

Kleenex compilation timeout

On some plots, some versions of the Kleenex programs are not included due to the C code generation or C compilation timing out after 30 seconds. This is caused by the transducer determinization, which can lead to a blowup in SST size.

Benchmarked programs

flip_ab

As a baseline for throughput, we use the program `flip_ab` shown below.

```
main := (~a/ "b" | ~b/ "a" | ~\n/)*
```

This program simply takes an input consisting of lines of `as` and `bs`, and switches the two, while preserving newlines. This should provide a good baseline for throughput, as every byte in the input is touched but the transformation is very simple.

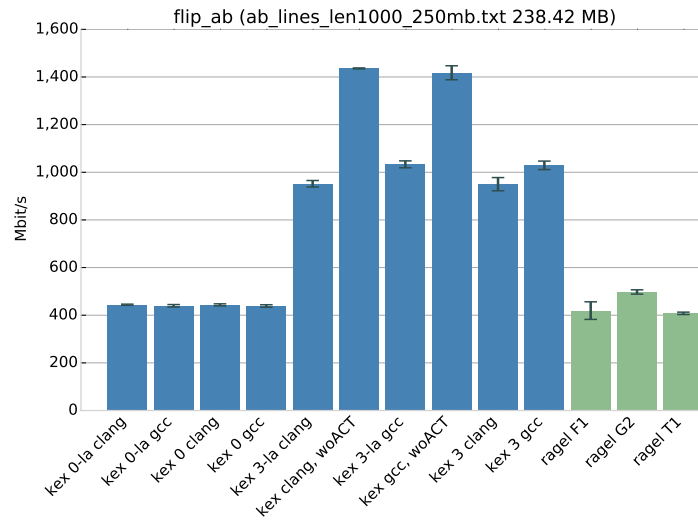


Figure 5.1: `flip_ab` run on a file with an average line length of 1000 characters.

We have benchmarked this program against a Ragel implementation on a file with an average line length of 1000 characters, and the results can be seen in Figure 5.1. The Ragel program can be seen in Figure 5.26.

The program was also benchmarked on data with a line length of 80 characters, yielding the same results. This is not unexpected, as neither Ragel nor Kleenex work on a line-by-line basis.

patho2

We have also benchmarked against pathological case for our implementation, which we call `patho2`.

```
main := ((~/[a-z]*a/ | /[a-z]*b/)? /\n/)+
```

This program forces our model to maintain two different outcomes until the end of each line, leaving us unable to stream any output until the line is finished.

In Figure 5.2, we see the results of benchmarking this program against a large number of different implementations. Kleenex performs well in comparison, which suggests that the other implementations take an even larger run-time performance hit than Kleenex does.

rot13

This program implements the ROT-13 rotational cipher and can be seen in Figure 5.3.

While superficially similar to `flip_ab`, the benchmark results in Figure 5.4 show a much more pronounced difference between the separated and non-separated versions as compared to Figure 5.1. A large part of this difference can likely be attributed to the extremely large amount of branching in the `rot13` term. The

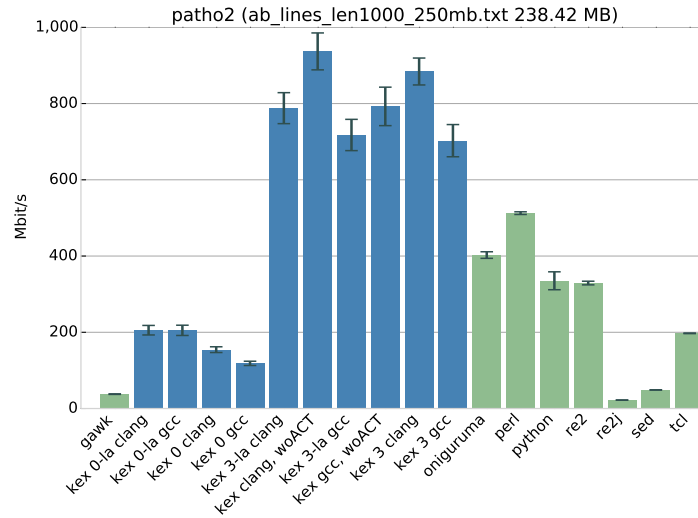


Figure 5.2: patho2 run on a file with an average line length of 1000 characters.

```
main := (rot13 | /./*)
```

```
rot13 := ~/A/ "N" | ~/B/ "O" | ~/C/ "P" | ~/D/ "Q" | ~/E/ "R" | ~/F/ "S"
        | ~/G/ "T" | ~/H/ "U" | ~/I/ "V" | ~/J/ "W" | ~/K/ "X" | ~/L/ "Y"
        | ~/M/ "Z" | ~/N/ "A" | ~/O/ "B" | ~/P/ "C" | ~/Q/ "D" | ~/R/ "E"
        | ~/S/ "F" | ~/T/ "G" | ~/U/ "H" | ~/V/ "I" | ~/W/ "J" | ~/X/ "K"
        | ~/Y/ "L" | ~/Z/ "M" | ~/a/ "n" | ~/b/ "o" | ~/c/ "p" | ~/d/ "q"
        | ~/e/ "r" | ~/f/ "s" | ~/g/ "t" | ~/h/ "u" | ~/i/ "v" | ~/j/ "w"
        | ~/k/ "x" | ~/l/ "y" | ~/m/ "z" | ~/n/ "a" | ~/o/ "b" | ~/p/ "c"
        | ~/q/ "d" | ~/r/ "e" | ~/s/ "f" | ~/t/ "g" | ~/u/ "h" | ~/v/ "i"
        | ~/w/ "j" | ~/x/ "k" | ~/y/ "l" | ~/z/ "m"
```

Figure 5.3: The program text for the rot13 program, which rotates each letter 13 places.

implementation does not currently attempt to balance the syntax tree at any point before constructing the OFT. In this case, it means that the bytecode produced by the disambiguation is very skewed in length: Giving the rot13 program an input of `A` leads to a 4 character bytecode², whereas a `z` gives a 54 character bytecode.

This could be improved significantly by either balancing the syntax tree, which would give a bytecode of logarithmic length in all cases, or compiling the alternation down to an OFT with multi-way branching rather than the current two-way branching. This latter option would allow for the bytecode to be a constant 4 for any of the 52 characters.

This program is also a good example of a program that could benefit from an extended action language. While the ROT13 cipher is a simple mathematical cipher, expressing it in Kleenex is currently quite cumbersome. Introducing an action that could allow e.g. computation on a register could allow this program to become much shorter, simpler, and faster.

²2 bytes for the Kleene star in `main`, 1 for the alternation in `main`, and 1 from `rot13`

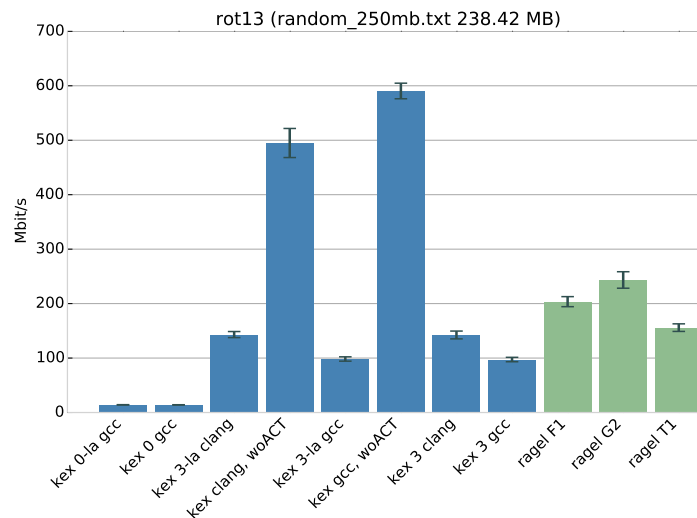


Figure 5.4: rot13 run on a file consisting of random data.

thousand_sep

This program takes a list of numbers of any length, and inserts thousands' separators in each of the numbers. As an example, `1234567890` becomes `1,234,567,890`.

```
main := (num /\n/)*
num := digit{1,3} ("," digit{3})*
digit := /[0-9]/
```

This is an example of a problem at which Kleenex excels. The problem is a somewhat cumbersome one to solve in many languages, and this is in large part due to the positions of the thousands separators being determined by the length of the number. Whereas one would have to either determine the length of the string in advance or work from the back of the string in other languages, the disambiguation provided by Kleenex allows for a clean formulation. As seen in Figure 5.5, our implementation performs well on this problem compared to both Python and Perl.

csv_project3

The program `csv_project3` implements a common task in string manipulation: Data projection.

The program, seen in Figure 5.6, takes a six-column CSV file as input, and projects its second and fifth columns out.

In Figure 5.7 we see how our implementation fares in comparison to a number of common tools, including `cut`, a tool specialized for precisely this use case.

Although the specialized tool does perform better than Kleenex, our pure SST implementation performs better than the other general purpose tools, and the split approach is still competitive.

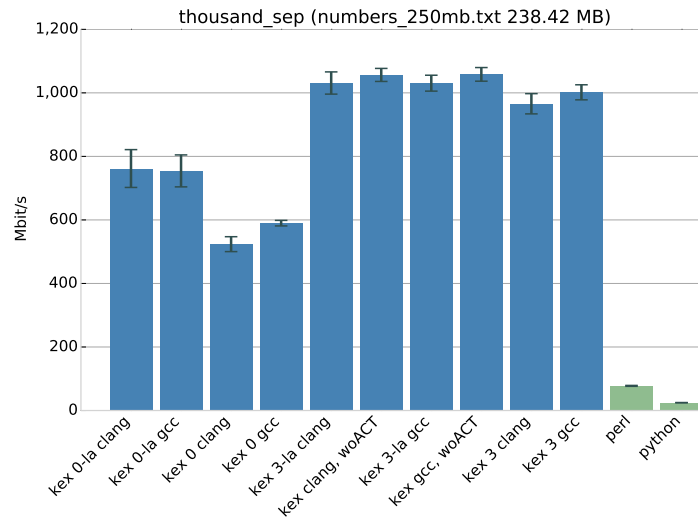


Figure 5.5: `thousand_sep` run on a file consisting of numbers with an average length of 1000 digits.

```
main := (row /\n/)*

row := ~(col /,/) col "\t" ~/,/ ~(col /,/)
      ~(col /,/) col ~/,/ ~col

col := /[^\n]*/
```

Figure 5.6: The code for `csv_project3`, which projects columns 2 and 5 out of a 6-column CSV-file.

iso_datetime_to_json

Parsing a timestamp into its subcomponents is an example of a common data extraction problem one might use regular expressions to solve.

The program `iso_datetime_to_json` (Figure 5.8) takes a file consisting of ISO 8601 combined date and time stamps [25] and rewrites them into JSON dictionaries consisting of their individual components. This program is heavily inspired by an example from page 237 of [26].

In Figure 5.9 the performance of the program is shown in comparison to related tools. There is not much difference between the optimization levels, and the variance is high for all the Kleenex versions. We do not have a good hypothesis for why this is the case, which makes it an interesting case for further performance profiling.

Benchmarked optimizations

In addition to benchmarking against other implementations, we have also conducted a number of internal benchmarks, in order to determine the effect of the optimizations we have applied.

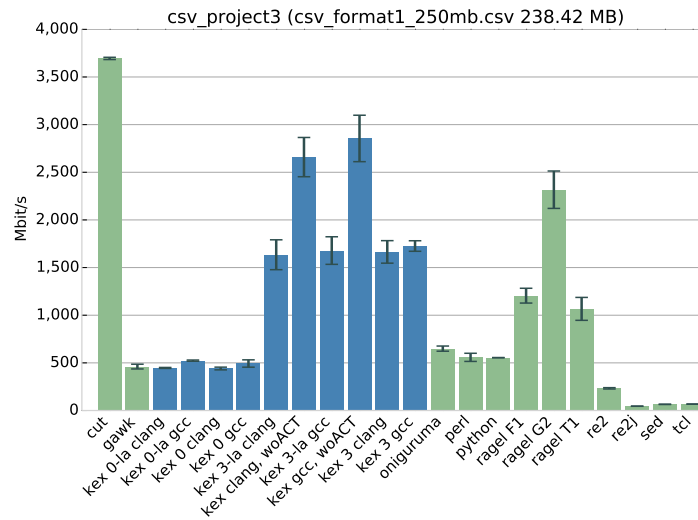


Figure 5.7: csv_project3 projects columns 2 and 5 out of a 6-column CSV file.

```

main := (dateTime ~/\\n/)+

dateTime := "{ 'year'=' year ~- / " "
            ", 'month'=' month ~- / " "
            ", 'day'=' day ~ / T / " "
            ", 'hours'=' hours ~ / : / " "
            ", 'minutes'=' minutes ~ / : / " "
            ", 'seconds'=' seconds " "
            ", 'tz'=' timezone " "
            "}\\n"

year := /(?:[1-9][0-9]*)?[0-9]{4}/
month := /1[0-2]|0[1-9]/
day := /3[0-1]|0[1-9]|1[0-2][0-9]/
hours := /2[0-3]|0[0-9]/
minutes := /[0-5][0-9]/
seconds := /[0-5][0-9]/
timezone := /Z|[-+](?:2[0-3]|0[0-9]):[0-5][0-9]/

```

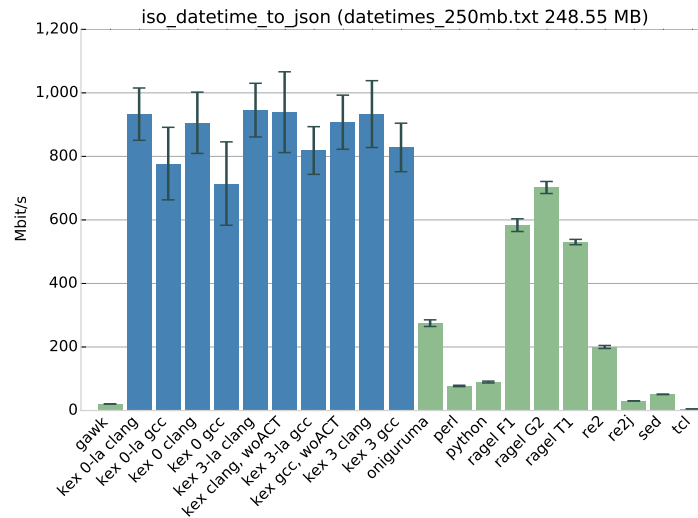
Figure 5.8: The program iso_datetime_to_json, which parses an ISO 8601 combined date and timestamp into a JSON dictionary.

Constant propagation

In nearly all of our benchmarks, we see a marked improvement in throughput from the constant propagation, which is what we expected.

Lookahead

The lookahead optimization shows no significant improvement in any of our benchmarks. It is targeted towards programs that matches on many keywords, and there are not any good examples of this in among the benchmark programs.

Figure 5.9: Throughput for the program `iso_datetime_to_json`.

It does have a positive effect which is not measured in the benchmark, namely that it decreases the number of states in the combined SST and the oracle SST, thus leading to smaller executable sizes. The difference is negligible, however, and evidently does not contribute to better runtime performance.

Single SST vs oracle-action split

One of the factors measured in our benchmarks was how well the two-part oracle and action automaton fares in comparison to a single SST.

In general the data suggests that the single SST tends to be just as fast, if not faster than the split automaton.

Furthermore, in benchmarks like Figure 5.7 and Figure 5.4, we experience a significant performance downgrade from using the action-enabled implementation.

We conjecture that the slowdown comes from a number of factors. One factor could be that the bytecode stream produced by the oracle often is larger than the original input data, which would throttle the application due to lack of memory bandwidth. Another factor could be the way we transfer data between transducers. Currently we use Unix pipes, where an implementation that uses a shared memory model may improve performance.

The major theoretical advantage of the split approach is that it is inherently parallel, as it splits up the disambiguation from the actions that are executed. We conjecture that this allows more computationally expensive actions to be added without loss of performance, as the bottleneck currently seems to be in the disambiguation stage.

Bytecodes of suppressed terms

The removal of unnecessary bytecode generation for suppressed terms had a big impact on the `csv_project3` benchmark, doubling the throughput in on the higher optimization levels, as seen in Figure 5.10.

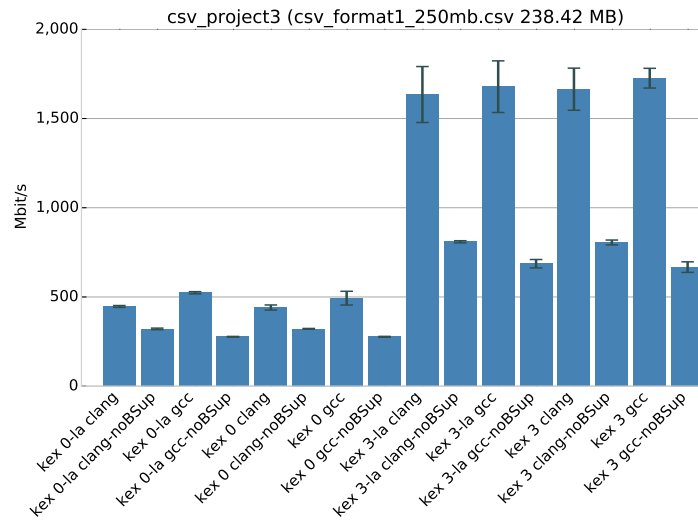


Figure 5.10: csv_project3 with and without suppressed bitcode optimization. The bars labelled noBSup show the non-suppressed implementation

The efficiency of this optimization depends highly on how large a portion of the program is suppressed, as well as how many actions are used. For many of our programs the optimization provides little, if any, benefit, as they often contain few suppressed terms. On programs that do use suppression, the effect is significant.

Bitcode vs bytecode

The implementation of the action enabled machines uses bytes rather than bits for sending data from the oracle to the action automaton. This has the disadvantage that the navigational indices can easily increase the data being processed by a factor of $2\text{-}3\times$ or more.

In order to see if we could obtain any speed gains from moving to bitcodes, rather than bytecodes, we implemented a bitcode version of the machine and runtime.

As Figure 5.11 and Figure 5.12 show, the bitcode implementation does not improve on the performance of the programs that are significantly slower with actions enabled.

As Figure 5.13 shows, no performance gains were obtained in a test where the pure SST is at the same speed as the action enabled version, either. This suggests that the memory bandwidth limitations, if any, costs less than the extra operations required to do bit addressing. This was surprising to us, and it would be interesting to see if this is the case on other architectures as well.

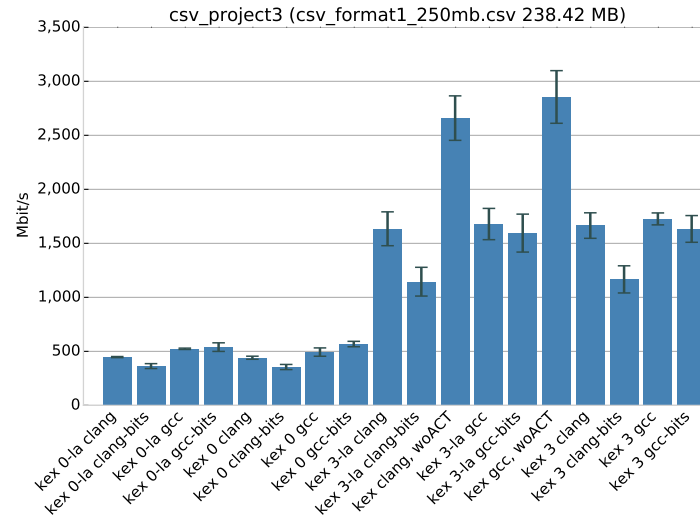


Figure 5.11: Bitcode vs. bytecode implementation of csv_project3. The implementations labelled bits use the bitcode implementation.

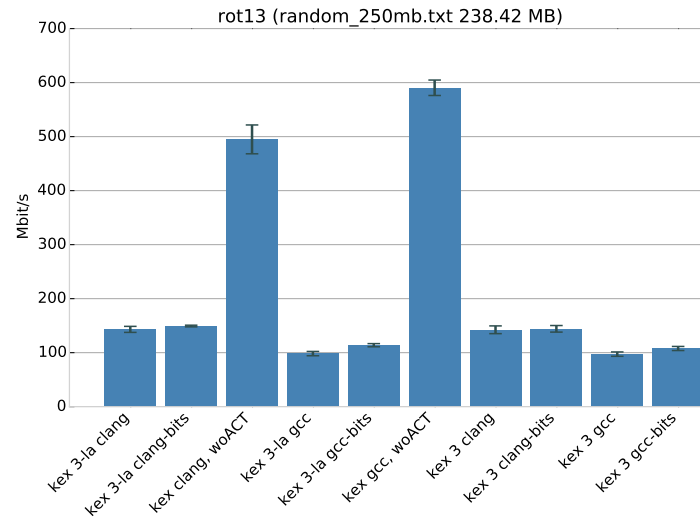


Figure 5.12: Bitcode vs. bytecode implementation of rot13. The implementations labelled bits use the bitcode implementation.

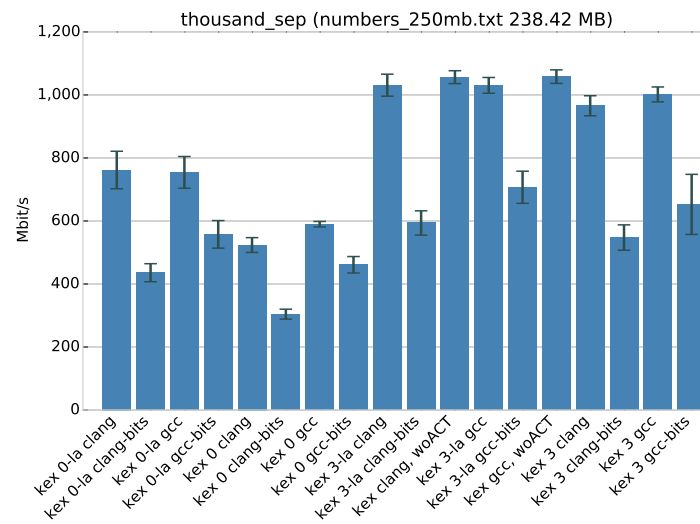


Figure 5.13: Bitcode vs. bytecode implementation of `thousand_sep`. The implementations labelled `bits` use the bitcode implementation.

5.2 Use cases

In this section we will cover practical use cases for Kleenex, both ones that work right now and which we have tested, but also uses which are possible with small changes to the language and the runtime. The aim of this section is to demonstrate the expressivity and versatility of the language.

Data conversion

Kleenex makes it feasible to define much more complex regular transformations than traditional regular expression engines. This makes rewriting between regular data formats an approachable task. In Figure 5.14, we have a program, `ini2json`, which rewrites an INI configuration file to a JSON nested dictionary.

```
start: stripini >> ini2json

// Strips the comments
stripini := (~comment | ~blank | /[^\n]*\n/)*

comment := ws /[^\n]*/
blank := ws /\n/

// Convert the stripped file
ini2json := "{\n" sections "}\n"

sections := (section ", " /\n/)* section /\n/
section :=
  ind "\" header "\": {\n" (~/\n/ keyvalues)? ind "}"

header := ~ws ~/[/ / [^\n]]*/ ~/]/ ~ws

keyvalue := ind ind key "": " ~/=/ value
keyvalues := (keyvalue ", " /\n/)* keyvalue "\n"

key := ~ws "\" /[^\n; \t=\\[\n]]*/ "\" ~ws

value := ~ws /"[^\n]*"/ ~ws
        | ~ws "\" escapedValue "\" ~ws

escapedValue := (~/\[/ "\\\\" | ~/" / "\\\\" | /[^\n]/)*

ws := /[ \t]*/
ind := " "
```

Figure 5.14: Kleenex program with a pipeline of two transductions. First comments and blank lines are stripped from an INI file, which is then converted to a specific JSON representation

The INI file format is regular, with a number of sections, each containing a number of keys. Most regular expression implementations cannot take advantage of this regularity, however, as it has Kleene-star depth of 2 for the parts of the format you would normally wish to extract.

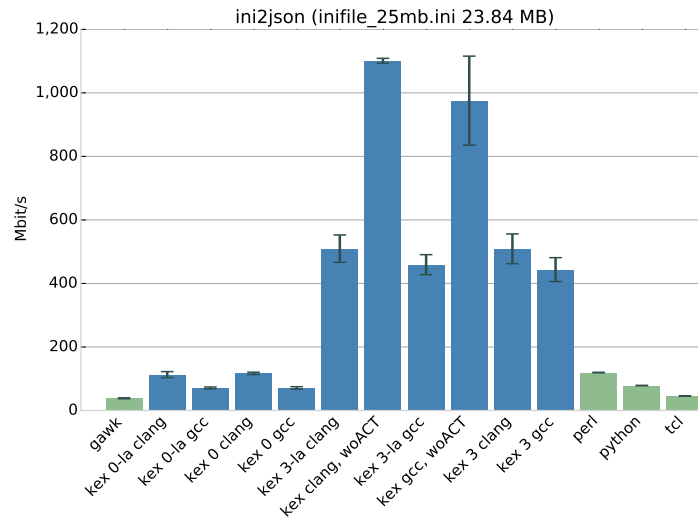


Figure 5.15: Throughput of the program ini2json.

In Figure 5.15 the throughput can be seen in comparison to some other languages that would normally be used for this sort of task.

Man-in-the-middle attacks

An example of such an attack is matching valid HTML forms and changing their actions. The Kleenex program in Figure 5.16 changes the action to submits the form to a another server, which can then track the user and gain sensitive information.

Several of Kleenex's properties makes it a good fit for this domain³. It is easy to express the conditions to do specifically targeted attacks in Kleenex, such as replacing a specific form if the title and a few other elements matches. The streaming nature of Kleenex and smoothly degrading performance makes it hard to detect the attack based on performance, and makes it feasible to perform the attack on all traffic that passes through a node, which avoids the problem of filtering traffic and only applying it to e.g. GET requests.

```

main := /<form/ sp /action=/ orig@url evil sp />/ main
      | ./ main
      | ""

url  := q? /["' >]/* q?
q    := ~/"|'/
addq := "\""
sp   := / /*
evil := addq "http://evil.com/?url=" !orig addq

```

Figure 5.16: Man-in-the-middle attack on HTML pages with forms

³This is a bit worrying to us

Log rewriting

Kleenex is a good fit for rewriting log files, either to convert them to a common format, to compress them, or to create tools that can automatically annotate log files with column names and other information that is implicit in the log format. The following Kleenex program is of the last kind, as it takes apache log files and tags each field with its column name.

```

main := "[" loglines? "]" \n

loglines := (logline ", " /\n/)* logline /\n/
logline := "{" host ~sep ~userid ~sep ~authuser sep timestamp sep
           request sep code sep bytes sep referer sep useragent "}"

host := "\"host\":" ip "\""
userid := "\"user\":" rfc1413 "\""
authuser := "\"authuser\":" /\n+/ "\""
timestamp := "\"date\":" ~/\n/ /\n+/ ~/\n/ "\""
request := "\"request\":" quotedString
code := "\"status\":" integer "\""
bytes := "\"size\":" (integer | /-/) "\""
referer := "\"url\":" quotedString
useragent := "\"agent\":" quotedString

ws := /\t +/
sep := ", " ~ws

quotedString := /"([^\n]|\\")*/
integer := /[0-9]+/
ip := integer (/./ integer){3}
rfc1413 := /-/

```

Figure 5.17: The `apache_log` program, which parses and annotates apache log files.

Lexical analysis and syntax highlighting

Kleenex can also be used for lexical analysis, i.e. recognizing the lexemes of a given programming language, and outputting tokens for a parser to process. Lexers are typically auto-generated by a lexer generator, such as Flex.[\[27\]](#)

In order for Kleenex to be useful as a lexer, it needs the ability to integrate with another language and be extended with arbitrary actions as discussed in Section 6.2.

A program demonstrating basic lexing on PL/0 [\[28\]](#) is shown in Figure 5.19. This is of course a bit contrived, as it is almost as hard to use the generated tokens as it is to just do the lexical analysis again.

Another closely related problem is *syntax highlighting*, where lexemes in the source code needs to be embraced by color codes depending on the environment in which the program is shown, i.e. custom div tags for web page or color codes for terminal applications. In Figure 5.20 we show a syntax highlighter for Kleenex which outputs ANSI color codes.

An actual real world example can be seen in Figure A.2, namely a Kleenex program that syntax highlights Kleenex programs with pygments $\text{\LaTeX}$ color codes. All Kleenex programs in this thesis have been highlighted by this program.

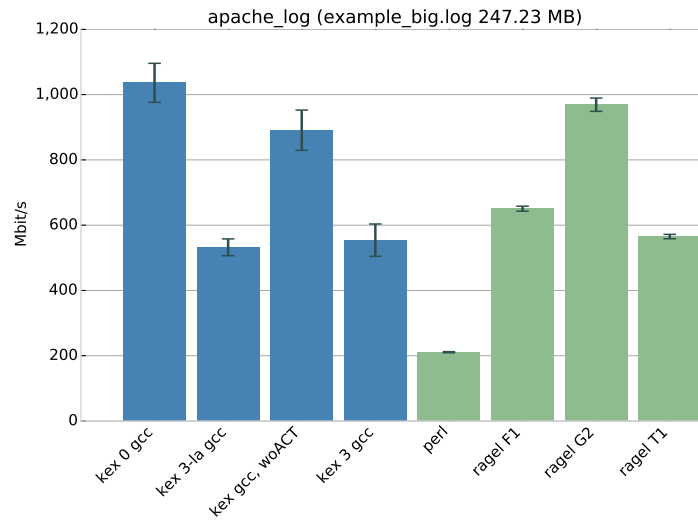


Figure 5.18: Throughput comparison on the apache_log program.

```

main := ((keyword | symbol | ident | number) " " | ws)*

number := "NUMBER(" digit+ ")"
ident := "IDENT(" letter (letter | digit)* ")"
keyword := /BEGIN|CALL|CONST|DO|END|IF|ODD|PROCEDURE|THEN|VAR|WHILE/
symbol :=
    ~/\+/"PLUS" | ~/-/"MINUS" | ~/\*/"TIMES" | ~/\// "SLASH"
    | ~/\(/ "LPAREN" | ~/\)/"RPAREN" | ~;/"/ "SEMICOLON" | ~/,/ "COMMA"
    | ~/\./ "PERIOD" | ~/:="/ "BECOMES" | ~/=/"EQL" | ~/<"/ "NEQ"
    | ~/<="/ "LEQ" | ~/>="/ "GEQ" | ~/</"LSS" | ~/>/"GTR"
digit := /[0-9]/
letter := /[a-zA-Z]/
ws := /[ \t\n\r]*/

```

Figure 5.19: Lexer for the instructional programming language PL/0.

Protocol parsing

Some protocols have regular subsets, or are entirely regular, and Kleenex excels at parsing these. A good example of this are messages in the IRC protocol[29]. The Kleenex program in Figure 5.21 is almost identical to the BNF grammar given in the RFC, which made it very easy to implement compared to the complexity of the protocol.

```

main := ( escape | comment | term
          | symbol | ignored | ws* ) *
term  := black ~/ (constant | match | ident) end
      | (teal constant | yellow match | blue ident) end
ignored := /[()|{|},:[]/
ident   := (letter | /[0-9_]/)+
symbol  := yellow /<-|\+=|:=|>>|\*|\?|\+/ end
constant := "/" ( /\./ | /[^\"]]/ ) * "/"
comment  := black ( /\[/[^\n]*\n/
                  | /\[/[^\n]*\*\/ end
match     := /\[/ ( /^[^\n]/ | /\./ )+ /\[/
escape    := /\[/
          | blue /\[x[0-9a-fA-F]{2}/ end
          | /\[t\nr]/
sp := / / *
letter := /[a-zA-Z]/
word := letter+
ws := /\[t\r\n ]/
red   := "\x1b[31m"
green := "\x1b[32m"
yellow:= "\x1b[33m"
blue  := "\x1b[34m"
end   := "\x1b[39;49m"
black := "\x1b[30m"
teal  := "\x1b[36m"

```

Figure 5.20: Syntax highlighter for Kleenex

```

main := (message | "Malformed line: " /[^\r\n]*\r?\n/)*

message := (~:/ "Prefix: " prefix "\n" ~/ /)?
          "Command: " command "\n"
          "Parameters: " params? "\n"
          ~crlf

command := letter+ | digit{3}

prefix := servername
         | nickname ((!/ user)? /@/ host )?

user := /[^\n\r @]/+ // Missing \x00

middle := nospcrlfcl ( /:/ | nospcrlfcl )*

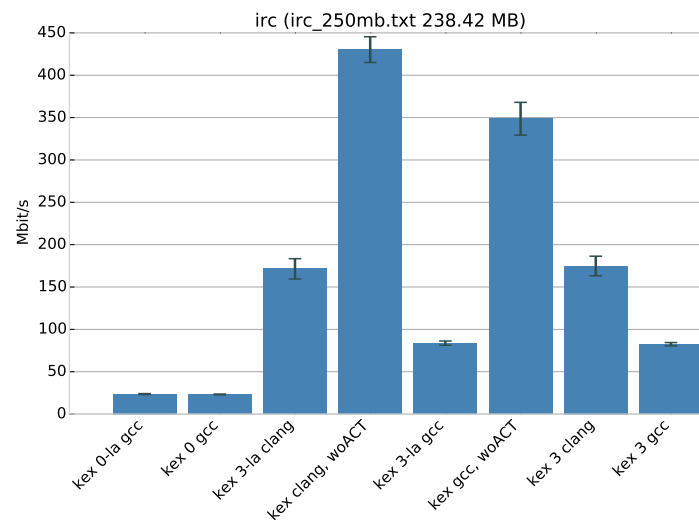
params := (~/ / middle ", "){,14} ( ~/ /: trailing )?
         | (~/ / middle ){14} ( // /:/ trailing )?
trailing := ( /:/ | // / | nospcrlfcl )*
nickname := ( letter | special ) ( letter | special | digit){,10}

host := hostname | hostaddr
servername := hostname
hostname := shortname ( /\. / shortname)*
hostaddr := ip4addr
shortname := (letter | digit) (letter | digit | /- /)* (letter | digit)*
ip4addr := (digit{1,3} /\. / ){3} digit{1,3}

letter := /[a-zA-Z]/
digit := /[0-9]/
hexdigit := /[a-fA-F0-9]/
crlf := /\r?\n/
nospcrlfcl := /[^\n\r :]/ // Missing \x00
special := /[\\[\ \_ ^ { } ]]/

```

Figure 5.21: The program `irc`, which parses the IRC protocol.

Figure 5.22: Throughput of the program `irc`.

5.3 Comparison

In this section we compare Kleenex with DReX and Ragel, two other projects that have use cases in common with Kleenex, based on performance and expressive power.

DReX

DReX is a declarative language which was presented in [1]⁴. It is based on a set of combinators which can express all regular string transformations[21].

In the paper [1] they present a number of benchmark programs to demonstrate the expressivity of their language. We have written equivalent Kleenex programs, shown in Figure 5.23.

```
// delete_comm
delete_comm := (~/\V\// ~line | line)*
line := /[^\n]*\n/

// insert_quotes
insert_quotes := ("\" / [a-zA-Z]+/ "\"" | /.)/

// get_tags
get_tags := (<[^>]*> | ~/./*

// reverse
reverse := (a@[^;]*;/ c@(!a !b) b@(!c))* !b

// swap_bibtex and align_bibtex
swap_bibtex := (header field* !title put_rest footer)*
align_bibtex := head@header field* foot@footer
               (!head head@header put_rest field* !title !foot foot@footer)*

field := title@(sp /title/ sp /=/ sp /\{[^]*\},?\n/)
        | f@(sp word sp /=/ sp /\{[^]*\},?\n/) [ rest += f ]

header := /@/ word sp /\{/ sp alnum /, \n/
footer := /}/ (sp|/\n/)*
put_rest := !rest [ rest <- "" ]

word := /[A-Za-z_]+/
alnum := /[A-Za-z0-9_]+/
sp := /[ \t]*/
```

Figure 5.23: Kleenex versions of the programs presented in the DReX paper from POPL 2015 [1]

To be able to properly compare implementations, we also wanted to benchmark our implementation against theirs. However, we were unable to get their implementation to work with more than 2 MB of input data. We did run a benchmark of swap-bibtex with this low amount of data, the results of which can be seen in Figure 5.24 and Figure 5.25. Similar results were achieved for the other programs.

⁴An online version is available at <http://drexonline.com>

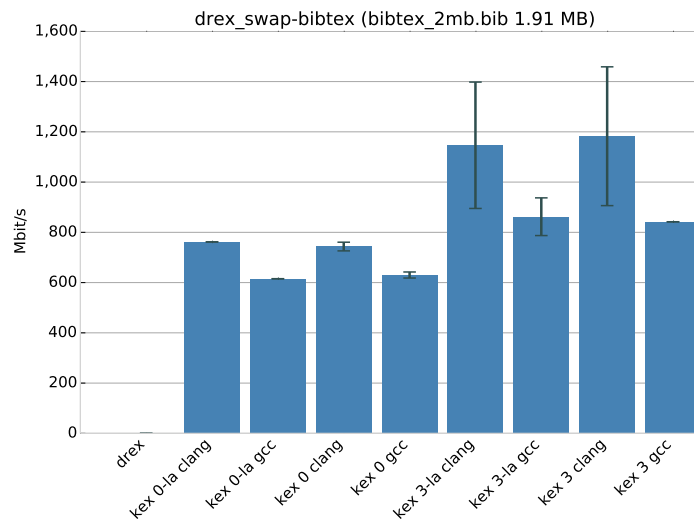


Figure 5.24: Throughput of the program `drex_swap-bibtex` on 2 MB of input data.



Figure 5.25: Execution time of the program `drex_swap-bibtex` on 2 MB of input data.

In their benchmark they build the program AST directly instead of providing source code, and the paper does not provide the full source code either. Thus we are unable to present any of the full DReX programs we tested against.

We personally find it easier to grasp and use Kleenex compared to DReX. While DReX is theoretically quite expressive and elegant, we find it much too cumbersome for practical purposes.

Ragel

Ragel[20] is an open source state machine compiler targeted at input validation, parsing, and protocol implementations. The syntax of Kleenex used Ragel as inspiration, and thus the languages have a similar look and feel.

A Ragel program consists of one or more *Finite State Machine* definitions, which is embedded in a host language that Ragel supports⁵. A FSM consists of several variables containing the normal regular expression operators, extra operators such as intersection and difference, and user defined actions. These actions consists of code fragments from the host language, so the user can perform arbitrary computation at any point in the machine. The Ragel compiler translates the state machines into the host language, so the end result is a file in the host language without external dependencies.

Ragel does not perform disambiguation. It is up to the user to specify the order in which the rules are tried by giving them priorities, and to make sure that all ambiguity is resolved before actions are encountered. While this does not matter much when writing e.g. parsers, it makes some of the string transformations in the benchmark prohibitively difficult to write. As an anecdotal data point, we gave up trying to express `ini2json` in Ragel after trying for an entire day.

Currently Kleenex and Ragel fill different niches. Ragel has wide language support, and is excellent for specifying state machines that would otherwise be implemented ad-hoc in the host language. Kleenex is currently better at string to string rewriting, but is unable to integrate with other languages. A better comparison can be made if Kleenex is extended with support for integration with other languages.

For string transformations, the performance of Ragel and Kleenex is comparable, mostly with a small lead to Kleenex in our benchmarks.

The Ragel program `flip_ab` we wrote for the benchmarks above can be seen in Figure 5.26.

⁵Ragel currently supports C, C++, Obj-C, C#, D, Java, Go and Ruby

```

#include <string.h>
#include <stdio.h>
#include <sys/time.h>
#include <stdint.h>
#include <inttypes.h>

#define BUFFER_SIZE (200*1024*1024)
char buffer[BUFFER_SIZE] = {0};
char outchar;
%%{
    machine flip_ab;
    action out {
        fputs(&outchar, stdout);
    }

    a = 'a' > { outchar = 'b'; };
    b = 'b' > { outchar = 'a'; };

    main := (a | b)* $ out;
}%%

%% write data;

int main(int argc, char **argv) {
    int cs;
    char *eof;

    while(fgets(buffer, sizeof(buffer), stdin)) {
        char *p = &buffer[0];
        char *pe = p + strlen(buffer);
        %% write init;
        %% write exec;

        fputc('\n', stdout);

        if(p+1 != pe) {
            fprintf(stderr, "matching failed (%p %p)!\n", p, pe);
            return 1;
        }
    }

    return 0;
}

```

Figure 5.26: The Ragel implementation of flip_ab

Chapter 6

Conclusion

6.1 Related work

Streaming string transducers (SSTs) were first introduced by Alur, Černý[30]. They are equivalent to a combinator language[21], which are used in the language DReX[1].

Kearns [31], Frisch and Cardelli [32] devise a 3-pass linear-time greedy regular expression parsing. Grathwohl, Henglein, Nielsen, Rasmussen devise a linear-time two-pass [33] and an optimally streaming [9] greedy regular expression parsing algorithm. The latter algorithm can be made linear-time by dropping the optimality requirement.

SSTs can be extended to cost register automata, where output strings are generalized to arbitrary monoids[34]. They belong to the complexity class NC[35] and are thus inherently parallelizable. Similarly, Veanes, Molnar, Mytkowics use symbolic transducers[36, 37, 12] to extend BEK[38] with data parallelism[39].

6.2 Future work

The current implementation does not currently ensure that the programs received are properly tail recursive, or that they do not contain problematic paths. To detect proper tail recursion, one could construct a directed graph of the nonterminals, adding a strong edge between nonterminals that must be of different order, and a weak edge between nonterminals that may share different order. The problem is then reduced to finding strongly connected components containing a strong edge.

Fully determinizing the oracle OFT into an SST can lead to a large blowup in the number of states. For those problematic machines, adaptively choosing to compile down to an implementation that simulates the OFT directly could make some currently intractable problems tractable. For those cases, one would likely expect a performance hit.

Compiling the machines down to use multi-striding may lead to speedup in execution times, as similar results have been seen on NFAs [40].

The machines could be compiled down to use jump tables, rather than the current goto construction. Judging by the benchmarks run against Ragel, the goto approach seems likely to perform better, however, this may be worth exploring.

Allowing the user to define functions on terms would allow for more flexible code re-use. As an example, to use a color in Figure 5.20, the programmer needs to use both a color start, as well as an end tag. This could benefit from allowing

the user to declare macro functions on the term level. Implementation can be done purely as a desugaring operation.

Extending Kleenex to allow more complex actions could allow the language to be useful in more situations.

A useful extension of the action language would be the ability to run a transducer, or a pipeline consisting of several transducers, on the contents of a given register, transducing the result into another register. This would allow the programmer to separate matching from rewriting. It would also give the programmer a tool for combatting exploding oracle SST sizes, by letting them decompose their problem. The existing infrastructure can be reused for implementing this, mainly requiring implementation of input redirection. This could be done in the same fashion as the output redirection provided by the push/pop actions.

Allowing more complex code, either a DSL, arbitrary C-code or similar as actions could allow a programmer to leverage the parsing capabilities of Kleenex, without the indirection of having to parse Kleenex' generated output.

In the same vein, having the Kleenex compiler generate shared libraries could make the language feasible as a parser inside another program. One could imagine an IRC daemon using Kleenex to parse the protocol into its constituent parts, a program using Kleenex to build up a syntax tree for an INI file, and so on.

Compilation of an OFAT with a limited set of actions down to an single SST could yield a potential speedup, as we have observed a general tendency for our single SST implementation to be faster than the split oracle-action version.

The current implementation makes use of separate processes with Unix pipes for communication in the implementation of the pipeline, as well as between the oracle and action automata. Using threading or a similar shared memory-space model may give performance benefits, at the cost of an increase in complexity.

Shared memory space between the different phases could make parallel evaluation of two or more transducers on the same input an area of exploration. One possible semantics could be, that two the parallel transducers evaluate to the longest common prefix of their transductions, or each could possibly transduce into their own buffer, allowing a subsequent transduction to merge them.

6.3 Closing remarks

We have extended the Kleenex language with support for actions, allowing for more complex transductions to take place. As a theoretical basis for this extension we have introduced a family of transducers we call ordered finite action transducers (OFATs), and shown how they can be simulated using two automata, a streaming string transducer oracle, and an action automaton which is capable of evaluating arbitrary actions along its path. We also show that an OFAT with the Kleenex core actions is equivalent to an NSST.

We have defined a core subset of Kleenex and defined its semantics in terms of the OFAT model, and shown how to rewrite the full Kleenex language to this core language.

We have shown that by compiling an OFT to an SST and implementing registers as dynamic arrays, we can achieve worst case $O(mn)$ transduction times when our register actions are not used, where m is the number of states in the OFT and n is the input size. This matches the implementation, which shows linear transduction times for fixed automata. Further, we have shown that it is possible to implement

general Kleenex actions with $O(nm^2(r + 1))$ worst case transduction times, where m is the number of states in the OFT, n is the size of the input, and r is the number of registers used in actions.

We have extended the repg Kleenex implementation with concept of a pipeline, making multi-phase transductions more practical for the programmer.

To test the performance and expressiveness of Kleenex and our extensions, we have constructed programs that demonstrate the expressiveness of Kleenex compared to other languages and tools. We have conducted benchmarks against these languages, achieving good performance characteristics on large amounts of data.

Finally we have contributed to a paper about the Kleenex language and compilation process, which is currently in review for the 2016 Symposium on Principles of Programming Languages (POPL).

Chapter 7

References

- [1] Rajeev Alur, Loris D’Antoni, and Mukund Raghothaman. Drex: A declarative language for efficiently evaluating regular string transformations. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 125–137. ACM, 2015.
- [2] Bjørn Bugge Gratwohl, Fritz Henglein, Ulrik Terp Rasmussen, Kristofer Aalund Søholm, and Sebastian Paaske Tørholm. Kleenex: Compiling nondeterministic transducers to deterministic streaming transducers. *Submitted for publication, 43rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2016)*, 2016.
- [3] Michael Sipser. *Introduction to the Theory of Computation*. Cengage Learning, 2nd edition, 2005. ISBN 978-0534950972.
- [4] Ken Thompson. Programming techniques: Regular expression search algorithm. *Commun. ACM*, 11(6):419–422, June 1968. ISSN 0001-0782. doi: 10.1145/363347.363387. URL <http://doi.acm.org/10.1145/363347.363387>.
- [5] Robert McNaughton and Hisao Yamada. Regular expressions and state graphs for automata. *IEEE Transactions on Electronic Computers*, 1(EC-9):39–47, 1960.
- [6] Michael O Rabin and Dana Scott. Finite automata and their decision problems. *IBM journal of research and development*, 3(2):114–125, 1959.
- [7] Jean Berstel. *Transductions and Context-Free Languages*. Teubner, 1979.
- [8] Jean Berstel. *Transductions and Context-Free Languages*. Teubner Studienbücher, Stuttgart, 1979. URL <http://www-igm.univ-mlv.fr/~berstel/LivreTransductions/LivreTransductions14dec2009.pdf>.
- [9] Niels Bjørn Bugge Grathwohl, Fritz Henglein, and Ulrik Terp Rasmussen. Optimally Streaming Greedy Regular Expression Parsing. In *Theoretical Aspects of Computing - ICTAC 2014 - 11th International Colloquium, Bucharest, Romania, September 17-19, 2014. Proceedings*, pages 224–240, 2014. doi: 10.1007/978-3-319-10882-7_14.
- [10] Rajeev Alur and Pavol Černý. Expressiveness of streaming string transducers. In *Proc. Foundations of Software Technology and Teoretical Computer Science (FSTTCS)*, 2010. doi: 10.4230/LIPIcs.FSTTCS.2010.1. URL <http://dx.doi.org/10.4230/LIPIcs.FSTTCS.2010.1>.

- [11] Bruce W Watson. Implementing and using finite automata toolkits. *Natural Language Engineering*, 2(04):295–302, 1996. ISSN 1469-8110. doi: 10.1017/S135132499700154X.
- [12] Margus Veanes, Pieter Hooimeijer, Benjamin Livshits, David Molnar, and Nikolaj Bjorner. Symbolic finite state transducers: Algorithms and applications. In *Proceedings of the 39th Annual Symposium on Principles of Programming Languages*, POPL ’12, pages 137–150, New York, NY, USA, 2012. doi: 10.1145/2103656.2103674.
- [13] Gertjan van Noord and Dale Gerdemann. Finite State Transducers with Predicates and Identities. *Grammars*, 4(3):263–286, 2001. ISSN 1386-7393. doi: 10.1023/A:1012291501330.
- [14] Rajeev Alur and JV Deshmukh. Nondeterministic streaming string transducers. *Automata, Languages and Programming*, 2011.
- [15] Daan Leijen and Erik Meijer. Parsec: Direct style monadic parser combinators for the real world, 2002.
- [16] The RE2 authors. RE2, 2015. URL <https://github.com/google/re2>.
- [17] The RE2J authors. RE2J, 2015. URL <https://github.com/google/re2j>.
- [18] The GNU Project, 2015. URL <http://www.gnu.org/software/coreutils/coreutils.html>.
- [19] K. Kosako. The Oniguruma regular expression library, 2014. URL <http://www.geocities.jp/kosako3/oniguruma/>.
- [20] Adrian Thurston. Ragel state machine compiler, 2015. URL <http://www.colm.net/open-source/ragel/>.
- [21] Rajeev Alur, Adam Freilich, and Mukund Raghothaman. Regular combinators for string transformations. In *Proceedings of the Joint Meeting of the Twenty-Third EACSL Annual Conference on Computer Science Logic (CSL) and the Twenty-Ninth Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*, page 9. ACM, 2014.
- [22] Larry Wall, Tom Christiansen, and Jon Orwant. *Programming Perl*. O’Reilly, 3rd edition, July 2000.
- [23] Mark Lutz. *Programming Python*, volume 8. O’Reilly, 4th edition edition, December 2010. ISBN 978-0-596-15810-1.
- [24] Brent B Welch, Ken Jones, and Jeffrey Hobbs. *Practical programming in Tcl and Tk*. Prentice Hall, 4th edition edition, 2003. ISBN 0130385603.
- [25] ISO. *ISO 8601:2004. Data elements and interchange formats — Information interchange — Representation of dates and times*. ISO, 2004. URL http://www.iso.org/iso/catalogue_detail?csnumber=40874.
- [26] Jan Goyvaerts and Steven Levithan. *Regular Expressions Cookbook*. O’Reilly, 2009. ISBN 978-0-596-52068-7.

- [27] The Flex Project. flex: The Fast Lexical Analyzer, 2015. URL <http://flex.sourceforge.net/>.
- [28] Niklaus Wirth. *Algorithms + Data Structures = Programs*. Prentice-Hall, 1976. ISBN 0-13-022418-9.
- [29] Christophe Kalt. Rfc 2812 internet relay chat: Client protocol. *Request for Comments*, pages 1–63, 2000.
- [30] Rajeev Alur and Pavol Černý. Streaming transducers for algorithmic verification of single-pass list-processing programs. *ACM SIGPLAN Notices*, 46(1): 599–610, 2011.
- [31] S.M. Kearns. Extending regular expressions with context operators and parse extraction. *Software - Practice and Experience*, 21(8):787–804, 1991. doi: 10.1002/spe.4380210803.
- [32] A. Frisch and L. Cardelli. Greedy regular expression matching. In *Proc. 31st International Colloquium on Automata, Languages and Programming (ICALP)*, volume 3142 of *Lecture notes in computer science*, pages 618–629, Turku, Finland, July 2004. Springer.
- [33] Niels Bjørn Bugge Grathwohl, Fritz Henglein, Lasse Nielsen, and Ulrik Terp Rasmussen. Two-pass greedy regular expression parsing. In *Proc. 18th International Conference on Implementation and Application of Automata (CIAA)*, volume 7982 of *Lecture Notes in Computer Science (LNCS)*, pages 60–71. Springer, July 2013. doi: 10.1007/978-3-642-39274-0_7.
- [34] Rajeev Alur, Loris D’Antoni, Jyotirmoy Deshmukh, Mukund Raghothaman, and Yifei Yuan. Regular functions and cost register automata. In *Proceedings of the 2013 28th Annual ACM/IEEE Symposium on Logic in Computer Science*, pages 13–22. IEEE Computer Society, 2013.
- [35] Eric Allender and Ian Mertz. Complexity of regular functions. In *Proc. LATA*, 2015.
- [36] Nikolaj Bjørner and Margus Veanes. Symbolic transducers. Technical Report MSR-TR-2011-3, Microsoft Research, 2011.
- [37] Loris D’Antoni and Margus Veanes. Minimization of symbolic automata. In *Proceedings of the 41th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, San Diego, California, January 2014. ACM Press. doi: 10.1145/2535838.2535849.
- [38] Pieter Hooimeijer, Benjamin Livshits, David Molnar, Prateek Saxena, and Margus Veanes. Fast and precise sanitizer analysis with bek. In *Proceedings of the 20th USENIX conference on Security*, pages 1–1. USENIX Association, 2011.
- [39] Margus Veanes, David Molnar, Todd Mytkowicz, and Benjamin Livshits. Data-parallel string-manipulating programs. In *Proceedings of the 42nd annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. ACM Press, 2015.

- [40] Matteo Avalu, Fulvio Risso, and Riccardo Sisto. Efficient multistriding of large non-deterministic finite state automata for deep packet inspection. In *ICC*, pages 1079–1084. IEEE, 2012. ISBN 978-1-4577-2052-9. URL <http://dblp.uni-trier.de/db/conf/icc/icc2012.html#AvaluRS12>.

Appendix A

Appendix

```

data Table delta = Table { tblTable :: [[delta]], tblDigitSize :: Int }
    deriving (Eq, Ord, Show)

newtype ConstId = ConstId { getConstId :: Int } deriving (Eq, Ord, Show)
newtype BlockId = BlockId { getBlockId :: Int } deriving (Eq, Ord, Show)
newtype TableId = TableId { getTableId :: Int } deriving (Eq, Ord, Show)
newtype BufferId = BufferId { getBufferId :: Int } deriving (Eq, Ord, Show)

data Expr =
    SymE Int -- ^ next[i] (i less than value of AvailableSymbolsE)
  | AvailableSymbolsE -- ^ Number of available symbols
  | CompareE Int [Int] -- ^ Compare(&next[i], str, length(str))
  | ConstE Int
  | FalseE
  | TrueE
  | LteE Expr Expr
  | LtE Expr Expr
  | GteE Expr Expr
  | GtE Expr Expr
  | EqE Expr Expr
  | OrE Expr Expr
  | AndE Expr Expr
  | NotE Expr
    deriving (Eq, Ord, Show)

data Instr delta =
    AcceptI -- ^ accept (Program stops)
  | FailI -- ^ fail (Program stops)
  | AppendI BufferId ConstId -- ^ buf := buf ++ bs
  | AppendTblI BufferId TableId Int -- ^ buf := buf ++ tbl(id)(next[i])[0 .. sz(id) - 1]
  | AppendSymI BufferId Int -- ^ buf := buf ++ next[i]
  | ConcatI BufferId BufferId -- ^ buf1 := buf1 ++ buf2; reset(buf2)
  | ResetI BufferId -- ^ buf1 := []
  | AlignI BufferId BufferId -- ^ align buf1 buf2. assert that buf1 is empty, and
    -- make sure that subsequent writes to buf1 are aligned
    -- such that concatenating buf2 and buf1 with the current
    -- contents of buf2 will be efficient.
    -- This instruction is a hint to the runtime, and does not
    -- affect the final result.
  | IfI Expr (Block delta) -- ^ if (e :: Bool) { ... }
  | GotoI BlockId -- ^ goto b
  | NextI Int Int (Block delta) -- ^ if (!getChars(min,max)) { ... }
  | ConsumeI Int -- ^ next += i
  | ChangeOut BufferId -- ^ change the output buffer to buf, push old one to the stack
  | RestoreOut -- ^ pop to the previous output buffer
    deriving (Eq, Ord, Show)

type Block delta = [Instr delta]

data Program delta =
    Program
    { progTables :: M.Map TableId (Table delta)
    , progConstants :: M.Map ConstId [delta]
    , progStreamBuffer :: BufferId
    , progBuffers :: [BufferId]
    , progInitBlock :: BlockId
    , progBlocks :: M.Map BlockId (Block delta)
    }
    deriving (Eq, Show)

-- | A pipeline consisting of programs, either from fused automata or from pairs of
-- bitcode and action automata.
type Pipeline delta gamma = Either [Program delta] [(Program delta, Program gamma)]

```

Figure A.1: The intermediate language definition

```

start: syntax

syntax := "\\begin{KleenexVerb}\\n" (
  color_comment suppressed end
  | color_match match end
  | color_string constant end
  | comment
  | color_ident ident end
  | color_symbol symbol end
  | ignored
  | ws)* "\\end{KleenexVerb}\\n"

suppressed := tilde (matchingPar3 | ident | match)

matchingPar3 := /\(/ ([^()]/ | matchingPar2)* /\)/
matchingPar2 := /\(/ ([^()]/ | matchingPar1)* /\)/
matchingPar1 := /\(/ /^[^()]/ * /\)/

ignored := curlystart | curlyend | /\[\(\),:\[\]/
ident := (letter | /[0-9_]/)+
symbol := /<=|+=|:=|>|\*|\?|\+|!|@/
constant := dquote (/^[^x5C"'{}]/ | curlystart | curlyend
  | squote | color_escape escape end)* dquote
comment := onelineComment
match := /\(/ (color_escape escape end | curlystart
  | curlyend | /^[^x5C\\n]/)+ /\)/

escape := bs bs
  | bs /x[0-9a-fA-F]{2}/
  | bs (/[tnr.\][+?*()]/ | /\//)
  | bs dquote
  | bs curlystart

onelineComment :=
  color_comment /\// (/^[^n\x5C]/ | color_escape escape end)* end /\n/

letter := /[a-zA-Z]/
word := letter+
ws := /\t\r\n /

dquote := ~/" / "\\PYZdq{"
tilde := ~/~ / "\\PYZti{"
bs := ~/\ / "\\PYZbs{"
squote := ~/' / "\\PYZsq{"
curlystart := ~/{ / "\\PYZob{"
curlyend := ~}/ / "\\PYZcb{"

color_string := "\\PY{l+s}{"
color_comment := "\\PY{c+l}{"
color_match := "\\PY{l+s+sx}{"
color_ident := "\\PY{n+nf}{"
color_symbol := "\\PY{o}{"
color_escape := "\\PY{esc}{"

end := "}"

```

Figure A.2: The code used for syntax highlighting all Kleenex code in this paper.

Kleenex: Compiling Nondeterministic Transducers to Deterministic Streaming Transducers

Abstract

We present and illustrate Kleenex, a language for expressing general nondeterministic finite transducers, and its novel compilation to streaming string transducers with essentially optimal streaming behavior, worst-case linear-time performance and sustained high throughput. In use cases it achieves consistently high throughput rates around the 1 Gbps range on stock hardware, performing well, especially in complex use cases, in comparison to both specialized and related tools such as AWK, sed, grep, RE2, Ragel and regular-expression libraries.

Categories and Subject Descriptors D.3.1 [Formal Definitions and Theory]: Semantics; D.3.2 [Language Classifications]: Specialized application languages; F.1.1 [Models of Computation]: Automata

Keywords regular, automaton, nondeterministic, transducer, determinization, streaming

1. Introduction

Imagine you want to implement syntax highlighting. This can be thought of as parsing the input into its tokens and processing each token according to its class. For illustration, assume we have one keyword, `for`, and alphabetic identifiers as the only tokens. The lexical structure of the input is essentially described by the *regular expression (RE)* $((\text{for}|[a-z]^*))^*$, where whitespace is, for simplicity, represented by the single blank between the two closing parentheses. This scenario highlights the following:

Ambiguity by design. The RE is ambiguous. The intended semantics is that the left alternative has higher priority than the right. This is *greedy* disambiguation: Choose the left alternative if possible, treating e^* as its *unfolding* ee^* [1]. Accordingly, in our example `for` matches the left alternative, not the right.

Regular expression parsing. Note that the RE has star height 2; in particular, we need to *parse* the input under multiple Kleene stars. For our RE the parse of a string is a list of segments (corresponding to the outer Kleene star), with each segment represented by a pair, a token and whitespace (corresponding to concatenation), where each token is tagged (corresponding to the alternation) to indicate that it is either the keyword `for` or an

identifier; an identifier, in turn, consists of a list (corresponding to the inner Kleene star) of characters.

Output actions. We need to output something, the highlighted tokens, not just *accept* or *reject* a string as is done by finite *automata*. Note that output actions are not specified in our RE.

We would like to do the highlighting in a *streaming* fashion, using as little internal storage as possible and performing output actions as early as they are determined by the input prefix read so far, at a high sustained input processing rate, in particular in worst-case linear-time in the length of the input stream with a low factor depending linearly on the size of the RE. We would like to accomplish this automatically for *arbitrary* REs (or similar input format specification) and output actions, with speeds that in practice adapt to how much output actually needs to be produced; in particular, performance should gracefully approach pure acceptance testing as more and more output actions are removed. How?

It turns out that the set of parses are exactly the elements of the RE read as a *type* [23, 30]: Kleene-star is the (finite) list type constructor, concatenation the Cartesian product, alternation the sum type and an individual character the singleton type containing that character. A *Thompson automaton* [52] represents an RE in a strong sense: the complete paths—paths from initial to final state—are in one-to-one correspondence with the parses [27]. If a string has 4 parses (e.g. “for for”), then there are exactly 4 complete paths accepting it. Let us look at bit closer at a Thompson automaton: It is nondeterministic, with ϵ -transitions, easily constructed, having $O(m)$ states and transitions from an RE of size m . It has exactly one initial and one accepting node. Every state is either *nondeterministic*: it has two outgoing ϵ -transitions (“left” or “right”); or it is *deterministic*: it has exactly one outgoing transition labeled by ϵ or an input symbol, or it is the final state, which has no outgoing transition. Every complete path is determined by a sequence of bits used as an oracle [27]. Starting with the initial state, follow all outgoing transitions from deterministic states; upon arriving at a nondeterministic state query the oracle to determine whether to go left or right, until the final state is reached. The bit sequence of query responses yields a prefix-free binary code for the string accepted on the designated path. This *bit-code* can also be computed directly from the RE underlying the Thompson NFA [30, 40]. Since a bit-code represents a particular parse, a string can have multiple bit-codes if and only if the RE (and thus Thompson automaton) is ambiguous: The *greedy* parse of a string, which we are interested in, corresponds to the *lexicographically least* amongst its bit-codes [27].

The *greedy RE parsing problem* is producing this lexicographically least bit-code for a string matching a given RE. This can be done by an *optimally streaming* algorithm, running in time linear in the size of the input string for fixed RE [28]: The bits in the output are produced as soon as they are uniquely determined by the input prefix read so far, assuming the input string will eventually be accepted. The algorithm maintains an ordered *path tree* from the initial state to all the automata states reachable by the input prefix

read so far. A branching node represents both sides of an alternation that are both still viable. The (possibly empty) path segment from the initial state to the first branching node is what can be output based on the input prefix processed so far, without knowing which of the presently reached states will eventually accept the rest of the input. This works for all REs and all inputs; e.g., it automatically results in constant memory space consumption for REs that are deterministic modulo finite look-ahead, e.g. one-unambiguous REs [16].

Let us step back a bit. It is possible to aggressively (“earliest possible”) and efficiently stream out the bit-code of the greedy parse of an input string under a given RE as the input is streaming in: worst-case linear time in the input string size, no backtracking and each input symbol can be processed in time $O(m)$, linear in the size of the RE and of its Thompson NFA. (Here it is critical that Thompson NFAs have ϵ -transitions since equivalent ϵ -free automata require $\Omega(m \log m)$ transitions [43] and standard ϵ -free NFA-constructions [7, 24, 31] even $\Omega(m^2)$.)

Coming back to our syntax highlight problem we can use this algorithm to parse the input, build the parse tree from the bit-code and recursively descend it to perform the syntax highlighting. We might (correctly) suspect that the highlighting can be done by piping the bit-code into a separate highlighter process, eliding the materialization of the bit-code.¹ In this paper we show we can do better yet: The algorithm can be generalized to simulating arbitrary nondeterministic finite-state transducers, NFAs with output actions. Furthermore, we can compile their nondeterminism away by producing theoretically and practically very efficient streaming string transducers [3, 4, 6].

1.1 Contributions

This paper makes the following novel contributions:

- An aggressively *streaming* algorithm for *nondeterministic finite state transducers* (NFST) for ordered output alphabets, which emits the lexicographically least output sequence generated by all accepting paths of an input string. It runs in $O(mn)$ time, for automata of size m and inputs of size n .
- An effective determinization of NFSTs into a subclass of *streaming string transducers* (SST) [6], finite state machines with string registers that are updated linearly when entering the state upon reading an input symbol. The number of registers required adapts to the number of output actions in the NFST: The fewer output actions the fewer registers. In particular, without special-casing, no registers are generated—yielding a deterministic finite automata (DFA).
- An expressive declarative language, *Kleenex*, for specifying NFSTs with full support for and clear semantics of unrestricted nondeterminism by greedy disambiguation. A basic Kleenex program is a context-free grammar with embedded semantic output actions, but syntactically restricted to ensure that the input is regular.² Basic Kleenex programs can be functionally composed into pipelines. The central technical aspect of Kleenex is its semantic support for unbridled (regular) nondeterminism and its effective determinization and compilation to SSTs, thus both highlighting and complementing their significance.
- An implementation, including some empirically evaluated optimizations, of Kleenex that generates SSTs and sequential machines rendered as standard single-threaded C-code, which is eventually compiled to X86 machine code. The optimizations,

¹ All Kleenex code in this paper was highlighted with a Kleenex program emitting $\LaTeX$ -commands.

² This facilitates avoiding the $\Omega(M(n))$ lower bound for context-free grammar parsing, where $M(n)$ is the complexity of multiplying $n \times n$ matrices.

which are neither conclusive nor final, illustrate the design robustness obtained by the underlying theories of ordered NFST’s and SST’s.

- Use cases that illustrate the expressive power of Kleenex, and a performance comparison with related tools, including Ragel [53], RE2 [50] and specialized string processing tools. These document Kleenex’s consistently high performance (typically around 1 Gbps, single core, on stock hardware) even when compared to expressively more specialized tools with special-cased algorithms and tools with no or limited support for nondeterminism.

2. Transducers

The semantics of Kleenex will be given by translation to non-deterministic *finite state transducers*, which are finite automata extended with output in a free monoid. In this section, we will recall the standard definition (see e.g. Berstel [10]). Since Kleenex is deterministic, we also need to define a disambiguated semantics which allows us to interpret any non-deterministic transducer as a partial function, even when it may have more than one possible output for a given input string.

In the following, an *alphabet* is understood to be a finite subset $\{0, 1, \dots, n-1\} \subseteq \mathbb{N}$ of consecutive natural numbers with their usual ordering. We fix two alphabets Σ and Γ called the *input* and *output* alphabets, respectively.

Definition 1 (Finite State Transducer). A *finite state transducer* (FST) over Σ, Γ is a structure $\mathcal{T} = (\Sigma, \Gamma, Q, q^-, q^f, E)$ where

- Q is a finite set of *states*;
- $q^-, q^f \in Q$ are the *initial* and *final* states, respectively;
- $E : Q \times (\Sigma \cup \{\epsilon\}) \times (\Gamma \cup \{\epsilon\}) \times Q$ is the *transition relation*.

We write $q \xrightarrow{x|y} q'$ whenever $(q, x, y, q') \in E$. The *support* of $q \in Q$ is defined as $\text{supp}(q) = \{x \in \Sigma \cup \{\epsilon\} \mid \exists q', v. q \xrightarrow{x|v} q'\}$.

A *path* in $\mathcal{T}$ is a sequence of transitions

$$q_0 \xrightarrow{x_1|y_1} q_1 \xrightarrow{x_2|y_2} \dots \xrightarrow{x_n|y_n} q_n$$

It has *input label* $u = x_1x_2\dots x_n$ and *output label* $v = y_1y_2\dots y_n$ (ϵ denotes the empty string). We write $q_0 \xrightarrow{u|v} q_n$ if a path from q_0 to q_n with input label u and output label v exists.

$\mathcal{T}$ is *normalized* if for every state $q \in Q$, either $\text{supp}(q) = \{\epsilon\}$ or $\text{supp}(q) \subseteq \Sigma$; and furthermore $\text{supp}(q^f) \subseteq \Sigma$. We write $q \downarrow$ for q such that $\text{supp}(q) \subseteq \Sigma$. The formulation of our simulation algorithm in Section 4 becomes simpler when restricting our attention to normalized transducers, since we can take advantage of the following separation property:

Proposition 1. If $\mathcal{T}$ is normalized, then $p \xrightarrow{u|v} r \downarrow$ if and only if there exists a q such that $z = xy$ and $p \xrightarrow{u|x} q \downarrow \xrightarrow{v|y} r \downarrow$.

Definition 2 (Relational Semantics). An FST $\mathcal{T}$ denotes a relation $\llbracket \mathcal{T} \rrbracket \subseteq \Sigma^* \times \Gamma^*$ with $(u, v) \in \llbracket \mathcal{T} \rrbracket$ iff $q^- \xrightarrow{u|v} q^f$.

The relations definable as FSTs are the *rational relations* [10]. In the special case where for any $u \in \Sigma^*$ there is at most one $v \in \Gamma^*$ such that $(u, v) \in \llbracket \mathcal{T} \rrbracket$, the transducer computes a partial function. Any FST can be translated to an equivalent normalized FST.

In the following we give a refined semantics which allows us to interpret any FST as denoting a partial function, using the assumed ordering on alphabets to disambiguate between outputs. Our semantics requires restricting paths to be nonproblematic [23]:

If a path contains a non-empty loop $q' \xrightarrow{\epsilon|v'} q'$ with empty input label, then the path is said to be *problematic*, otherwise it is

nonproblematic. If there is a nonproblematic path from q to q' with labels u, v , then we write a subscript on the arrow: $q \xrightarrow{u|v}_{np} q'$.

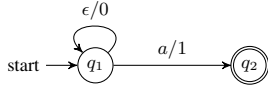
The output words (elements of Γ^*) are lexicographically ordered: $w_1 \leq w_2$ if either w_1 is a prefix of w_2 , or there exist words w'_1, w'_2 and symbols $b_1, b_2 \in \Gamma$ such that $w_1 = w'_1 b_1$, $w_2 = w'_2 b_2$ and $b_1 < b_2$. We use the ordering on output words to choose a single path from a non-empty set of paths:

Definition 3 (Functional Semantics). Any transducer $\mathcal{T}$ denotes a partial function $\llbracket \mathcal{T} \rrbracket_{\leq} : \Sigma^* \rightarrow \Gamma^* \cup \{\emptyset\}$ where

$$\llbracket \mathcal{T} \rrbracket_{\leq}(u) = \min\{v \mid q^- \xrightarrow{u|v}_{np} q^f\}.$$

Note that a partial function $A \rightarrow B \cup \{\emptyset\}$ is considered here to be a map $A \rightarrow 2^B$ where the cardinality of every subset in its range is at most two, and we tacitly identify elements $a \in A$ with their singleton sets $\{a\}$.³

Why the restriction to nonproblematic paths? Consider the following transducer $\mathcal{T}$:



Then $\min\{v \mid q_1 \xrightarrow{a|v} q_2\} = \emptyset$, as evidenced by the following infinitely descending chain of outputs: $1 \geq 01 \geq 001 \geq 0001 \geq \dots$. Operationally, such a chain corresponds to a non-terminating backtracking search. On the other hand, the number of nonproblematic paths with a given input label is always finite, ensuring well-foundedness of the lexicographic order. Every problematic path has a corresponding nonproblematic path with the same input label; consequently, $\text{dom}(\llbracket \mathcal{T} \rrbracket_{\leq}) = \text{dom}(\llbracket \mathcal{T} \rrbracket)$.

3. Kleenex

The core syntax of Kleenex is essentially that of right regular grammars extended with *output actions* and choice operators. Semantically, a Kleenex program denotes a function which transforms an input string from a regular language into a sequence of action symbols, with the caveat that if the input grammar is ambiguous, then the production rules are chosen according to a greedy leftmost disambiguation strategy.

We will first present the abstract syntax of core Kleenex, which is given a semantics in terms of the transducers introduced in Section 2.

Definition 4 (Kleenex syntax). Let Σ and Γ be two alphabets. A Kleenex program is a non-empty list $p = d_0 d_1 \dots d_n$ of *definitions* d_i , each of the form $N_i := t_i$, where t_i is a *term* generated by the grammar:

$$t ::= 1 \mid N \mid a t \mid "b" t \mid t_0 t_1$$

In the above, N is assumed to range over some set of non-terminal identifiers $\{N_1, \dots, N_n\}$, $a \in \Sigma$ over *input symbols* and $b \in \Gamma$ over *output actions*.

We restrict the valid Kleenex programs to those where there is at most one definition for each non-terminal identifier.

Let p be a Kleenex program over non-terminals $\{N_1, \dots, N_n\}$. We define a set of states Q_p and two transition relations E_p^A, E_p^C as the smallest sets closed under the following rules:

$$\frac{N_i \in Q_p}{N_i \in Q_p} \quad \frac{N_i \in Q_p}{t_i \in Q_p} \quad \frac{N_i \in Q_p}{(N_i, \epsilon, \epsilon, t_i) \in E_p^A \cap E_p^C} (N_i := t_i)$$

³ In other words, we adjoin an element to model partial functions as total functions to *pointed sets*.

```

main := odd ~ / a /      N_main := N_odd | N_even
      | even ~ / a /      N_odd := a a "b" "b" N_odd
odd   := ~ / a a / "bb" odd | "c" a 1
even  := ~ / a / "c" even | "b" a 1

```

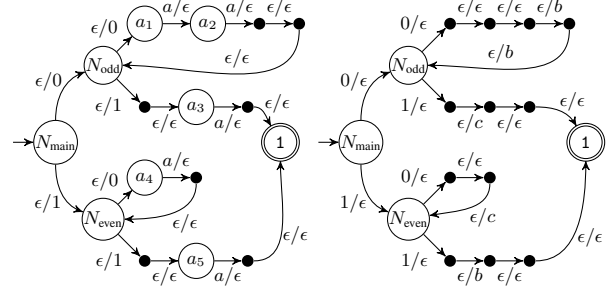


Figure 1: In the top left is a Kleenex program in the surface syntax and on the right is the desugared version. Below, the oracle transducer and action machine is shown, from left to right. The transduction realized by the program maps a^{2n+1} to $b^{2n}c$, and a^{2n+2} to $c^{2n}b$, respectively.

$$\frac{a t \in Q_p}{t \in Q_p \quad (a t, a, \epsilon, t) \in E_p^C \quad (a t, \epsilon, \epsilon, t) \in E_p^A}$$

$$\frac{"b" t \in Q_p}{t \in Q_p \quad ("b" t, \epsilon, \epsilon, t) \in E_p^C \quad ("b" t, \epsilon, b, t) \in E_p^A}$$

$$\frac{t_0 t_1 \in Q_p}{\{t_0, t_1\} \subseteq Q_p \quad (t_0 t_1, \epsilon, 0, t_0), (t_0 t_1, \epsilon, 1, t_1) \in E_p^C \quad (t_0 t_1, 0, \epsilon, t_0), (t_0 t_1, 1, \epsilon, t_1) \in E_p^A}$$

The sets are easily seen to be finite. They define two transducers, an *oracle* $\mathcal{T}_p^C = (\Sigma, 2, Q_p, N_1, 1, E_p^C)$ and an *action machine* $\mathcal{T}_p^A = (2, \Gamma, Q_p, N_1, 1, E_p^A)$, where $\mathcal{T}_p^A$ is easily seen to be deterministic, and $\mathcal{T}_p^C$ is non-deterministic and possibly ambiguous. The oracle intuitively translates an input string to a set of codes for the possible paths through p which reads the given string. The action machine translates a code to a sequence of actions.

Disambiguating according to the greedy leftmost strategy corresponds to choosing the lexicographically least code, and we can thus define the semantics as follows:

Definition 5 (Kleenex semantics). Let p be a Kleenex program and let $\mathcal{T}_p^C$ and $\mathcal{T}_p^A$ be defined as above. The program p denotes a partial function $\llbracket p \rrbracket : \Sigma^* \rightarrow \Gamma^* \cup \{\emptyset\}$ given by

$$\llbracket p \rrbracket = \llbracket \mathcal{T}_p^A \rrbracket \circ \llbracket \mathcal{T}_p^C \rrbracket_{\leq}$$

3.1 Syntactic sugar

The full syntax of our language is obtained by extending the syntax of core Kleenex with the following term-level constructors:

$$t ::= \dots \mid "v" \mid /e/ \mid \sim t \mid t_0 \cdot t_1 \mid t* \mid t+ \mid t?$$

$$\mid t\{n\} \mid t\{n, \dots\} \mid t\{m\} \mid t\{n, m\}$$

where $v \in \Gamma^*$, $n, m \in \mathbb{N}$, and e is a *regular expression*. The term $"v"$ is just shorthand for a sequence of action symbols. The regular expressions are special versions of Kleenex terms that do not contain identifiers. They always desugar to terms that output the matched input string: The sugared term $/e/$ adds a default action $"\alpha(a)"$ after every input symbol a in

e using a given default action map $\alpha : \Sigma \rightarrow \Gamma$. For example, the regular expression `/a*|b{n,m}|c?/` becomes the term $(a^*a) * | (b^n b) \{n, m\} | (c^*c) ?$. A *suppressed* subterm is written $\sim t$, and it desugars into t with all action symbols removed. Composition $t_0 \cdot t_1$ allows general sequential composition instead of the strict cons-like form of the core syntax. The operators $\cdot^*$, $\cdot^+$ and $\cdot^?$ desugar to their usual meaning as regular operators, as do operators $\cdot\{n\}$, $\cdot\{n, \}$, $\cdot\{, m\}$, and $\cdot\{n, m\}$.

By convention, the nonterminal named `main` is the entry point to a Kleenex program.

The desugaring can be described more precisely by a desugaring operator $\mathcal{D}[\cdot, \cdot]$. The result of desugaring a program $p = d_1 \dots d_n$ with initial term $N_1 := t_1$ is a program with initial term $N'_1 := \mathcal{D}[t_1, 1]$ which furthermore is a solution to the following set of equations:

$$\begin{aligned} \mathcal{D}[1, k] &= \mathcal{D}[\sim 1, k] = k \\ \mathcal{D}["b_1 \dots b_n", k] &= "b_1" \dots "b_n" k \\ \mathcal{D}[\sim("b" t), k] &= \mathcal{D}[\sim t, k] \\ \mathcal{D}[a t, k] &= a \mathcal{D}[t, k] \\ \mathcal{D}[\sim(a t), k] &= a \mathcal{D}[\sim t, k] \\ \mathcal{D}[\sim(t_0 | t_1), k] &= \mathcal{D}[\sim t_0, k] | \mathcal{D}[\sim t_1, k] \\ \mathcal{D}[N, k] &= N_{\mathcal{D}[t, k]} \quad (\text{where } N := t) \\ \mathcal{D}[\sim N, k] &= N_{\mathcal{D}[\sim t, k]} \quad (\text{where } N := t) \\ \mathcal{D}[e/, k] &= \mathcal{D}[t_e, k] \\ \mathcal{D}[t_0 \cdot t_1, k] &= \mathcal{D}[t_0, \mathcal{D}[t_1, k]] \\ \mathcal{D}[t_0 | t_1, k] &= \mathcal{D}[t_0, k] | \mathcal{D}[t_1, k] \\ \mathcal{D}[t^*, k] &= \mathcal{D}[t, \mathcal{D}[t^*, k]] | k \\ \mathcal{D}[t^+, k] &= \mathcal{D}[t, \mathcal{D}[t^+, k]] \\ \mathcal{D}[t^?, k] &= \mathcal{D}[t, k] | k \end{aligned}$$

In the above, a non-terminal name N_t on the right-hand side of an equation implies the presence of a definition $N_t := t$, and the term t_e corresponds to the regular expression e as described above. The range patterns are just expanded and then further desugared.

The system does not always have a well-defined solution: The generalized composition operator of sugared Kleenex allows one to write non-regular grammars, for example:

$$A := (aA) \cdot b | 1.$$

A program that does not have a well-defined desugaring is not considered to be well-formed, and we will not attempt to give it a semantics.

3.2 Custom register updates

We extend the syntax of Kleenex further with *register actions*:

$$\begin{aligned} t ::= & \dots \mid R @ t \mid !R \\ & \mid [R \leftarrow (R \mid "v")^*] \\ & \mid [R += (R \mid "v")^*] \end{aligned}$$

where R is a lower-case register name. Intuitively, these constructs allow one to store actions and perform them later. Writing $R @ t$ redirects all actions that would have resulted from running t into the register R , which can be performed later by writing $!R$. The register R can be either set to a sequence of actions $(R \mid "v")^*$ or appended with them, using the $\leftarrow$ and $+=$ construct, respectively.

At first glance it seems like adding custom register updates to Kleenex significantly alters the language and moves beyond the semantics discussed so far. However, the only requirement on the output actions is that they form a monoid, so this is not the case. We simply add actions like “set register R to v ” as output symbols

along with the output symbols $!v$. The output redirection caused by the $\cdot @ \cdot$ operator can be understood as a push operation: when $R @ t$ is written it means that in the scope of t the topmost register is R . If there are other redirection symbols in t , these will come in and out of scope as they are pushed and popped to the stack.

As an example, the following program swaps two input lines by storing them in registers `a` and `b` and outputting them in reverse order:

```
main := a@line b@line !b !a
line := /[^\n]*\n/
```

4. Simulation and determinization

In this section, we specify an algorithm for simulating FSTs under the functional semantics, allowing us to efficiently evaluate the oracle transducer defined in Section 3. We also show how the simulation algorithm can be implemented by finite deterministic *streaming string transducers* [3] whose states are identified by equivalence classes of simulation states. The latter construction gives a deterministic machine model for Kleenex programs which can be compiled to efficient code for executing on hardware.

We note that non-deterministic transducers are strictly more powerful than their deterministic counterparts, and can thus not always be determinized in general. Determinization procedures exist [9, 54] which result in a deterministic transducer with an infinite state set in the general case, and a finite state set if and only if the underlying transduction is *subsequential* [10, 44]. The oracle transducers of Kleenex programs are not subsequential in general. Our simulation algorithm is also different from the existing methods for determinizing transducers by also taking disambiguation into account.

In the following we fix a transducer $\mathcal{T} = (\Sigma, \Delta, Q, q^-, q^f, E)$. We will assume that $\mathcal{T}$ is normalized, and that it furthermore satisfies the following property:

Definition 6 (Prefix-free transducer). $\mathcal{T}$ is said to be *prefix-free* if for all $p, q, q' \in Q_{\mathcal{T}}$ where $\text{supp}(q), \text{supp}(q') \subseteq \Sigma$ we have that if $p \xrightarrow{x|y} q$ and $p \xrightarrow{x|y'} q'$ then $y \neq y'$.

It is easy to verify that the oracle transducers constructed Section 3 are both normalized and prefix-free. Note that they will always have $\Delta = \mathbb{2}$, but our construction generalizes to oracle alphabets of all sizes.

4.1 Generalized state set simulation

Let D be a finite and totally ordered set, and write $S(D, Q)$ for the set of partial functions $Q \rightarrow D^* \cup \{\emptyset\}$. Elements $A \in S(D, Q)$ can be seen as generalized subsets of Q where every member q is labeled by some element $A(q) \in D^*$, and every non-member has $A(q) = \emptyset$.

We extend word concatenation in D^* to the set $D^* \cup \{\emptyset\}$ by setting $x\emptyset = \emptyset x$. For $u, v \in D^* \cup \{\emptyset\}$, write $u \preceq v$ if u is a *prefix* of v , i.e. there is a unique w such that $v = uw$. Write $u \prec v$ if w has length at least one. Let $u \wedge v$ refer to the longest p such that $u = pv'$ and $v = pv''$ for some v', v'' . Note that in view of this definition, $\emptyset$ becomes a neutral element with $u \wedge \emptyset = u = \emptyset \wedge u$.

We define a right action $\cdot : S(D, Q) \times \Sigma^* \rightarrow S(D \cup \Delta, Q)$ on the generalized state sets as follows:

Definition 7 (Right action). Let $A \in S(D, Q)$ and $u \in \Sigma^*$. We define

$$(A \cdot u)(q) = \min\{A(p)v \mid p \xrightarrow{u|v}_{np} q \downarrow\}.$$

When $D = \Delta$ the right action can be seen as a map $S(\Delta, Q) \times \Sigma^* \rightarrow S(\Delta, Q)$. It is easily seen that the right action is related to the functional semantics in the following way:

Proposition 2. Let $A(q) = \epsilon$ if $q = q^-$ and $A(q) = \emptyset$ otherwise. Then $(A \cdot u)(q^f) = \llbracket \mathcal{T} \rrbracket_{\leq}(u)$.

A generalized subset $A \in S(D, Q)$ is said to be *prefix-free* if $A(p) \not\prec A(q)$ for all $p, q \in Q$. When $\mathcal{T}$ is normalized and prefix-free, the right action preserves prefix-freeness of generalized subsets and commutes with word concatenation:

Proposition 3. If $\mathcal{T}$ is normalized and prefix-free and A is prefix-free, then for all $u, v \in \Sigma^*$,

1. $A \cdot u$ is prefix-free; and
2. $(A \cdot u) \cdot v = A \cdot uv$.

Proof. The first property follows directly by A and $\mathcal{T}$ being prefix-free. For the second, we have for $r \in Q$,

$$\begin{aligned} & ((A \cdot u) \cdot v)(r) \\ &= \min\{(A \cdot u)(q)y \mid q \xrightarrow{v|y}_{np} r \downarrow\} \\ &= \min\{\min\{A(p)x \mid p \xrightarrow{u|x}_{np} q \downarrow\}y \mid q \xrightarrow{v|y}_{np} r \downarrow\} \\ &= \min\{\min\{A(p)xy \mid p \xrightarrow{u|x}_{np} q \downarrow\} \mid q \xrightarrow{v|y}_{np} r \downarrow\} \\ &= \min\{A(p)xy \mid p \xrightarrow{u|x}_{np} q \downarrow \xrightarrow{v|y}_{np} r \downarrow\} \\ &= \min\{A(p)z \mid p \xrightarrow{uv|z}_{np} r \downarrow\} = (A \cdot uv)(r) \end{aligned}$$

The third equality is a consequence of the fact that A and $\mathcal{T}$ are prefix-free, together with the following easily proved fact about lexicographic ordering: $(\min X)y = \min\{xy \mid x \in X\}$ whenever X is a set of pairwise prefix-free words. The fourth equality is just associativity of minimum, and the last equality follows by the fact that $\mathcal{T}$ is normalized and Proposition 1. $\square$

For $x \in D^*$ and $A \in S(D, Q)$, define $xA \in S(D, Q)$ by $(xA)(q) = x(A(q))$. We say that x is a prefix of A if $A = xA'$ for some A' , which is equivalent to x being a prefix of every $A(q)$. The right action commutes with the prefix operation:

Proposition 4. Let $x \in D^*$, then $(xA) \cdot u = x(A \cdot u)$ for all $u \in \Sigma^*$.

Proof. Follows by the fact that lexicographic ordering satisfies $\min\{xy \mid y \in Y\} = x \min Y$. $\square$

Streaming simulation algorithm A streaming simulation algorithm on $\mathcal{T}$ processes an input from left to right and may write zero or more symbols to the output in each step.

Algorithm 1 (Streaming FST Simulation). Let $\mathcal{T}$ be a normalized and prefix-free transducer, and let the input $u = a_1a_2\dots a_n$ be given. Let $A_0 \in S(\Delta, Q)$ be defined as in Proposition 2. Reading symbol a_i , compute $B_i = A_i \cdot a_{i+1}$. Append $p_i = \bigwedge_{q \in Q} B_i(q)$ to the output stream and set $A_{i+1} = B'_i$, where the equality $B_i = p_i B'_i$ defines B'_i . When there are no more input symbols left, append $(A_n \cdot \epsilon)(q^f)$ to the output and return, or fail if $(A_n \cdot \epsilon)(q^f) = \emptyset$.

By Proposition 2, Proposition 3 and Proposition 4, the algorithm computes $\llbracket \mathcal{T} \rrbracket_{\leq}(u)$.

4.2 A deterministic computation model

We wish to translate Kleenex programs to completely deterministic programs without a simulation overhead.

Single-valued transducers in general, however, can only be determined if the underlying function is *subsequential* [10, 44], a property which is not satisfied in general for the oracle transducers constructed from Kleenex programs.

We turn instead to *streaming string transducers* [3] (SST), a deterministic model of computation which generalizes subsequential transducers by allowing copy-free updates to a finite set of word registers. It turns out that every transducer that can be simulated by our generalized state set algorithm can be expressed as an SST.

Definition 8 (Streaming String Transducer [3]). A deterministic *streaming string transducer* (SST) over alphabets Σ, Δ is a structure $\mathcal{S} = (X, Q, q^-, F, \delta^1, \delta^2)$ where

- X is a finite set of *register variables*;
- Q is a finite set of *states*;
- F is a partial function $Q \rightarrow (\Delta \cup X)^* \cup \{\emptyset\}$ mapping each *final state* $q \in \text{dom}(F)$ to a word $F(q) \in (\Delta \cup X)^*$ such that each $x \in X$ occurs at most once in $F(q)$;
- δ^1 is a transition function $Q \times \Sigma \rightarrow Q$;
- δ^2 is a *register update* function $Q \times \Sigma \times X \rightarrow (\Delta \cup X)^*$ such that for each $q \in Q, a \in \Sigma$ and $x \in X$, there is at most one $y \in X$ such that x occurs in $\delta^2(q, a, y)$.

The semantics are defined as follows. A *configuration* of an SST $\mathcal{S}$ is a pair (q, ρ) where $q \in Q_{\mathcal{S}}$ is a state, and $\rho : X_{\mathcal{S}} \rightarrow \Delta^*$ is a *valuation*. A valuation extends as a monoid homomorphism to a map $\hat{\rho} : (X_{\mathcal{S}} \cup \Delta)^* \rightarrow \Delta^*$ by setting $\rho(x) = x$ for $x \in \Delta$. The initial configuration is (q^-, ρ^-) where $\rho^-(x) = \epsilon$ for all $x \in X_{\mathcal{S}}$.

A configuration steps to a new configuration given an input symbol: $\delta_{\mathcal{S}}((q, \rho), a) = (\delta_{\mathcal{S}}^1(q, a), \rho')$, where $\rho'(x) = \hat{\rho}(\delta_{\mathcal{S}}^2(q, a, x))$. The transition function extends to a transition function on words $\delta_{\mathcal{S}}^*$ by $\delta_{\mathcal{S}}^*((q, \rho), \epsilon) = (q, \rho)$ and $\delta_{\mathcal{S}}^*((q, \rho), au) = \delta_{\mathcal{S}}^*(\delta_{\mathcal{S}}((q, \rho), a), u)$.

Every SST $\mathcal{S}$ denotes a partial function $\llbracket \mathcal{S} \rrbracket : \Sigma^* \rightarrow \Delta^* \cup \{\emptyset\}$ where for any $u \in \Sigma^*$, we define

$$\llbracket \mathcal{S} \rrbracket(u) = \begin{cases} \hat{\rho}'(F_{\mathcal{S}}(q')) & \text{if } \delta^*((q^-, \rho^-), u) = (q', \rho') \\ & \text{and } q' \in \text{dom}(F_{\mathcal{S}}) \\ \text{undefined} & \text{otherwise} \end{cases}$$

4.3 Tabulation

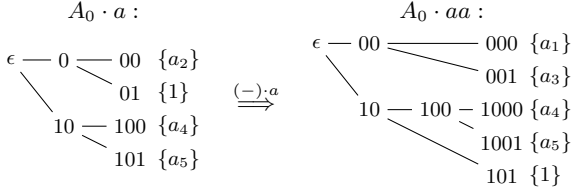
We need to come up with a representation of our streaming simulation algorithm as an SST with a designated register used for streaming output. Our representation needs to satisfy the property of being finite state as well as the property that the output register contains the output $p_1p_2\dots p_i$ of Algorithm 1 after reading input symbol a_i . The latter requirement means that we must somehow statically encode the prefix structure of all potential outputs in the states of the SST, since SSTs cannot access the contents of registers. It turns out that this is possible by letting the states of the SST be equivalence classes of generalized state sets, where the equivalence relates state sets that agree on state ordering and prefix structure.

Trees We will call a prefix-free generalized state set A an *ordered tree* with node set

$$N_A = \{A(p) \wedge A(q) \mid p, q \in Q, A(p) \wedge A(q) \neq \emptyset\}.$$

Under this view, the leaves of A seen as a tree is the subset of nodes $L_A = \{A(q) \mid q \in Q, A(q) \neq \emptyset\} \subseteq N_A$, and the leaves are labeled by $A^{-1} : L_A \rightarrow 2^Q$. Since A is assumed prefix-free, we have for any nodes $x, y \in N_A$ that $x \preceq y$ if and only if there is a $z \in N_A$ such that $x = y \wedge z$. In this case x is called an *ancestor* of y and z , which in turn are called the *descendants* of x . Importantly, the root node of any (sub)tree is the longest common prefix of its descendants.

Example 1. We illustrate the tree interpretation as follows. Consider the oracle transducer from Figure 1. Let A_0 be the generalized state set that maps N_{main} to ϵ and every other state to $\emptyset$. Then the state sets $A_0 \cdot a$ and $A_0 \cdot aa$ can be seen as trees in the following way:



We will consider two generalized state sets to be equivalent if they are indistinguishable as ordered trees.

Definition 9 (Ordered tree isomorphism). Let D_1, D_2 be totally ordered and let $A_1 \in S(D_1, Q)$ and $A_2 \in S(D_2, Q)$ be trees. An ordered tree isomorphism between A_1 and A_2 is a bijective map $h : N_{A_1} \rightarrow N_{A_2}$ such that for all $p, q \in Q$:

1. $h(A_1(p) \wedge A_1(q)) = A_2(p) \wedge A_2(q)$; and
2. $A_1(p) \leq A_1(q)$ if and only if $A_2(p) \leq A_2(q)$.

We write $h : A_1 \equiv A_2$ and say that A_1 and A_2 are *equivalent* when h is an ordered tree isomorphism between A_1 and A_2 . Tree equivalence is preserved by the right action:

Proposition 5. If $A \in (D_1, Q), B \in (D_2, Q)$ and $h : A \equiv B$ then for all $a \in \Sigma$, we have $A \cdot a \equiv B \cdot a$.

Proof sketch. Since h is an order isomorphism and since A and B are prefix-free, we have for all $q \in Q$ exists $p_q \in Q$ and $y_q \in \Delta^*$ such that $(A \cdot a)(q) = A(p_q)y_q$ and $(B \cdot a)(q) = h(A(p_q))y_q$. Observe that for any $n \in N_{A \cdot a}$ there exists $q_1, q_2 \in Q$ such that

$$\begin{aligned} n &= (A \cdot a)(q_1) \wedge (A \cdot a)(q_2) \\ &= \begin{cases} A(q_1)(y_{q_1} \wedge y_{q_2}) & \text{if } A(q_1) = A(q_2) \\ A(q_1) \wedge A(q_2) & \text{otherwise} \end{cases} \end{aligned}$$

Furthermore, there does not exist $q_1, q_2, r_1, r_2 \in Q$ such that $A(q_1)(y_{q_1} \wedge y_{q_2}) = A(r_1) \wedge A(r_2)$, since that would imply that $A(q_1)$ is a prefix of $A(r_1)$ and $A(r_2)$. We define a map $h' : N_{A \cdot a} \rightarrow N_{B \cdot a}$ such that for all $q_1, q_2 \in Q$,

$$\begin{aligned} h'((A \cdot a)(q_1) \wedge (A \cdot a)(q_2)) &= \begin{cases} h(A(q_1)(y_{q_1} \wedge y_{q_2})) & \text{if } A(q_1) = A(q_2) \\ h(A(q_1) \wedge A(q_2)) & \text{otherwise.} \end{cases} \end{aligned}$$

This is a well-defined function by the previous observations, and a tree isomorphism by the fact that h is a tree isomorphism. $\square$

Canonical representatives Call a generalized set $A \in S(D, Q)$ *canonical* if

1. $\text{rng}(A)$ is prefix closed: if $y \in \text{rng}(A)$ and $x \preceq y$ then $x \in \text{rng}(A)$; and
2. $\text{rng}(A)$ is downwards closed: if $x b \in \text{rng}(A)$ for $b' < b$ then $x b' \in \text{rng}(A)$ (for $b, b' \in \Delta$).

Write $\tilde{S}(D, Q)$ for the subset of canonical trees. The set is finite, as every canonical tree A has a prefix closed node set, so the longest word in N_A is bounded by $|\text{dom}(A)| - 1$ (the maximum depth of a tree with $|\text{dom}(A)|$ leaves).

Any tree has a canonical representative:

Proposition 6. For any set D and tree $A \in S(D, Q)$, there is a unique $C \in \tilde{S}(N, Q)$ with $A \equiv C$.

As a consequence, there is a reduction map $[\cdot] : S(D, Q) \rightarrow \tilde{S}(N, Q)$ such that $A \equiv B$ if and only if $[A] = [B]$, implying that the quotient set $S(D, Q)/\equiv$ must be finite. Any $A \in S(D, Q)$ is

thus canonically represented by a homomorphism $h_A : N_{[A]} \rightarrow N_A$ such that $A = h_A \circ [A]$.

In view of Proposition 5, this means that we can statically enumerate all possible trees up to tree isomorphism by computing with the canonical representatives. Any concrete tree reachable by the simulation algorithm is an instance of a canonical tree composed with a suitable homomorphism. An SST implementing the simulation algorithm can thus take the set of canonical trees as its states, and will then need to maintain the associated homomorphism via register updates.

Paths We need to represent tree homomorphisms using SST registers such that the effect of computing right actions on the underlying tree can be expressed as SST updates.

For a tree $A \in S(D, Q)$, any node $x \in N_A$ has a unique maximal decomposition $x = x_0 x_1 \dots x_n$ such that each $x_0 x_1 \dots x_i \in N_A$ for all $0 \leq i \leq n$. Intuitively, this reflects the full path from the root node to the node x , and we can define the map

$$\begin{aligned} \text{path}_A : N_A &\rightarrow N_A^* \\ \text{path}_A(x) &= (x_0, x_0 x_1, \dots, x_0 x_1 \dots x_n), \end{aligned}$$

which maps nodes to their maximal path decomposition (we use the tuple notation to distinguish between the two levels of monoids). In view of this and the fact that homomorphisms must preserve descendants, then for any homomorphism $h : A \equiv B$ there is a unique $\kappa_h : N_A \rightarrow N_B$ such that

$$h(x) = \kappa_h(t_0) \kappa_h(t_1) \dots \kappa_h(t_n), \quad (1)$$

where $\text{path}_A(x) = (t_0, t_1, \dots, t_n)$. Intuitively, κ can be seen as a “differential” representation of h , representing the change of h between a node and its immediate ancestor. By viewing κ_h as a map $N_A \rightarrow D_B^*$ which extends uniquely to a monoid homomorphism $\widehat{\kappa}_h : N_A^* \rightarrow D_B^*$, we obtain $h = \widehat{\kappa}_h \circ \text{path}_A$. Considering the unique isomorphism $h_A : [A] \equiv A$, write κ_A for the associated decomposition satisfying (1), and we thus have

$$A = \widehat{\kappa}_A \circ \text{path}_{[A]} \circ [A] \quad (2)$$

The path-operator is easily seen to be a tree isomorphism since it preserves node ordering and prefix structure. That is, for any $A \in S(D, Q)$, we have $\text{path}_A : A \equiv A^\#$ where $A^\# \in S(N_D, Q)$ is defined by $A^\# = \text{path}_A \circ A$. Using this notation, (2) becomes

$$A = \widehat{\kappa}_A \circ [A]^\#, \quad (3)$$

SST construction We construct an SST implementing the FST simulation algorithm and sketch a proof of its correctness.

Theorem 1. For any normalized prefix-free transducer $\mathcal{T} = (\Sigma, \Delta, Q, q^-, q^f, E)$, there is an SST $\mathcal{S}$ such that $\llbracket S \rrbracket = \llbracket T \rrbracket_{\leq}$.

Proof. We define S as follows. Let A_0 be defined as in Algorithm 1, and observe that $A_0 \in \tilde{S}(N, Q)$. The states are the canonical trees labeled by Q :

$$\begin{aligned} Q_S &= \{[A] \mid A \in S(\Delta, Q)\} \cup \{A_0\} \subseteq \tilde{S}(N, Q), \\ q_S^-(q) &= A_0(q) \end{aligned}$$

The registers will be identified by canonical tree nodes:

$$X_S = \bigcup \{N_C \mid C \in Q_S\}.$$

The final output and the transition maps are given as follows:

$$\begin{aligned} F_S(C) &= (C^\# \cdot \epsilon)(q^f), \\ \delta_S^1(C, a) &= [C \cdot a], \\ \delta_S^2(C, a, x) &= \begin{cases} \kappa_{C^\# \cdot a}(x) & \text{if } x \in N_{[C^\# \cdot a]} \\ \epsilon & \text{otherwise} \end{cases} \end{aligned}$$

We claim that $\mathcal{S}$ computes the same function as $\mathcal{T}$ under the functional semantics.

For $u \in \Sigma^*$ let (C_u, ρ_u) refer to the value $\delta_S^*((q_S^-, \rho^-), u) = (C_i, \rho_i)$. We show that for any $u \in \Sigma^*$, we have $\widehat{\rho}_u \circ (C_u^\# \cdot \epsilon) = A_0 \cdot u$.

Suppose that this holds. Then for any $u \in \Sigma^*$, we have by the above and Proposition 2 that $\llbracket S \rrbracket(u) = \widehat{\rho}_u(F_S(C_u)) = \widehat{\rho}_u \circ (C_u^\# \cdot \epsilon)(q^f) = (A_0 \cdot u)(q^f) = \llbracket \mathcal{T} \rrbracket_\leq(u)$.

Our claim follows as a special case of the following lemma. $\square$

Lemma 1. *Let $A \in S(\Delta, Q)$ and $\rho : X_S \rightarrow \Delta^*$ such that $A = \widehat{\rho} \circ [A]^\#$. Then for any $u \in \Sigma^+$ with $\delta_S^*([A], \rho, u) = (C, \rho')$ we have $\widehat{\rho}' \circ C^\# = A \cdot u$.*

Proof. By induction on u . For $u = a$ we have $C = [[A] \cdot a] = [A \cdot a]$ and $\rho' = \widehat{\rho} \circ \kappa_{[A]^\# \cdot a}$. We can easily verify that $\widehat{\rho}' = \widehat{\rho} \circ \kappa_{[A]^\# \cdot a}$ so for any $q \in Q$,

$$\begin{aligned} \widehat{\rho}' \circ [A \cdot a]^\#(q) &= \widehat{\rho} \circ \widehat{\kappa_{[A]^\# \cdot a}} \circ [A \cdot a]^\#(q) \\ &= \widehat{\rho} \circ ([A]^\# \cdot a)(q) \\ &= \widehat{\rho}(\min\{[A]^\#(p)y \mid p \xrightarrow{a|y}_{np} q \downarrow\}) \\ &= \min\{\widehat{\rho} \circ [A]^\#(p)y \mid p \xrightarrow{a|y}_{np} q \downarrow\} \\ &= \min\{A(p)y \mid p \xrightarrow{a|y}_{np} q \downarrow\} = (A \cdot a)(q) \end{aligned}$$

The second equality follows by observing that $A \equiv [A] \equiv [A]^\#$, so by Proposition 5, we have $A \cdot a \equiv [A]^\# \cdot a$ and thus $[A \cdot a] = [[A]^\# \cdot a]$. Therefore, $\widehat{\kappa_{[A]^\# \cdot a}} \circ [A \cdot a]^\# = \widehat{\kappa_{[A]^\# \cdot a}} \circ [[A]^\# \cdot a]^\# = [A]^\# \cdot a$ by using the identity (3). The fourth equality is justified by the fact that $[A]^\#(p) \leq [A]^\#(q)$ if and only if $A(p) \leq A(q)$.

For $u = au'$ where $u' \neq \epsilon$, we have $(C, \rho') = \delta_S^*([A \cdot a], \widehat{\rho} \circ \kappa_{[A]^\# \cdot a})$. By the previous argument we can apply the induction hypothesis, and we obtain $C = [(A \cdot a) \cdot u']$ and $\widehat{\rho}' \circ C^\# = (A \cdot a) \cdot u'$. The result then follows by Proposition 3. $\square$

Example 2. We illustrate how the construction works by showing how Example 1 is implemented as an SST update between states $[A_0 \cdot a]$ and $[A \cdot aa]$. The register update is obtained by computing $\kappa_{[A_0 \cdot a]^\# \cdot a}$. The tree $[A_0 \cdot a]^\#$ looks as follows:

$$\begin{aligned} (\bar{\epsilon}) - (\bar{\epsilon}, \bar{0}) &\xrightarrow{\quad} (\bar{\epsilon}, \bar{0}, \bar{00}) \{a_2\} \\ &\xrightarrow{\quad} (\bar{\epsilon}, \bar{0}, \bar{01}) \{1\} \\ (\bar{\epsilon}, \bar{1}) &\xrightarrow{\quad} (\bar{\epsilon}, \bar{1}, \bar{10}) \{a_4\} \\ &\xrightarrow{\quad} (\bar{\epsilon}, \bar{1}, \bar{11}) \{a_5\} \end{aligned}$$

Recall that each node is a full path in the canonical tree $[A_0 \cdot a]$. The node names from $N_{[A_0 \cdot a]}$ are overlined and elements of the path monoid $N_{[A_0 \cdot a]}^*$ is written $(x_1, x_2, \dots)$. The tree $[A_0 \cdot a]^\# \cdot a$ looks as follows:

$$\begin{aligned} (\bar{\epsilon}) &\xrightarrow{\quad} (\bar{\epsilon}, \bar{0}, \bar{00}) \xrightarrow{\quad} (\bar{\epsilon}, \bar{0}, \bar{00}, 0) \{a_1\} \\ &\xrightarrow{\quad} (\bar{\epsilon}, \bar{0}, \bar{00}, 1) \{a_3\} \\ (\bar{\epsilon}, \bar{1}) &\xrightarrow{\quad} (\bar{\epsilon}, \bar{1}, \bar{10}) \xrightarrow{\quad} (\bar{\epsilon}, \bar{1}, \bar{10}, 0) \{a_4\} \\ &\xrightarrow{\quad} (\bar{\epsilon}, \bar{1}, \bar{10}, 1) \{a_5\} \\ &\xrightarrow{\quad} (\bar{\epsilon}, \bar{1}, \bar{11}) \{1\} \end{aligned}$$

Note that symbols that are not overlined are output symbols from Δ . The map $\kappa' = \kappa_{[A_0 \cdot a]^\# \cdot a} : N_{[A_0 \cdot aa]} \rightarrow (N_{[A_0 \cdot a]} \cup \Delta)^*$ gives us the relevant SST update strings:

$$\begin{aligned} \kappa'(\bar{\epsilon}) &= (\bar{\epsilon}) & \kappa'(\bar{0}) &= (\bar{0}, \bar{00}) & \kappa'(\bar{00}) &= 0 \\ \kappa'(\bar{01}) &= 1 & \kappa'(\bar{1}) &= (\bar{1}) & \kappa'(\bar{10}) &= (\bar{10}) \\ \kappa'(\bar{100}) &= 0 & \kappa'(\bar{101}) &= 1 & \kappa'(\bar{11}) &= (\bar{11}) \end{aligned}$$

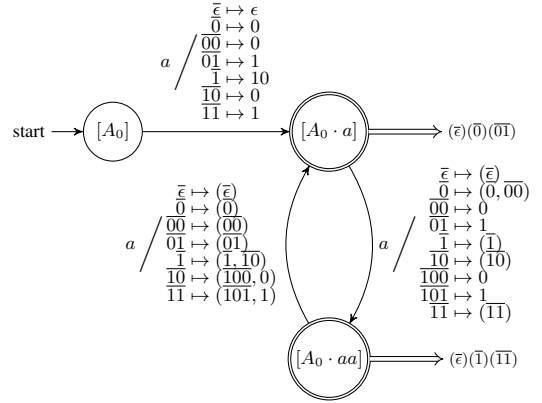


Figure 2: Example of SST computing the same function as the oracle transducer in Figure 1. Each transition is tagged by a register update, and the nodes of the canonical tree identifying the destination state make up the registers. The wide arrows exiting the accepting states indicate the final output string. Note that this always includes the root variable $(\bar{\epsilon})$ which thus acts as an interface for streaming output (although for this particular example, nothing can output until the end of the input).

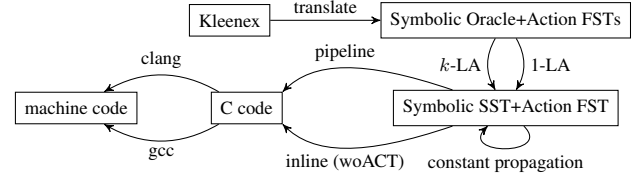


Figure 3: Compilation paths. 1-LA is symbolic SST construction with single-symbol transitions; k -LA is construction of SST with up to k symbols of lookahead for some k determined by the program. The “pipeline” translation path indicates that the resulting program keeps the oracle SST and action FST separate, with data being piped from the SST to the FST at runtime. The “inline” path indicates that the action FST is fused into the oracle SST.

The full construction of an SST from the oracle transducer in Figure 1 can be seen in Figure 2.

5. Implementation

Our implementation compiles a Kleenex program to machine code by implementing the transducer constructions described in the earlier sections. We have also implemented several optimizations to decrease the size of the generated SSTs and improve the performance of the generated code. We will briefly describe these in the following section, and we note that they are all orthogonal to the underlying principles behind our compilation.

The possible compilation paths of our implementation can be seen in Fig. 3.

5.1 Transducer pipeline

It is possible to chain together several Kleenex programs in a pipeline, letting the output of one serve as the input of the next. This can for example be used to strip unwanted characters before performing a transformation. By using the optional *pipeline pragma*, `start:  $t_1 \gg \dots \gg t_n$` , a programmer can specify that the entry point is t_1 and that the output should be chained together as specified, with the final output being that of t_n . The implementation

does this by spawning a process for each transducer and setting up UNIX pipes between them.

5.2 Inlining the action transducer

When we have constructed the oracle SST we end up with two deterministic machines which need to be composed. We can either do this at runtime, piping the output of the oracle SST into the action FST, or we can apply a form of deforestation to inline the output of the action FST directly in the SST (this is straightforward since the action FST is deterministic by construction). The former approach is advantageous if the Kleenex program produces a lot of output and is highly nondeterministic.

5.3 Constant propagation

The SSTs generated by our construction contains quite a lot of trivial register updates which can be eliminated in order to achieve better run-time efficiency. Consider the SST in Fig. 2, where all registers but $(\bar{0})$ and $(\bar{1})$ are easily seen to have a constant known value in each state. Eliminating the redundant registers means that we only have to maintain two registers at run-time.

We achieve this by constant propagation: computing reaching definitions by solving a set of data-flow constraints (see e.g. [8]).

5.4 Symbolic representation

Text transformation programs often contain idioms which have a rather redundant representation as pure transducers. A program might for example match against a whole range of characters and proceed in the same way regardless of which one was matched. This will, however, lead to a transition for each concrete character in the generated FST, even though all transitions have the same source and destination states.

A more succinct representation can be obtained by using a symbolic representation of the transition relation by introducing transitions whose input labels are *predicates*, and whose output labels are *terms* indexed by input symbols. Replacing input labels with predicates has been described first described by Watson [60]. Such symbolic transducers have been developed further and have recently received quite a bit of attention, with applications in verification and verifiable string transformations [54, 56–58].

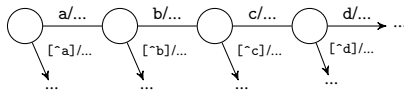
Our implementation of Kleenex uses a symbolic representation for basic ranges of symbols in order to get rid of most redundancies. The simulation algorithm and the SST construction can be generalized to the symbolic case without altering the fundamental structure, so we have elided the details of this optimization. We refer the reader to the cited literature for the technical details of symbolic transducers.

5.5 Finite lookahead

A common pattern in Kleenex programs are definitions of the form

```
token := ~/abcd/ commonCase | ~/[a-z]+/ fallback
```

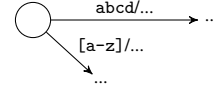
that is, a specific pattern appearing with higher priority than a more general fallback pattern. Patterns of this form will result in (symbolic) SSTs containing the following kind of structure:



The primary case and the fallback pattern are simulated in lockstep, and in each state there is a transition for when the common case fails after reading 0, 1, 2, etc. symbols.

If the SST was able to look more than one symbol ahead before determining the next state, we would be able to tabulate a much coarser set of simulation states and do away with the fine-grained

interleaving. For the above example, we would like a transition structure like the following:



If the first four symbols of the input are abcd, the upper transition is taken. If this is not the case, but the first symbol is a, then the lower transition is taken. The idea is that any string successfully matched by the primary case will satisfy the test abcd, so if the transition with [a-z] is taken, then the FST states corresponding to the primary case can be removed from the generalized state set and tabulation can continue with a simpler simulation state.

The semantics of SSTs with lookahead are still deterministic despite the seeming overlap of patterns, as the model requires that any pair of tests are either disjoint (no string will satisfy both at the same time), or one test is completely contained in another (if a string satisfies the first test, it also satisfies the second). This restriction gives a total order between tests, specifying their priority—the most specific test must be tried first.

6. Benchmarks

We have run comparisons with different combinations of the following tools:

RE2, Google’s automata-based regular expression C++ library [50].
RE2J, a recent re-implementation of RE2 in Java [51].

GNU AWK, GNU grep, and GNU sed, programming languages and tools for text processing and extraction [49].

Oniglib, a regular expression library written in C++ with support for different character encodings [33].

Ragel, a finite state machine compiler with multiple language backends [53].

In addition, we implemented test programs using the standard regular expression libraries in the scripting languages Perl [59], Python [34], and Tcl [61].

Meaning of plot labels Kleenex plot labels indicate the compilation path, and follow the format `[<0|3>[~1a] | woACT] [clang|gcc]`. 0/3 indicates whether constant propagation was disabled/enabled. 1a indicates whether lookahead was enabled. clang/gcc indicates which C compiler was used. The last part indicates that custom register updates are disabled, in which case we generate a single fused SST as described in 6.3. These are only run with constant propagation and lookahead enabled.

Experimental setup The benchmark machine runs Linux, has 32 GB RAM and an eight-core Intel Xeon E3-1276 3.6 GHz CPU with 256 KB L2 cache and 8 MB L3 cache. Each benchmark program was run 15 times, after first doing two warm-up rounds. Version numbers of libraries, etc. are included in the appendix. All C and C++ files have been compiled with -O3.

Difference between Kleenex and the other implementations Unless otherwise stated, the structure of all the non-Kleenex implementations is a loop that reads input line by line and applies an action to the line. Hence, in these implementations there is an interplay between the regular expression library used and the external language, e.g., RE2 and C++. In Kleenex, line breaks do not carry any special significance, so the multi-line programs can be formulated entirely within Kleenex.

Ragel optimization levels Ragel is compiled with three different optimization levels: T1, F1, and G2. “T1” and “F1” means that the generated C code should be based on a lookup-table, and “G2” means that it should be based on C goto statements.

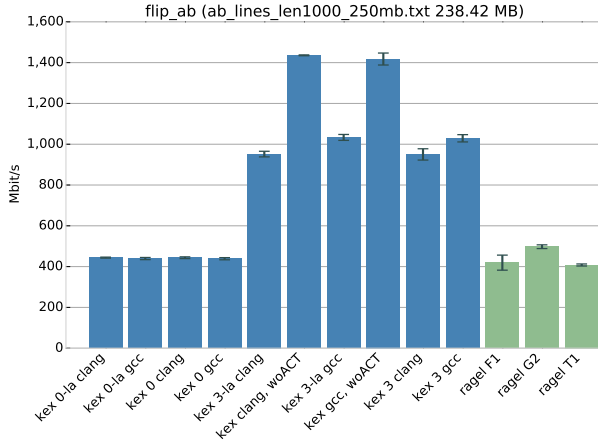


Figure 4: `flip_ab` run on lines with average length 1000.

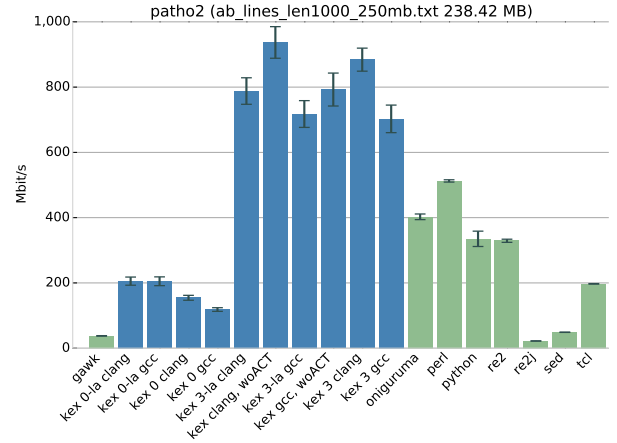


Figure 5: `patho2` run on lines with average length 1000.

Kleenex compilation timeout On some plots, some versions of the Kleenex programs are not included. This is because the C compiler has timed out (after 30 seconds). As we fully determinize the transducers, the resulting C code can explode in some cases. This is an area for future improvement.

6.1 Baseline

The following three programs are intended to give a baseline impression of the performance of Kleenex programs.

flip_ab The program `flip_ab` swaps “a”s and “b”s on all its input lines. In Kleenex it looks like this:

```
main := ("b" ~/a/ | "a" ~/b/ | /\n/)*
```

We made a corresponding implementation with Ragel, using a `while`-loop in C to get each new input line and feed it to the automaton code generated by Ragel.

Implementing this functionality with regular expression libraries in the other tools would be an unnatural use of them, so we have not measured those.

The performance of the two implementations run on input with an average line length of 1000 characters is shown in Figs. 4.

patho2 The program `patho2` forces Kleenex to wait until the very last character of each line has been read before it can produce any output:

```
main := ((~/[a-z]*a/ | /[a-z]*b/)? /\n/)+
```

In this benchmark, the constant propagation makes a big difference, as Fig. 5 shows. Due to the high degree of interleaving and the lack of keywords, in this program the look-ahead optimization to reduces overall performance.

This benchmark was not run with Ragel because Ragel requires the programmer to do all disambiguation manually when writing the program; the C code that Ragel generates does not handle ambiguity in a predictable way.

6.2 Rewriting

Thousand separators The following Kleenex program inserts thousand separators in a sequence of numbers:

```
main := (num /\n/)*
num := digit{1,3} ("," digit{3})*
digit := /[0-9]/
```

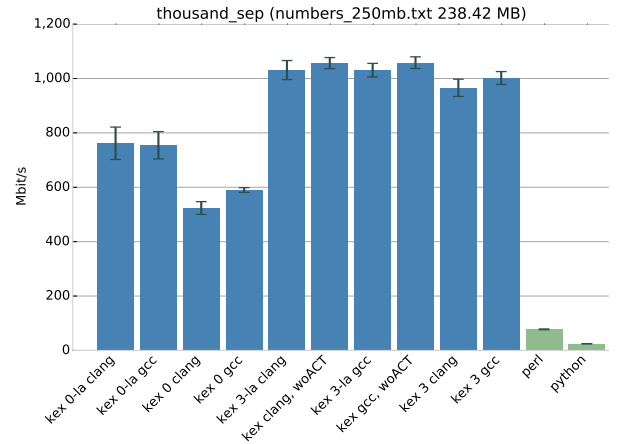


Figure 6: Inserting thousand separators on random numbers with average length 1000.

We evaluated the Kleenex implementation along with three other implementations using Perl, and Python. The performance can be seen in Fig. 6. Both Perl and Python are significantly slower than all of the Kleenex implementations; this is a problem that is tricky to formulate with normal regular expressions (unless one reads the input right-to-left).

CSV rewriting The program `csv_project3` deletes columns two and five from a CSV file:

```
main := (row /\n/)*
col := /[^\n]*/
row := ~(col /,/ ) col "\t" ~/,/ ~(col /,/ )
      ~(col /,/ ) col ~/,/ ~col
```

Various specialized tools exist that can handle this transformation are included in Fig. 7; GNU `cut` is a command that splits its input on certain characters, and GNU `AWK` has built-in support for this type of transformation.

Apart from `cut`, which is really fast for its own use-case, the Kleenex implementation is the fastest. The performance of Ragel is slightly lower, but this is likely due to the way the implementation produces output: In a Kleenex program, output strings are automat-

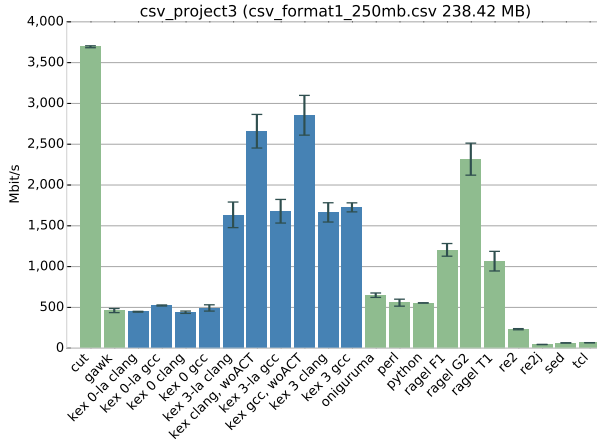


Figure 7: csv_project3 reads in a CSV file with six columns and outputs columns two and five. “gawk” is GNU AWK that uses the native AWK way of splitting up lines. “cut” is a command from GNU coreutils that splits up lines.

ically put in an output buffer which is flushed routinely, whereas a programmer has to manually handle buffering when writing a Ragel program.

IRC protocol handling The following Kleenex program parses the IRC protocol as specified in RFC 2812.⁴ It follows roughly the output style described in part 2.3.1. Note that the Kleenex source code and the BNF grammar in the RFC are almost identical. Fig. 8 shows the throughput on 250 mb data.

```
main := (message | "Malformed line: " /[^\r\n]*\r?\n)*
message := (~/:/ "Prefix: " prefix "\n" ~/ /)?
           "Command: " command "\n"
           "Parameters: " params? "\n"
           ~crlf
command := letter+ | digit{3}
prefix := servername
         | nickname ((/! user)? /@ host )?
user := /[^\n\r @!]+ // Missing \x00
middle := nospcrlfcl ( /:/ | nospcrlfcl )*
params := ( ~/ / middle ", " ){14} ( ~/ / trailing )?
         | ( ~/ / middle ){14} ( ~/ / trailing )?
trailing := ( /:/ | /:/ | nospcrlfcl )*
nickname := (letter | special)
            (letter | special | digit){10}
host := hostname | hostaddr
servername := hostname
hostname := shortname ( /\./ shortname)*
hostaddr := ip4addr
shortname := (letter | digit) (letter | digit | /-)*
            (letter | digit)*
ip4addr := (digit{1,3} /\./ ){3} digit{1,3}
```

6.3 With or without action-separation

One can choose to use the machine resulting in combining the oracle and the action machine when compiling Kleenex. Doing so results in only one process doing both the disambiguation and outputting, which in some cases is faster and in other slower. Figs. 7, 9, and 11 illustrate both situations. It depends on the structure of the problem whether it pays off to split up the work in two processes; if all the work happens in the oracle and the action

⁴<https://tools.ietf.org/html/rfc2812>

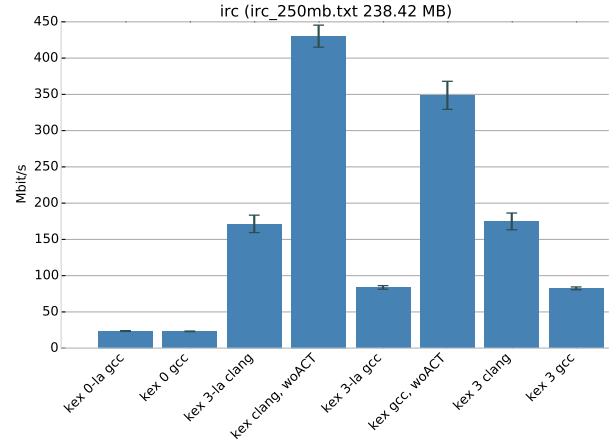


Figure 8: Throughput when parsing 250 MiB random IRC data.

machine nearly does not do anything, then the added overhead incurred by the process context switches becomes noticeable. On the other hand, in cases where both machines do much work, the fact that two CPU cores can be utilized speeds up the program. This would be more likely if Kleenex had support for actions which could perform arbitrary computation, e.g. in the form of embedded C code.

7. Use cases

In this section we will briefly touch upon various interesting use cases for Kleenex.

JSON logs to SQL We have implemented a Kleenex program (code in Appendix) that transforms a JSON log file into an SQL insert statement. The program works on the logs provided by Issuu.⁵

The Ragel version we implemented outperforms Kleenex by about 50% (Fig. 9), indicating that further optimizations of our SST construction should be possible.

Apache CLF to JSON The Kleenex program below rewrites Apache CLF⁶ log files into a list of JSON records:

```
main := "[" loglines? "]" \n
loglines := (logline ", " /\n)* logline /\n/
logline := "{" host ~sep ~userid ~sep ~authuser sep
             timestamp sep request sep code sep
             bytes sep referer sep useragent "}"
host := "\"host\": \" ip \""
userid := "\"user\": \" rfc1413 \""
authuser := "\"authuser\": \" /[^\n]+/ \""
timestamp := "\"date\": \" ~/[^\n]+/ ~/]/ \""
request := "\"request\": \" quotedString
code := "\"status\": \" integer \"
bytes := "\"size\": \" (integer | /-/) \"
referer := "\"url\": \" quotedString
useragent := "\"agent\": \" quotedString
ws := /[ \t ]+/
sep := ", " ~ws
quotedString := /"([^\n]|\\")*/
```

⁵The line-based data set consists of 30 compressed parts and part one is available from <http://labs.issuu.com/anodataset/2014-03-1.json.xz>

⁶<https://httpd.apache.org/docs/trunk/logs.html#common>

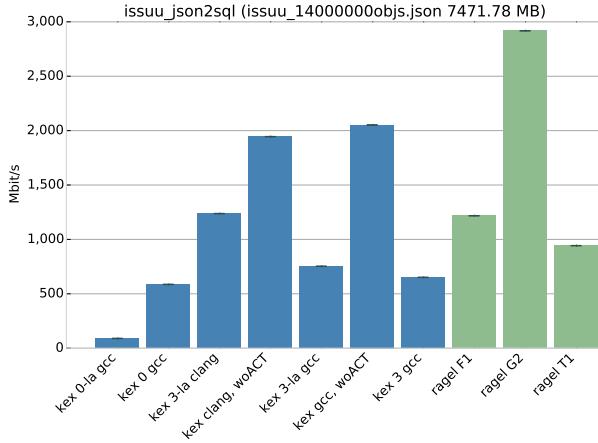


Figure 9: The speeds of transforming JSON objects to SQL INSERT statements using Ragel and Kleenex.

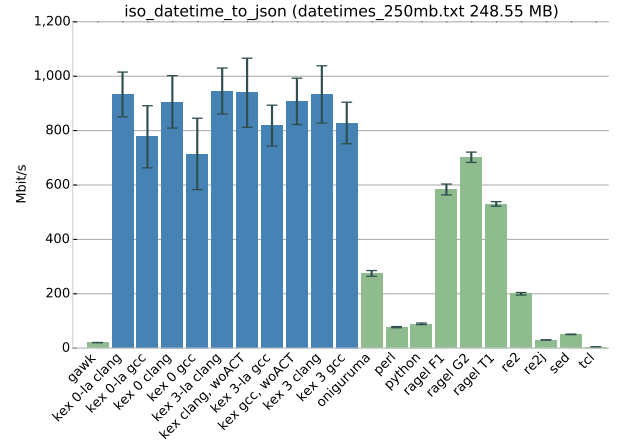


Figure 11: The performance of the conversion of ISO time stamps into JSON format.

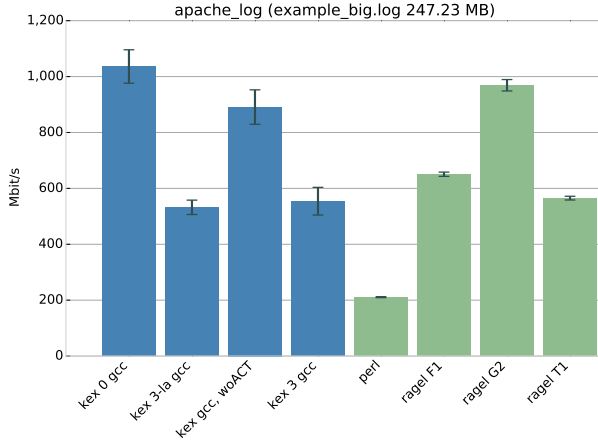


Figure 10: Speed of the conversion from the Apache Common Log Format to JSON.

```
integer := /[0-9]+/
ip := integer (/\./ integer){3}
rfc1413 := /-/
```

This is a re-implementation of a Ragel program.⁷ Fig. 10 shows the benchmark results. The versions compiled with clang were not included, as the compilation timed out after 30 seconds. Curiously, the non-optimized Kleenex program is the fastest in this case.

ISO date/time objects to JSON Inspired by an example in [26], the program `iso_datetime_to_json` (code in Appendix) converts date and time stamps in an ISO standard format to a JSON object. Fig. 11 shows the performance.

URL parsing Kleenex allows one to naturally follow the URL specification given in RFC1738.⁸ We implemented a URL parser by directly following the BNF-grammar in the RFC; its code can be found in the Appendix.

⁷<https://engineering.emcien.com/2013/04/5-building-tokenizers-with-ragel>

⁸<http://www.ietf.org/rfc/rfc1738.txt>

Syntax highlighting Kleenex can be used to write syntax highlighters in; in fact, the Kleenex syntax in this paper was highlighted with a Kleenex program. The code for a version that emits ANSI color codes is included in the Appendix.

HTML comments The following Kleenex program finds HTML comments with basic formatting commands and renders them in HTML after the comment. For example, `<!-- doc: *Hello* world -->` becomes `<!-- doc: *Hello* world --><div><b>Hello</b> world </div>`.

```
main := (comment | /.)*
comment := /<!-- doc:/ clear doc* !orig /-->/
          "<div>" !render "</div>"
doc := ~/\*/ t@/[~*]*/ ~/\*/
      [ orig += "*" t "*" ] [ render += "<b>" t "</b>" ]
      | t@/./ [ orig += t ] [ render += t ]
clear := [ orig <- "" ] [ render <- "" ]
```

8. Related Work

We discuss related work in the context of current and future work.

8.1 Regular expression matching

Regular expression *matching* has different meanings in the literature.

For *acceptance testing*, which corresponds to classical automata theory, Bille and Thorup [11] improve on Myers' [37] log-factor improved RE-membership testing of classical NFA-simulation, based on tabling. They design an $O(kn)$ algorithm [12] with word-level parallelism, where $k \leq m$ is number of strings occurring in an RE. The tabling technique may be promising in practice; the algorithms have not been implemented and evaluated empirically, though.

In *subgroup matching* as in PCRE [29], an input is not only classified as accepting or not, but a substring is returned for each sub-RE in an RE designated to be of interest. Subgroup matching is often implemented by backtracking over alternatives, which yields the greedy match.⁹ It may result in exponential-time behavior, however. Consequently, considerable human effort is expended to engineer REs to perform well. REs resulting in exponential run-time behavior are used in algorithmic attacks, leading

⁹Committing to the left alternative before checking that the remainder of the input is accepted is the essence of parsing expression grammars [22].

to proposals for countermeasures to such attacks by classifying REs with slow backtracking performance [42, 46], where the countermeasures in turn appear to be attackable. Even in the absence of inherently hard matching with backreferences [1], backtracking implementations with avoidable performance blow-ups are amazingly wide-spread. This may be due to a combination of their good best-case performance and PCRE-embellishments driven by use cases. Some submatch libraries with guaranteed worst-case linear-time performance, notably RE2 [50], are making inroads, however. Myers, Oliva and Guimaraes [36] and Okui, Suzuki [41] describe a $O(mn)$, respectively $O(m^2n)$ POSIX-disambiguated matching algorithms. Sulzmann and Lu [47] use Brzozowski [17] and Antimirov derivatives [7] for Perl-style subgroup matching for greedy and POSIX disambiguation.

Full RE *parsing* generalizes submatching: it returns a list of matches for each Kleene-star, also for nested ones. Kearns [32], Frisch and Cardelli [23] devise 3-pass linear-time *greedy* RE parsing; they require 2 passes over the input, the first consisting of reversing the entire input, before generating output in the third pass. Grathwohl, Henglein, Nielsen, Rasmussen devise a two-pass [27] and an optimally streaming [28] greedy regular expression parsing algorithm. Streaming guarantees that line-by-line RE matching can be coded as a single RE matching problem. Sulzmann and Lu [48] remark that POSIX is notoriously difficult to implement correctly and show how to use Brzozowski derivatives [17] for POSIX RE parsing;

There are specialized RE matching tools and techniques too numerous to review comprehensively. We mention a few employing automaton optimization techniques applicable to Kleenex, but presently unexplored. Yang, Manadhata, Horne, Rao, Ganapathy [62] propose an OBDD representation for subgroup matching and apply it to intrusion detection REs; the cycle counts per byte appear a bit high, but are reported to be competitive with RE2. Sidhu and Prasanna [45] implement NFAs directly on an FPGA, essentially performing NFA-simulation in parallel; it outperforms GNU `grep`. Brodie, Taylor, Cytron [14] construct a multistride DFA, which processes multiple input symbols in parallel, and devise a compressed implementation on stock FPGA, also achieving very high throughput rates. Likewise, Ziria employs tabled multistriding to achieve high throughput [25]. Navarro and Raffinot [39], show how to code DFAs compactly for efficient simulation.

8.2 Ambiguity

REs may be ambiguous, which is irrelevant for acceptance testing, but problematic for submatching and parsing since the output depends on which amongst possibly multiple matches is to be returned. Brüggemann-Klein [15] provides an efficient $O(m^2)$ RE ambiguity testing algorithm. Vansummeren [55] illustrates differences between POSIX, first/longest and greedy matches. Colcombet [18] analyzes notions of (non)determinism of automata.

8.3 Transducers

From RE parsing it is a surprisingly short distance to the implementation of arbitrary nondeterministic finite state transducers (NFSTs) [10, 35]. In contrast to the situation for *automata*, nondeterministic transducers are strictly more powerful than deterministic transducers; this, together with observable ambiguity, highlights why RE parsing is more challenging than RE acceptance testing.

As we have seen, efficient RE parsing algorithms operate on arbitrary NFAs, not only those corresponding to REs. Indeed, REs are not a particularly convenient or compact way of specifying regular languages: they can be represented by *certain* small NFAs with low tree-width, but may be inherently quadratically bigger even for DFAs [20, Theorem 23]. This is why Kleenex employs

context-free grammars restricted to denote regular languages, with embedded output actions, to denote NFSTs.

We have shown that NFSTs, in particular unambiguous NFSTs, can be implemented by a *subclass* of streaming string transducers (SSTs). SSTs extensionally correspond to regular transductions, functions implementable by 2-way deterministic finite-state transducers [3], MSO-definable string transductions [21] and a combinator language analogous to regular expressions [5]. The implementation techniques used in Kleenex appear to be directly applicable to all SSTs, not just the ones corresponding to NFSTs.

Allender and Mertz [2] show that the functions computable by register automata, which generalize output strings to arbitrary monoids, are in NC and thus inherently parallelizable. This is achievable by performing relational NFST-composition by matrix multiplication on the matrix representation of NFSTs [10], which can be performed by parallel reduction. This is tantamount to running an NFST from all states, not just the input state, on input string fragments. Mytkowicz, Musuvathi, Schulte [38] observe that there is often a small set of cut states sufficient to run each NFST. This promises to be an interesting parallel harness for a suitably adapted Kleenex implementation running on fragments of very large inputs.

Veanes, Molnar, Mytkowicz [58] employ symbolic transducers [13, 19, 57] and a data-parallel intermediate language in the implementation of BEK for multicore execution.

9. Conclusions

We have presented Kleenex, a convenient language for specifying nondeterministic finite state transducers; and its compilation to machine code representations of streaming state transducers, which emit the output

Kleenex is comparatively expressive and performs consistently well—for complex regular expressions with nontrivial amounts of output almost always better in the evaluated use cases—vis-à-vis text processing tools such as RE2, Ragel, `grep`, `AWK`, `sed` and RE-libraries of Perl, Python and Tcl.

We believe Kleenex’s clean semantics, streaming optimality, algorithmic generality, worst-case guarantees and absence of tricky code and special casing provide a useful basis for

- extensions to deterministic visible push-down automata, restricted versions of backreferences and approximate/probabilistic matching;
- known, but so far unexplored optimizations, such as multi-character input processing, automata minimization and symbolic representation, hybrid NFST-simulation/SST-construction (analogous to NFA-simulation with NFA-state set memoization to implement on-demand DFA-construction);
- massively parallel (log-depth, linear work) input processing.

Acknowledgments

We thank Issuu for releasing their data set to the research community.

References

- [1] A. V. Aho. Algorithms for finding patterns in strings. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume Algorithms and Complexity (A), pages 255–300. Elsevier and MIT Press, 1990. ISBN 0-444-88071-2 and 0-262-22038-5.
- [2] E. Allender and I. Mertz. Complexity of regular functions. In *Proc. LATA*, 2015.
- [3] R. Alur and P. Černý. Expressiveness of streaming string transducers. In *Proc. Foundations of Software Technology and Theoretical Computer Science (FSTTCS)*, 2010. URL <http://dx.doi.org/10.4230/LIPIcs.FSTTCS.2010.1>.

- [4] R. Alur and P. Černý. Streaming transducers for algorithmic verification of single-pass list-processing programs. *ACM SIGPLAN Notices*, 46(1):599–610, 2011.
- [5] R. Alur, A. Freilich, and M. Raghothaman. Regular combinators for string transformations. In *Proceedings of the Joint Meeting of the Twenty-Third EACSL Annual Conference on Computer Science Logic (CSL) and the Twenty-Ninth Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*, CSL-LICS '14, pages 9:1–9:10, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2886-9. . URL <http://doi.acm.org/10.1145/2603088.2603151>.
- [6] R. Alur, L. D'Antoni, and M. Raghothaman. DRex: A declarative language for efficiently evaluating regular string transformations. In *Proc. 42nd ACM Symposium on Principles of Programming Languages (POPL)*, 2015.
- [7] V. Antimirov. Partial derivatives of regular expressions and finite automaton constructions. *Theor. Comput. Sci.*, 155(2):291–319, 1996. ISSN 0304-3975. .
- [8] A. W. Appel. *Modern Compiler Implementation in ML*. Cambridge University Press, 1998. ISBN 0521582741. URL <http://dl.acm.org/citation.cfm?id=522388>.
- [9] M.-P. Béal and O. Carton. Determinization of transducers over finite and infinite words. *Theoretical Computer Science*, 289(1):225–251, Oct. 2002. ISSN 03043975. .
- [10] J. Berstel. *Transductions and Context-Free Languages*. Teubner Stuttgart, 1979.
- [11] P. Bille and M. Thorup. Faster regular expression matching. In *Proc. 36th International Colloquium on Automata, Languages and Programming (ICALP)*, pages 171–182, July 2009.
- [12] P. Bille and M. Thorup. Regular expression matching with multi-strings and intervals. In *Proc. 21st ACM-SIAM Symposium on Discrete Algorithms (SODA)*, 2010.
- [13] N. Bjørner and M. Veanes. Symbolic transducers. Technical Report MSR-TR-2011-3, Microsoft Research, 2011.
- [14] B. Brodie, D. Taylor, and R. Cytron. A scalable architecture for high-throughput regular-expression pattern matching. *ACM SIGARCH Computer Architecture News*, 34(2):202, 2006. . URL <http://dx.doi.org/10.1145/1150019.1136500>.
- [15] A. Brüggemann-Klein. Regular expressions into finite automata. *Theor. Comput. Sci.*, 120(2):197–213, 1993. ISSN 0304-3975. .
- [16] A. Brüggemann-Klein and D. Wood. One-unambiguous regular languages. *Information and computation*, 140(2):229–253, 1998.
- [17] J. A. Brzozowski. Derivatives of regular expressions. *J. ACM*, 11(4):481–494, 1964. ISSN 0004-5411. .
- [18] T. Colcombet. Forms of determinism for automata. In *Proc. 29th Symposium on Theoretical Aspects of Computer Science (STACS)*, volume 14, pages 1–23. LIPIcs, 2012.
- [19] L. D'Antoni and M. Veanes. Minimization of symbolic automata. In *Proceedings of the 41th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, San Diego, California, January 2014. ACM Press. .
- [20] K. Ellul, B. Krawetz, J. Shallit, and M.-w. Wang. Regular expressions: New results and open problems. *Journal of Automata, Languages and Combinatorics*, 10(4):407–437, 2005.
- [21] J. Engelfriet and H. Hoogeboom. MSO definable string transductions and two-way finite-state transducers. *ACM Transactions on Computational Logic (TOCL)*, 2(2):216–254, 2001. ISSN 1529-3785.
- [22] B. Ford. Parsing expression grammars: a recognition-based syntactic foundation. In *ACM SIGPLAN Notices*, number 1 in 39, pages 111–122. ACM, 2004.
- [23] A. Frisch and L. Cardelli. Greedy regular expression matching. In *Proc. 31st International Colloquium on Automata, Languages and Programming (ICALP)*, volume 3142 of *Lecture notes in computer science*, pages 618–629, Turku, Finland, July 2004. Springer.
- [24] V. Glushkov. The abstract theory of automata. *Russian Mathematical Surveys*, 16(5):1–53, 1961. . URL <http://dx.doi.org/10.1070/RM1961v016n05ABEH004112>.
- [25] M. Gowda, G. Stewart, G. Mainland, B. Radunović, D. Vytiniotis, and D. Patterson. Ziria: Language for rapid prototyping of wireless phy. In *Proceedings of the 20th annual international conference on Mobile computing and networking*, pages 359–362. ACM, 2014.
- [26] J. Goyvaerts and S. Levithan. *Regular Expressions Cookbook*. O'Reilly, 2009. ISBN 978-0-596-52068-7.
- [27] N. B. B. Grathwohl, F. Henglein, L. Nielsen, and U. T. Rasmussen. Two-pass greedy regular expression parsing. In *Proc. 18th International Conference on Implementation and Application of Automata (CIAA)*, volume 7982 of *Lecture Notes in Computer Science (LNCS)*, pages 60–71. Springer, July 2013. .
- [28] N. B. B. Grathwohl, F. Henglein, and U. T. Rasmussen. Optimally Streaming Greedy Regular Expression Parsing. In *Theoretical Aspects of Computing - ICTAC 2014 - 11th International Colloquium, Bucharest, Romania, September 17-19, 2014. Proceedings*, pages 224–240, 2014. .
- [29] P. Hazel. Pcre – perl-compatible regular expressions. Concatenation of PCRE man pages, January 3 2010.
- [30] F. Henglein and L. Nielsen. Regular expression containment: Coinductive axiomatization and computational interpretation. *SIGPLAN Notices, Proc. 38th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL)*, 46(1):385–398, January 2011. .
- [31] L. Ilie and S. Yu. Follow automata. *Information and computation*, 186(1):140–162, 2003.
- [32] S. Kearns. Extending regular expressions with context operators and parse extraction. *Software - Practice and Experience*, 21(8):787–804, 1991. .
- [33] K. Kosako. The Oniguruma regular expression library, 2014. URL <http://www.geocities.jp/kosako3/oniguruma/>.
- [34] M. Lutz. *Programming Python*, volume 8. O'Reilly, 4th edition, December 2010. ISBN 978-0-596-15810-1.
- [35] M. Mohri. Finite-state transducers in language and speech processing. *Computational linguistics*, 23(2):269–311, 1997.
- [36] E. Myers, P. Oliva, and K. Guimarães. Reporting exact and approximate regular expression matches. In *Combinatorial Pattern Matching*, pages 91–103. Springer, 1998.
- [37] G. Myers. A four Russians algorithm for regular expression pattern matching. *J. ACM*, 39(2):432–448, 1992. ISSN 0004-5411. .
- [38] T. Mytkowicz, M. Musuvathi, and W. Schulte. Data-parallel finite-state machines. In *Proceedings of the 19th international conference on Architectural support for programming languages and operating systems*, pages 529–542. ACM, 2014.
- [39] G. Navarro and M. Raffinot. Compact dfa representation for fast regular expression search. *Algorithm Engineering*, pages 1–13, 2001.
- [40] L. Nielsen and F. Henglein. Bit-coded regular expression parsing. In *Proc. 5th Int'l Conf. on Language and Automata Theory and Applications (LATA)*, Lecture Notes in Computer Science (LNCS), pages 402–413. Springer, May 2011.
- [41] S. Okui and T. Suzuki. Disambiguation in regular expression matching via position automata with augmented transitions. Technical Report 2013-002, The University of Aizu, June 2013.
- [42] A. Rathnayake and H. Thielecke. Static analysis for regular expression exponential runtime via substructural logics. *CoRR*, abs/1405.7058, 2014. URL <http://arxiv.org/abs/1405.7058>.
- [43] G. Schnitger. Regular expressions and nfes without ϵ -transitions. In *STACS 2006*, pages 432–443. Springer, 2006.
- [44] M. Schützenberger. Sur une variante des fonctions séquentielles. *Theoretical Computer Science*, 4(1):47–57, Feb. 1977. .
- [45] R. Sidhu and V. Prasanna. Fast Regular Expression Matching Using FPGAs. In *Proc. 9th Annual IEEE Symposium on Field-Programmable Custom Computing Machines, 2001. FCCM '01*, pages 227–238, 2001.
- [46] S. Sugiyama and Y. Minamide. Checking time linearity of regular expression matching based on backtracking. In *IPSP Transactions on Programming*, number 3 in 7, pages 1–11, 2014.

- [47] M. Sulzmann and K. Z. M. Lu. Regular Expression Sub-matching Using Partial Derivatives. In *Proceedings of the 14th symposium on Principles and practice of declarative programming*, PPDP '12, pages 79–90, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1522-7. . URL <http://doi.acm.org/10.1145/2370776.2370788>.
- [48] M. Sulzmann and K. Z. M. Lu. Posix regular expression parsing with derivatives. In *Proc. 12th International Symposium on Functional and Logic Programming*, FLOPS '14, Kanazawa, Japan, June 2014.
- [49] The GNU Project, 2015. URL <http://www.gnu.org/software/coreutils/coreutils.html>.
- [50] The RE2 authors. RE2, 2015. URL <https://github.com/google/re2>.
- [51] The RE2J authors. RE2J, 2015. URL <https://github.com/google/re2j>.
- [52] K. Thompson. Programming techniques: Regular expression search algorithm. *Commun. ACM*, 11(6):419–422, 1968. ISSN 0001-0782. . URL <http://dx.doi.org/10.1145/363347.363387>.
- [53] A. Thurston. Ragel state machine compiler, 2015. URL <http://www.colm.net/open-source/ragel/>.
- [54] G. van Noord and D. Gerdemann. Finite State Transducers with Predicates and Identities. *Grammars*, 4(3):263–286, 2001. ISSN 1386-7393. .
- [55] S. Vansummeren. Type inference for unique pattern matching. *ACM Trans. Program. Lang. Syst.*, 28(3):389–428, 2006. ISSN 0164-0925. .
- [56] M. Veanes. Symbolic String Transformations with Regular Lookahead and Rollback. In *Ershov Informatics Conference (PSI'14)*. Springer Verlag, 2014. URL <http://research.microsoft.com/apps/pubs/default.aspx?id=213110>.
- [57] M. Veanes, P. Hooimeijer, B. Livshits, D. Molnar, and N. Bjorner. Symbolic finite state transducers: Algorithms and applications. In *Proceedings of the 39th Annual Symposium on Principles of Programming Languages*, POPL '12, pages 137–150, New York, NY, USA, 2012. .
- [58] M. Veanes, D. Molnar, T. Mytkowicz, and B. Livshits. Data-parallel string-manipulating programs. In *Proceedings of the 42nd annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. ACM Press, 2015.
- [59] L. Wall, T. Christiansen, and J. Orwant. *Programming Perl*. O'Reilly, 3rd edition, July 2000.
- [60] B. W. Watson. Implementing and using finite automata toolkits. *Natural Language Engineering*, 2(04):295–302, 1996. ISSN 1469-8110. .
- [61] B. B. Welch, K. Jones, and J. Hobbs. *Practical programming in Tcl and Tk*. Prentice Hall, 4th edition edition, 2003. ISBN 0130385603.
- [62] L. Yang, P. Manadhata, W. Horne, P. Rao, and V. Ganapathy. Fast submatch extraction using obdds. In *Proceedings of the Eighth ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, ANCS '12, pages 163–174, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1685-9. . URL <http://doi.acm.org/10.1145/2396556.2396594>.

Kleenex: Compiling Nondeterministic Transducers to Deterministic Streaming Transducers

Supplementary material

1. Benchmarks and use cases

Version numbers of tools, libraries, etc.:

Name	Version
gcc	4.8.4
clang	3.5.0
Perl	5.20.1
Python	2.7.9
Tcl	8.6.3
GNU AWK	4.0.2
GNU grep	2.21
GNU sed	4.2.1
GNU coreutils	8.21
Oniguruma	5.9.6
RE2	GitHub: bdb5058
RE2J	1.0
Ragel	6.9
DRex	20150114

2. Example programs

rot13 The *rot13* program shifts letters in the English alphabet by 13 places. In Kleenex it can be implemented as follows:

```
main := (r13 | /.)*
r13 := ~/a/ "n" | ~/b/ "o" | ~/c/ "p" | ~/d/ "q"
      | ~/e/ "r" | ~/f/ "s" | ~/g/ "t" | ~/h/ "u"
      | ~/i/ "v" | ~/j/ "w" | ~/k/ "x" | ~/l/ "y"
      | ~/m/ "z" | ~/n/ "a" | ~/o/ "b" | ~/p/ "c"
      | ~/q/ "d" | ~/r/ "e" | ~/s/ "f" | ~/t/ "g"
      | ~/u/ "h" | ~/v/ "i" | ~/w/ "j" | ~/x/ "k"
      | ~/y/ "l" | ~/z/ "m"
```

INI file to JSON Consider the task of rewriting an INI file to an equivalent JSON dictionary that contains entries for each configuration group. This can be expressed in Kleenex, and Fig. 2 shows the performance numbers.

BibTeX rewriting The following are all re-implementations of some of the test programs used in DReX.

This program deletes line comments:

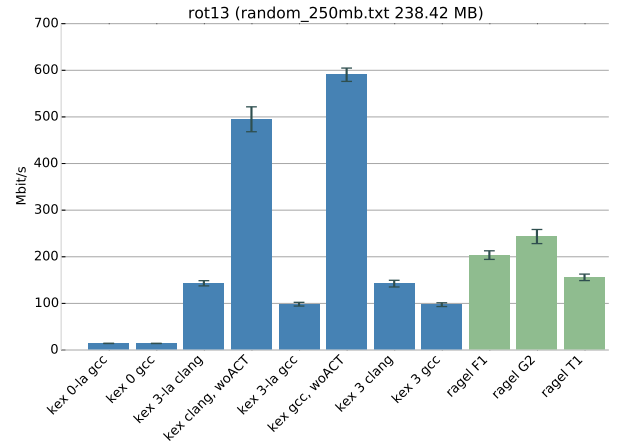


Figure 1: The speed of the *rot13* program. The choice of C compiler and the constant propagation make a big difference.

```
main := (~/\[/ ~line | line)*
line := /[^\n]*\n/
```

This program strips out all tags from an XML file:

```
//Concatenate all XML tags, ignore things in between.
main := (tag | ~/.)*
tag := /<[^>]*>/
```

This program shifts the title field in a BibTeX file to the entry one position up:

```
// Moves all title entries up to the previous entry
// in a bibtex file. The last entry is deleted.

// Uncomment to also swap title to the top
//start: align >> swap
start: align
align := head@header field* foot@footer
      ( !head head@header put_rest field*
        !title !foot foot@footer)*
swap := (header field* !title put_rest footer)*
field := title@(sp /title/ sp /=/ sp /\{[^\}]*\},?\n/)
      | f@(sp word sp /=/ sp /\{[^\}]*\},?\n/) [ rest += f ]
header := /\@/ word sp /\{/ sp alnum /\n/
footer := /\}/ (sp /\n/)*
put_rest := !rest [ rest <- "" ]
word := /[A-Za-z_]+/
alnum := /[A-Za-z0-9_]+/
sp := /\s*/
```

[Copyright notice will appear here once 'preprint' option is removed.]

```

/* Takes a FASTA file and counts how often each of the following patterns
occur in the DNA sequences.

1. agggtaaa|tttaccct
2. [cgt]gggtaaa|tttacc[acg]
3. a[act]ggtaaa|tttacc[agt]t
4. ag[act]gtaaa|tttac[agt]ct
5. agg[act]taaa|ttta[agt]cct
6. aggg[acg]aaa|ttt[cgt]ccct
7. aggg[cgt]aa|tt[acg]accct
8. agggta[cgt]a|t[acg]taccct
9. agggtaa[cgt]|[acg]ttaccct

Output is given in unary representation, counting how many matches each of these patterns has.
*/
start: init >> core1 >> core2 >> core3 >> cleanup

init := filter addCounts

// Filter out sequence identifiers + newlines
filter := (sequenceIdent | ~/\n/ | /./*

// for now we ignore the sequence separation
sequenceIdent := ~/>[^\n]*/

// Add counters for all 9 types at the end of the file
addCounts := "\n"
           "[type 1: ]\n"
           "[type 2: ]\n"
           "[type 3: ]\n"
           "[type 4: ]\n"
           "[type 5: ]\n"
           "[type 6: ]\n"
           "[type 7: ]\n"
           "[type 8: ]\n"
           "[type 9: ]\n"

a := ~/[aA]/
c := ~/[cC]/
g := ~/[gG]/
t := ~/[tT]/

core1 := (seq1 | seq2 | seq3 | /./*
core2 := (seq4 | seq5 | seq6 | /./*
core3 := (seq7 | seq8 | seq9 | /./*

seq1 := (a g g g t a a a | t t t a c c c t) [a <- a "1"] | /type 1: / !a
seq2 := ((c|g|t) g g g t a a a | t t t a c c c (a|c|g)) [b <- b "1"] | /type 2: / !b
seq3 := (a (a|c|t) g g t a a a | t t t a c c c (a|g|t) t) [c <- c "1"] | /type 3: / !c
seq4 := (a g (a|c|t) g t a a a | t t t a c (a|g|t) c t) [d <- d "1"] | /type 4: / !d
seq5 := (a g g (a|c|t) t a a a | t t t a (a|g|t) c c t) [e <- e "1"] | /type 5: / !e
seq6 := (a g g g (a|c|g) a a a | t t t (c|g|t) c c c t) [f <- f "1"] | /type 6: / !f
seq7 := (a g g g t (c|g|t) a a | t t (a|c|g) a c c c t) [g <- g "1"] | /type 7: / !g
seq8 := (a g g g t a (c|g|t) a | t (a|c|g) t a c c c t) [h <- h "1"] | /type 8: / !h
seq9 := (a g g g t a a (c|g|t) | (a|c|g) t t a c c c t) [i <- i "1"] | /type 9: / !i

cleanup := ~/[^ ]*/ /.*/

```

Figure 10: The regex-dna benchmark from <http://benchmarksgame.alioth.debian.org/>

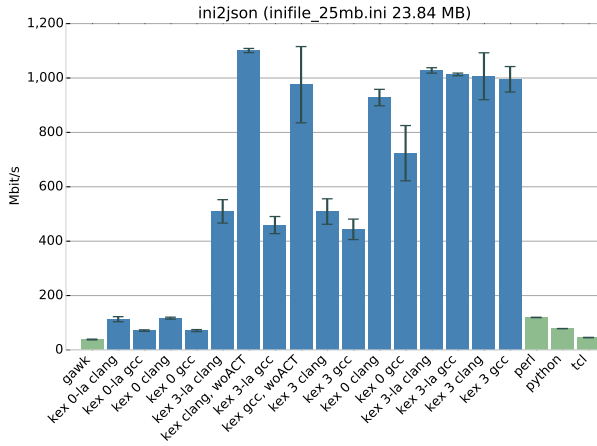


Figure 2: Speed of transforming roughly 25 megabytes of INI-file to a JSON-format.

```

start: stripini >> ini2json

// Strips the comments
stripini := (~comment | ~blank | /[^\n]*\n/)*

comment := ws /[^\n]*/
blank := ws /\n/

// Convert the stripped file
ini2json := "{\n" sections "}\n"

sections := (section "," /\n/)* section /\n/
section :=
    ind "\" header \"\": {\n" (~/\n/ keyvalues)? ind "\""

header := ~ws ~/[ / [^\n]]*/ ~/] / ~ws

keyvalue := ind ind key ": " ~/=/ value
keyvalues := (keyvalue "," /\n/)* keyvalue "\n"

key := ~ws "\" " /["; \t=\\[\n]]*/ "\" ~ws

value := ~ws /"[^\n]*"/ ~ws
        | ~ws "\" \" escapedValue "\" ~ws

escapedValue := (~/[ / [^\n]]* | ~/" / [^\n]]* | /[\n]/)*

ws := /[ \t]*/
ind := " "

```

(a) Kleenex program

```

print "{\n";

my $firstSection = 1;
my $firstKeyValuePair = 1;
my $ind = " ";

while (my $line = <>) {
    next if ($line =~ qr/^s*|~s*$/);

    if (my ($heading) =
        $line =~ qr/^s*\[[^\n]*\]\s*$/) {
        print "\n$ind},\n" unless $firstSection;
        $firstSection = 0;
        $firstKeyValuePair = 1;
        print "$ind\"$heading\": {";
        next;
    }

    if (my ($key, $value) = $line =~
        qr/^s*([^\s;\\]*)\s*=\s*([^\n]*?)\s*$/) {
        unless ($value =~ qr/^".*"$/) {
            $value =~ s/\\/\\\\/g;
            $value =~ s/"/\\"/g;
            $value = qq{"$value"};
        }

        print ", " unless ($firstKeyValuePair);
        $firstKeyValuePair = 0;
        print qq{\n$ind$ind"$key": $value};
    }
}

print "\n$ind}" unless $firstSection;
print "\n}\n";

```

(b) Perl program

Figure 3: The Kleenex source for the INI to JSON transformation. Below is the Perl implementation for comparison.

```

// Kleenex program to transform datetime corresponding to
// the xml schema "datetime" object. Outputs a JSON-like format
// More or less completely taken from
// "Regular Expressions Cookbook", p. 237.
start: dateTimes

dateTimes := (dateTime ~/\n/)+

dateTime := "{ 'year'=' year ~/-/ " "
            ", 'month'=' month ~/-/ " "
            ", 'day'=' day ~/T/ " "
            ", 'hours'=' hours ~:/ " "
            ", 'minutes'=' minutes ~:/ " "
            ", 'seconds'=' seconds " "
            ", 'tz'=' timezone " "
            "}" ~\n"

year := /(?:[1-9][0-9]*)?[0-9]{4}/
month := /1[0-2]|0[1-9]/
day := /3[0-1]|0[1-9]|1[1-2][0-9]/
hours := /2[0-3]|0[1][0-9]/
minutes := /[0-5][0-9]/
seconds := /[0-5][0-9]/
timezone := /Z|[+-](?:2[0-3]|0[1][0-9]):[0-5][0-9]/

```

Figure 4: The Kleenex source for iso_datetime_to_json.

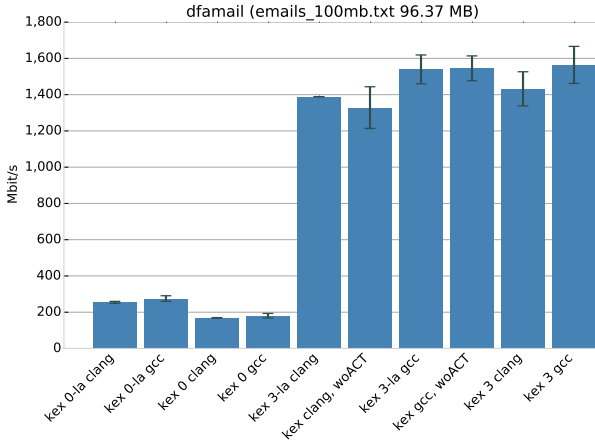


Figure 5: The Kleenex implementation of email with an added ~ on the entry-level name yields a degenerated SST without any output buffers which is a DFA.

```

main := ( escape | comment | term
          | symbol | ignored | ws* ) *
term  := black ~ ( constant | match | ident ) end
      | ( teal constant | yellow match | blue ident ) end
ignored := /[(){};,:[]/
ident   := ( letter | /[0-9_]/ ) +
symbol  := yellow /<=|:=|>|*|?|+|/ end
constant := /"/ ( /\./ | /[\\""]/ ) * /"/
comment  := black ( /\/*[\n]*\n/
                  | /\/*[^\n]*\n/ ) end
match    := /\// ( /[\n]/ | /\./ ) + /\//
escape   := /\//
          | blue /\x[0-9a-fA-F]{2}/ end
          | /\[tnr]/
sp       := / / *
letter   := /[a-zA-Z]/
word     := letter +
ws       := /[\t\r\n ]/
red      := "\x1b[31m"
green    := "\x1b[32m"
yellow   := "\x1b[33m"
blue     := "\x1b[34m"
end      := "\x1b[39;49m"
black    := "\x1b[30m"
teal     := "\x1b[36m"

```

Figure 6: A Kleenex program that highlights Kleenex syntax and emits ANSI color codes. A modified version of this was used to highlight the code in this paper.

```

main :=
  "INSERT INTO issuu_log (ts, visitor_uuid, "
  "visitor_useragent, visitor_country) VALUES\n"
  json2sql

json2sql := object "\n" ws json2sql
          | object ";\n" ws

object := "(" ~ /\{/ ws keyVals ws ~ /\}/ " )"
keyVals := (ws keyVal)+

keyVal :=
  ~/"ts"/                sep someInt                keepComma
  | ~/"visitor_uuid"/    sep stringReplaceQuotes keepComma
  | ~/"visitor_useragent"/ sep stringReplaceQuotes keepComma
  | ~/"visitor_country"/ sep stringReplaceQuotes dropComma
  | fb

fb := ~(/"/ someString /"/   sep ( /"/ someString /"/
                                | someInt
                                ) (dropComma | ""))

stringReplaceQuotes := qt someString qt
qt := " " ~/" // replace double with single quote
sep := ws ~:/ ws
someString := /[~"\n]*/
someInt := /-?[0-9]*/
someNumber := someInt /\./ someInt
contryCode := /[A-Z]{2}/

// Skip whitespace
ws := ~/[ \n]*/
keepComma := ws /,/
dropComma := ws ~/,/

```

Figure 7: The code for the Issuu JSON log file to SQL transformation.

```

[c]                                {
  compiler=gcc                     "c": {
  flags=-Wall -O3                  "compiler": "gcc",
                                   "flags": "-Wall -O3"
                                   },
  [haskell]                        "haskell": {
  compiler = ghc                   "compiler": "ghc",
  flags =                          "flags": ""
                                   }
}

```

Figure 8: An example of the INI to JSON transformation.

```

// Parses a list of RFC1738 generic URLs.
// https://www.ietf.org/rfc/rfc1738.txt
main := (genericurl /\n/)*
genericurl := scheme ~:/~ schemepart
scheme := "Scheme: " /[a-z0-9.-+~]/ " \n"
schemepart := ip_schemepart
              | "Scheme-part: " xchars " \n"
ip_schemepart := ~/\~ login (~/\~ urlpath)?
login := (user (~:/~ password)? ~/@/? hostport
hostport := host (~:/~ port)?
host := "Host: " (hostname | hostnumber) " \n"
hostname := domainlabels toplabel
domainlabels := (domainlabel /\./)*
domainlabel := alphanum (alphanum alphanum)?
toplabel := alpha (alphanum alphanum)?
alphanum := (alphanum | /\./)*
alphanum := alpha | digit
hostnumber := digits /\./ digits {3}
port := "Port: " digits " \n"
user := "User: " userstr " \n"
password := "Password: " userstr " \n"
userstr := (uchar | /\?&=\/)*
urlpath := "Path: " xchars " \n"
xchars := xchar*

alpha := /[a-zA-Z]/
digit := /[0-9]/
digits := /[0-9]+/
safe := /[\$_.-+~]/
extra := /[!*'(),]/
reserved := /[;\/?:@&=\/]/
escape := /%[0-9A-Fa-f]{2}/
unreserved := alpha | digit | safe | extra
uchar := unreserved | escape
xchar := unreserved | reserved | escape

```

Figure 9: RFC1738 generic URL parser in Kleenex

