Stream Processing Using Grammars and Regular Expressions

Ulrik Terp Rasmussen
DIKU, Department of Computer Science
University of Copenhagen, Denmark

September 25, 2016

PhD Thesis
This thesis has been submitted to the PhD School of the Faculty of Science, University of Copenhagen, Denmark

Abstract

In this dissertation we study expression based parsing and the use of grammatical specifications for the synthesis of fast, streaming string-processing programs.

In the first part we develop two linear-time algorithms for regular expression based parsing with Perl-style *greedy* disambiguation. The first algorithm operates in two passes in a semi-streaming fashion, using a constant amount of working memory and an auxiliary tape storage which is written in the first pass and consumed by the second. The second algorithm is a single-pass and optimally streaming algorithm which outputs as much of the parse tree as is semantically possible based on the input prefix read so far, and resorts to buffering as many symbols as is required to resolve the next choice. Optimality is obtained by performing a PSPACE-complete pre-analysis on the regular expression.

In the second part we present *Kleenex*, a language for expressing high-performance streaming string processing programs as regular grammars with embedded semantic actions, and its compilation to streaming string transducers with worst-case linear-time performance. Its underlying theory is based on transducer decomposition into oracle and action machines, and a finite-state specialization of the streaming parsing algorithm presented in the first part. In the second part we also develop a new linear-time streaming parsing algorithm for *parsing expression grammars* (PEG) which generalizes the regular grammars of Kleenex. The algorithm is based on a bottom-up tabulation algorithm reformulated using least fixed points and evaluated using an instance of the *chaotic iteration* scheme by Cousot and Cousot.

Resumé

I denne afhandling beskæftiger vi os med parsing med regulære udtryk samt anvendelsen af grammatiske specifikationer til syntese af hurtige, strømmende programmer til strengprocessering.

I første del udvikler vi to algoritmer til parsing med regulære udtryk i lineær tid, og med *grådig* afgørelse af flertydigheder i stil med Perl. Den første algoritme består af to faser, der afvikles på en semi-strømmende facon med konstant størrelse arbejdslager, samt et ekstra båndlager der henholdsvis skrives og læses af hver af de to faser. Den anden algoritme består af en enkelt fase og er optimalt strømmende i den forstand, at den udskriver så meget af parse-træet, som det er semantisk muligt ud fra det præfix af inddata, der på det givne tidspunkt er blevet indlæst. Algoritmen falder tilbage til buffering af så mange inputsymboler, som det er nødvendigt for at kunne afgøre næste valg. Optimalitet opnås ved hjælp af en PSPACE-fuldstændig præanalyse af det regulære udtryk.

I anden del præsenterer vi *Kleenex*, et sprog til at udtrykke højtydende, strømmende strengprocesseringsprogrammer som regulære grammatikker med indlejrede semantiske handlinger, samt dets oversættelse til streaming string transducers med worst-case lineær tids ydelse. Den underliggende teori er baseret på dekomponering af transducere i orakel- og handlingsmaskiner, samt en specialisering af den strømmende parsingalgoritme fra den første del som en endelig tilstandsmaskine. I anden del udvikler vi også en ny lineær tids, strømmende parsing algoritme til *parsing expression grammars* (PEG) der generaliserer de regulære grammatikker fra Kleenex. Algoritmen er baseret på en bottom-up tabelopstillingsalgoritme, der reformuleres ved brug af mindste fikspunkter, og som beregnes ved hjælp af en instans af Cousot og Cousots *chaotic iteration*.

Contents

List of Figures

Preface

This dissertation has been submitted to the PhD School of Science, Faculty of Science, University of Copenhagen, in partial fulfillment of the degree of PhD at Department of Computer Science (DIKU).

The dissertation is written as a synopsis of four enclosed research papers, including three peer-reviewed conference papers and one, as of yet, unpublished manuscript. Chapter 1 presents a brief introduction to the two topics of this dissertation. Chapters 2 and 3 each give a more comprehensive overview of the respective topic, including an outline of the area of research, the main problems to be solved, and my contribution in relation to existing work in the literature. Each chapter concludes with a brief outline of the perspectives for future work.

I could not have written this dissertation alone, so at this point I would like to take the opportunity to thank the people who have helped me along the way. First of all, the material presented here is the result of close collaboration with my coauthors, to whom I would like to express my sincere gratitude.

To Fritz Henglein, my supervisor, thank you for giving me both enormous freedom in my research and expert guidance when needed. Your passion and never-ending spirit has been a constant source of inspiration.

To Bjørn, thank you for being a great colleague, friend, office mate and travel companion.

To Dexter Kozen, thank you for hosting me for a wonderful five months at Cornell University. To all of my current and past office mates at both Cornell and DIKU, and to my colleagues in the DIKU APL section, thank you for providing a pleasant and stimulating work environment.

To my family, thank you for your support and understanding during my work on this dissertation.

To Lotte, thank you for everything.

Ulrik Terp Rasmussen

Chapter 1

Introduction

Programmers need to make several trade-offs when writing software. Most important, the software has to be correct while at the same time being able to handle all reasonably expected inputs and usage scenarios. In addition, the underlying implementation should be as simple as possible so that it can be maintained and adjusted without major risks of introducing errors. Furthermore, the software should also be efficient in the sense that requests are answered within a reasonable time frame, and there should be no way to bring the software to use excessive amounts of time and space, either by an adversary or by accident.

In this dissertation, we will focus on programs that process data in its simplest form: strings of symbols. In practice, programs of this kind can be found performing a variety of tasks including processing of user input, decoding of data formats and protocols, automated word processing, and searching over large amounts of sequential data, from log files and source code to sequenced DNA. The one task all programs have in common is the task of deducing the underlying structure of the data in order to be able to process it in a meaningful way. This task can be quite difficult to tackle in itself, and is not made easier by virtue of having to take into account the trade-offs mentioned earlier. There is therefore a need for general solutions and tools that can help overcome these challenges, thereby reducing the time and risk associated with software development.

Any general solution will also have to make trade-offs, so we should not expect a single approach to be able to solve all of our problems once and for all. In this dissertation, we will approach the problem from the perspective of automata theory and formal languages, which already have deep roots in the theoretical and practical aspects of parsing. It is the ultimate goal to provide a new set of methods based on solid foundations which can be used to build string-processing programs with strong performance guarantees, while still being flexible and expressive enough to not increase development costs.

The narrative of this dissertation can roughly be divided into two parts.

Regular Expression Based Parsing

The first part will be concerned with *regular expressions*, an algebraic formalism with a well-understood theory that is commonly used to express patterns of strings. Their conciseness and attractive computational properties have made them popular as a language for expressing string search and input validation programs. Since we rarely validate an input string without the intention of using it, most practical regular expression implementations also provide facilities for breaking up the string into parts based on the specified pattern, a process also known as *parsing*. However, the classical theory does not account for the issues that are normally associated with parsing, and as a result these data extraction facilities have been built as ad-hoc extensions on top of implementations of the classical interpretation of regular expressions as pure string patterns. This approach has missed some opportunities for greater expressivity, and has also resulted in the loss of the attractive performance guarantees that were associated with regular expressions in the first place.

We take a different approach, and work from a generalized theory of regular expressions that take parsing into account. From this perspective we find new algorithmic methods for solving the *regular expression parsing problem*: Given a regular expression and an input string, what is its associated *parse tree*?

Grammar-Based Stream Processing

In the second part we focus on formalisms for specifying string processing programs which operate based on the syntactic structure of their inputs. Programs of this kind perform a range of useful tasks, including advanced text substitution, filtering and formatting of logging data, as well as implementations of data exchange formats. As data of this kind is often generated at a high rate, string processing programs have to operate in a *streaming fashion* where they only store a small part of the input string in memory at any time. Writing and maintaining software which keeps track of the technical details of streaming while also dealing with the complexities of a data format is a challenging task.

We propose the use of *syntax-directed translation schemes* as a suitable formalism for expressing such programs. The formalism based on existing formalisms for describing string patterns, such as regular expressions, extended with *embedded semantic actions*—arbitrary program fragments which are executed based on how a given input string is matched by the specified pattern. We study two different formalisms, and methods for efficiently running specifications written in them in a streaming fashion. The first of these have been used in the design and implementation of the high-performance streaming string processing language *Kleenex*.

Chapter 2

Regular Expression Based Parsing

This chapter is concerned with the problem of parsing using *regular expressions*, which are mathematical expressions for denoting sets of strings, first introduced by Kleene to describe sets of events in mathematical models of the nervous system [51]. After their practical application for text search was pointed out by Thompson [94], regular expressions became a popular language for specifying complex text search patterns. They now enjoy applications in many diverse areas, including text editing [77], querying of data formats [22], detection of code duplication [86] and searching in sequenced DNA data [62]. Their popularity primarily stems from their simplicity, conciseness, and attractive theoretical properties. Most important, a computer program only has to spend time proportional to the length of a string in order to decide if it belongs to the set described by a given regular expression, guaranteeing that a search query will return within reasonable time.

Over the years, implementations have moved away from these theoretical foundations, and the nomenclature "regex" is now informally used to refer to the implemented versions of the original "regular expressions", with which they have little in common apart from syntax. Operators were added in order to increase the number of patterns that could be expressed, notably backreferences and recursion, and mechanisms for limited parsing in the form of *capturing groups* were introduced to accommodate advanced text substitution. Most of these extensions seem to have been added based on what was possible to implement as extensions to the existing search algorithms, and as a result the theoretical properties were lost: Matching a string against a regex can take exponential time in the length of the input, and it is not uncommon to see performance bugs due to seemingly innocent-looking regexes that suddenly trigger this behavior for rare pathological inputs[1,2].

[1] `http://stackstatus.net/post/147710624694/outage-postmortem-july-20-2016`

[2] `http://davidvgalbraith.com/how-i-fixed-atom/`

We will take a different approach, and work from a more general theory of regular expressions which takes the issues related to parsing into account. By changing our perspective on the problem, we reveal new and efficient algorithms for solving the core problem related to the use of regular expressions for data extraction. Furthermore, we will see that the generalization offers an increase in expressivity, enabling new and interesting applications of regular expressions.

We give a semi-formal exposition of the theory of regular expressions in Section 2.1, including its relation to finite automata. In Section 2.2, we show how popular "regex" software packages have extended this theory and discuss the trade-offs. We present the main problem of *regular expression based parsing* in Section 2.3, and relate it to a computational model called *finite transducers* in Section 2.4. In Section 2.5 we review the current approaches to solving this problem, and in Section 2.6 we present our contributions. We conclude this chapter in Section 2.7.

2.1 Regular Expressions In Theory

A regular expression (RE) is a formal mathematical expression using a limited set of operators. Their purpose is to serve as concise specifications of sets of strings with certain desirable properties.

It is assumed that some finite set of symbols Σ, also called the *alphabet*, is given. The alphabet specifies the valid symbols that may occur in the strings described by an RE. For example, Σ could be the set of the 256 bytes that can be represented by 8-bit words, or the full set of Unicode code points—in the remainder of this chapter we will just assume that Σ is the set of lowercase letters $\{\mathtt{a}, \mathtt{b}, ..., \mathtt{z}\}$. The *infinite* set of all strings over Σ is written Σ^*, that is

$$\Sigma^* = \{\varepsilon, \mathtt{a}, \mathtt{b}, ..., \mathtt{z}, \mathtt{aa}, \mathtt{ab}, ..., \mathtt{az}, ..., \mathtt{ba}, \mathtt{bb}, ..., \mathtt{bz}, ...\}$$

and so on, where ε stands for the *empty string*. Of course, appending the empty string to another string u results in the same string again: $\varepsilon u = u = u\varepsilon$. We will generally use letters u, v, w to refer to strings, and will avoid using them as symbols.

The syntax of REs can be compactly described by a generative grammar:

$$E ::= a \mid \epsilon \mid E_1^* \mid E_1 E_2 \mid E_1 + E_2$$

That is, the simplest REs consist of a single symbol a from Σ or the "unit expression" ϵ. Smaller REs E_1, E_2 can be combined to form larger ones by the "star operator" E_1^*, the "sequence operator" $E_1 E_2$ or the "sum operator" $E_1 + E_2$. These are listed in increasing order of precedence, i.e. $\mathtt{ab}^* + \mathtt{c}$ is parenthesized as $(\mathtt{a}(\mathtt{b}^*)) + \mathtt{c}$. Sequence and sum associate to the right, so $E_1 E_2 E_3$ and $E_1 + E_2 + E_3$ parenthesize as $E_1(E_2 E_3)$ and $E_1 + (E_2 + E_3)$, respectively.

The usual interpretation of REs is as denotations of *formal languages*, each of which is a subset of Σ^*. The sets $\{\texttt{cnn}, \texttt{bbc}\}$, $\{\texttt{a}, \texttt{aa}, \texttt{aaa}, ...\}$ and $\emptyset = \{\}$ are all examples of such, where the last is the degenerate case of the *empty language*. In order to define the meaning of REs, we will first need to introduce some operations on languages. Given two languages A and B, we can combine them into a new language AB formed by concatenating every string in A with every string in B, or formally:

$$AB = \{uv \mid u \in A, v \in B\}.$$

For example, if $A = \{\texttt{ab}, \texttt{cd}\}$ and $B = \{\texttt{e}, \texttt{f}\}$, then $AB = \{\texttt{abe}, \texttt{abf}, \texttt{cde}, \texttt{cdf}\}$. Concatenation can be iterated any number of times for a single language: For any number $n \geq 0$, define

$$A^n = \underbrace{AA \cdots A}_{n \text{ times}}$$

where $A^0 = \{\varepsilon\}$ is defined as the language containing just the empty string. For example, if $A = \{\texttt{a}, \texttt{b}\}$, then $A^3 = \{\texttt{aaa}, \texttt{aab}, \texttt{aba}, \texttt{abb}, \texttt{baa}, \texttt{bab}, \texttt{bba}, \texttt{bbb}\}$. The last language operation we will need is also the most powerful. For a language A, write A^* for the language formed by taking any number of strings from A and concatenating them. Formally, this is the language

$$A^* = \bigcup_{n=0}^{\infty} A^n = A^0 \cup A^1 \cup A^2 \cup ...$$

This is a quite powerful operation. For example, if we view the alphabet Σ as a language of single-symbol strings, then Σ^* is exactly the infinite set of all strings containing symbols from Σ. For another example:

$$\{\texttt{ab}, \texttt{c}\}^* = \{\varepsilon, \texttt{ab}, \texttt{c}, \texttt{abc}, \texttt{cba}, \texttt{abab}, \texttt{cc}, \texttt{abcab}, ...\}.$$

Every RE E is a description of a language $\mathcal{L}[\![E]\!]$ which is built using the operations we have just defined. The mapping from syntax to language operators should be quite apparent, and is formally defined as follows:

$$\begin{aligned} \mathcal{L}[\![a]\!] &= \{a\} & \mathcal{L}[\![\epsilon]\!] &= \{\varepsilon\} \\ \mathcal{L}[\![E_1^*]\!] &= \mathcal{L}[\![E_1]\!]^* & \mathcal{L}[\![E_1 E_2]\!] &= \mathcal{L}[\![E_1]\!]\mathcal{L}[\![E_2]\!] \\ \mathcal{L}[\![E_1 + E_2]\!] &= \mathcal{L}[\![E_1]\!] \cup \mathcal{L}[\![E_2]\!] \end{aligned}$$

It can be quite instructive to view an RE E as a *pattern* whose meaning as such is the set of strings $\mathcal{L}[\![E]\!]$ matched by it. This view also hints to their practical use for text search. For example, consider the following pattern:

$$(\texttt{he} + \texttt{she})(\texttt{was} + \texttt{is})((\texttt{very})^* + \texttt{not})(\texttt{happy} + \texttt{hungry} + \texttt{sad})$$

Ignoring the issue of word spacing, this matches an infinite number of variations of sentences of the following kind:

`he was not hungry,` `she is very happy,` `she is very very hungry,` `she is not sad,` `he is very very sad`, ...

While REs offer a lot of expressive power, there are many languages that they cannot express. For example, there is no way to specify the language of all strings of the form

$$\underbrace{\mathtt{aa}\cdots\mathtt{a}}_{n \text{ times}}\underbrace{\mathtt{bb}\cdots\mathtt{b}}_{n \text{ times}}$$

that is, strings with the same number of occurrences of *a*s and *b*s, but with all *a*s occurring before the *b*s. Patterns of this kind may occur in practice in the form of strings of matching parentheses, so surely it would be useful to be able to express them. However, this restriction of expressive power is deliberate. In order to see why, we have to look at the computational properties of REs.

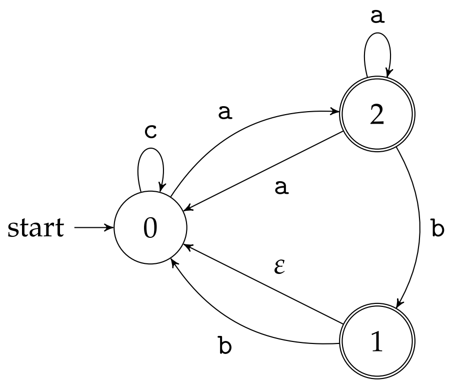
Finite Automata

At this point we have established the semantics of REs, and we have illustrated their power and limitations as a language for constructing string patterns. We now briefly review a general solution to the *recognition problem*:

$$\boxed{\text{Given an RE } E \text{ and a string } u, \text{ is } u \text{ in the language } \mathcal{L}[\![E]\!]?}$$

The limited expressivity of REs turns out to be an advantage when solving this problem, as it allows every RE to be converted into a particularly simple type of program called a *finite automaton* [54].

Automata are usually defined using state diagrams as follows:



The circles are called *states*, and the numbers within are names identifying them. The arrows between states are called *transitions*, and are labeled by either a single symbol or the empty string. A single state is a designated *starting state* and is marked as such. Each state is either *accepting* or *not accepting*, with accepting states drawn as double circles.

With every automaton M is associated a set of strings $\mathcal{L}[\![M]\!]$, in the same way that every RE is associated with one. However, where the language of an RE is defined in terms of language operators, the language of an automaton is defined in terms of a process. A specific set of rules specify how to "run" an automaton on some string u by keeping track of a number of "pebbles" that are placed on the states. The rules are as follows:

1. Place a pebble on the starting state and on any state reachable via one or more ε-transitions.

2. For each symbol in u from left to right:

    a) Pick up all pebbles, remembering what states had pebbles on them.

    b) For every state that had pebbles on it and has a transition matching the next symbol, put a pebble on the destination state.

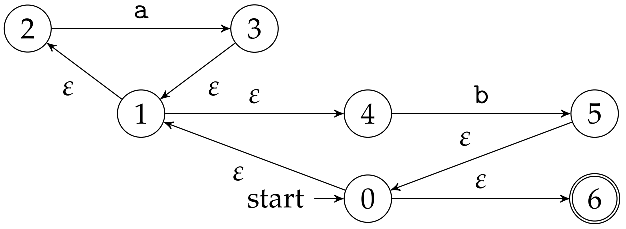
    c) Put a pebble on all states that can be reached via one or more ϵ-transitions from a state with pebbles on it.

If at least one accepting state has pebbles on it when all symbols have been processed, then u is *accepted* by the automaton, and we say that u is in $\mathcal{L}[\![M]\!]$; otherwise it is *rejected*. Note that we are not concerned with the number of pebbles on each state, just that it has a non-zero amount. The following demonstrates an accepting run of the automaton on the string `aabca`:

start	a	a	b	c	a	
$\{0\}$	$\{2\}$	$\{2,0\}$	$\{1,0\}$	$\{0\}$	$\{2\}$	accept

Our interest in finite automata comes from the fact that this process is very efficient to implement on a computer, which only has to look at each input symbol once. Since the set of states with pebbles on them can never be larger than the total amount of states, it will always take time proportional to the length of the string to decide whether it is recognized by the automaton or not.

There is a deep connection between REs and automata, namely that for every RE E there is an automaton M such that $\mathcal{L}[\![E]\!] = \mathcal{L}[\![M]\!]$ [51]. In other words, automata provide the recipe for efficiently "running" regular expressions. For an example, consider the RE $(a^*b)^*$ which has the following associated automaton:



Another remarkable fact is that this connection also holds in the other direction: for every automaton, there is an RE denoting its language. This equivalence with finite automata explains our hesitance towards adding more expressive power to REs, as they have exactly the amount of power they need while still enabling us to use efficient techniques based on finite automata to implement them.

The finite automata presented in this section are also called *non-deterministic finite automata* (NFA), due to the fact that there can be more than one state with pebbles on it when running it. This is done to distinguish them from the special case where exactly on state can ever have pebbles on it, in which case the automaton is called *deterministic* (DFA). Any NFA can be converted to a DFA [80], although this may cause the number of states to increase exponentially. On the other hand, running a DFA is often even faster than running an NFA, so this optimization can pay off when the input size is significantly larger than the RE itself.

2.2 Regular Expressions In Practice

In this section we briefly review some of the history of regex implementations and point out the central differences between them and the REs presented in the previous section. For a more comprehensive account of both the history and features of regexes, we refer to the book by Friedl [38].

REs were popularized in computing from 1968. Thompson [94] pointed out the application of text search and implemented them in the editors QED and `ed`. This later led to the creation of the specialized UNIX search tool `grep`, whose name comes from the `ed` command `g/`*re*`/p` for RE based searching [38]. Also in 1968, Johnson, Porter, Ackley, Ross [48] applied REs for building lexical analyzers for programming language compilers.

In 1986 Harry Spencer wrote the first general regex library for incorporation in other software. This was later adopted and heavily extended by Larry Wall in his Perl [101] programming language, which became popular for practical string manipulation tasks, in part due to regexes being built into its syntax. Its popularity eventually led to the notion of "Perl compatible regexes" (PCRE) for referring to implementations that behaved like Perl's. PCRE regexes have now made it into most other popular programming languages, either built into the language syntax or provided via a library.

There are several important differences between regexes and REs. On the surface, the syntax is slightly different from that of REs: the RE $\mathtt{a}(\mathtt{b}^* + \mathtt{c})\mathtt{d}^*$ would typically be written as the regex `a(b*|c)d*`, with variations depending on the exact flavor. In the following we will highlight the notable semantic differences that separate regex implementations from their theoretical counterparts.

Capturing Groups

The recognition semantics are extended to *matching* via the notion of *capturing groups* which are used to extract information about *how* a string is matched by a regex, and not just whether it is in its language or not. For every parenthesis in a regex, the implementation will report the position of the substring that matched the enclosed subexpression. The user can later refer to the substrings matched by capturing groups by using the names \1, \2, ..., and so on—the groups are numbered from left to right according to their opening parentheses. The following example shows a how a PCRE implementation will report matches when given the input aaaabbbb:

$$\texttt{a*(aa(bb|bbb))b*} \qquad \texttt{a a } \overbrace{\texttt{a a } \underbrace{\texttt{b b}}_{\texttt{\textbackslash 2}}}^{\texttt{\textbackslash 1}} \texttt{ b b}$$

The example illustrates that even though capturing groups seem benign, they introduce a new problem of *ambiguity*. The second capturing group could also have matched the substring bbb, but the left alternative was chosen by the implementation. This behavior is shared by all PCRE implementations, which prefer left alternatives over right ones. However, a second popular flavor of regex specified by the POSIX standard [47] would pick the right alternative based on a policy which maximizes the lengths of submatches from left to right. The differences in matching semantics of different regex flavors has resulted in much confusion among users, with the issue further exacerbated by the fact that the POSIX regex specification is unnecessarily obscure, leading to a situation where no two tools claiming to be POSIX actually report the same results for all inputs [37, 72, 55, 91].

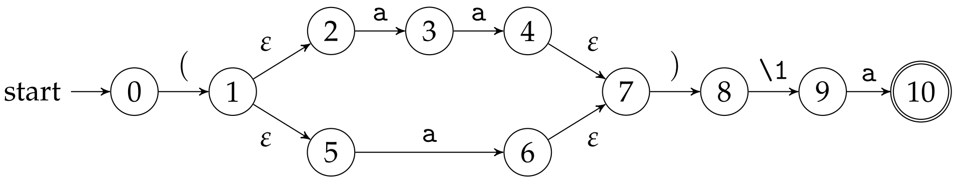
Another limitation of capturing groups is that they do not work well when combined with the star operator. For example, when the regex (a*b)* is matched against input abaaab, it is ambiguous whether the implementation should return ab or aaab for the substring \1. In some usage scenarios, the user might even want the position of both submatches (e.g. for reading a list of data items), but that is not possible under the regex matching paradigm.

Backreferences and Backtracking

Capturing groups enable another regex extension called *backreferences* that allow users to write \1, \2, ... to refer back to the substrings matched by capturing groups earlier in the pattern. For example, the regex (a*)b\1 will match any string of the form

$$\underbrace{\texttt{aa}\cdots\texttt{a}}_{n\text{ times}}\texttt{b}\underbrace{\texttt{aa}\cdots\texttt{a}}_{n\text{ times}}$$

No RE or finite automaton can express such a language. Implementations with backreferences typically work by using an extended form of automata where transitions can mark the beginnings and endings of a capturing groups, and other transitions can refer back to them. Consider the regex `(aa|a)\1a`, which has the following extended automaton:



We can run it using a strategy called *backtracking*, which resembles the pebble strategy from the previous section, but with the restriction that at most one state can have a pebble on it at any time. When the pebble can move to more than one state, we choose one arbitrarily and remember the state we came from in case we need to go back and try another one. If at any point the pebble cannot move to a new state but more symbols are in the string, then we backtrack to the last choice point and try the next alternative. This process is repeated until all input symbols have been consumed and the pebble is on an accepting state, or until all possible pebble movements have been exhausted. We write down the position in the input string whenever the pebble crosses an opening and closing parenthesis. When the pebble tries to transition a backreference, it uses these positions to check whether the remainder of the string matches the substring given by the last recorded positions.

For example, on input `aaa` in the automaton depicted above, the pebble first moves along the states 0,1,2,3,4,7,8, recording that the group `\1` matches `aa`. But now the pebble cannot move from 8 to 9, since only `a` is left, and `aa` is needed. It backtracks to the last choice (state 1) and now continues along states 5,6,7,8, recording that `\1` matches `a`. Since `aa` remains, the pebble can now move along states 9 and 10, and the automaton accepts.

This strategy works, and is used in PCRE style regex implementations. They agree on disambiguation by systematically always trying left alternatives first. For this reason, the PCRE disambiguation policy is also called *greedy* or *first-match* disambiguation. Although backtracking is usually fast in practice, it has the disadvantage that some regexes and inputs can make it spend an atrocious amount of time—exponential in the length of the input string.

As stated in the introduction of this chapter, the exponential time worst case does occur in practice, and also opens up systems to attacks by adversaries who construct input data to deliberately trigger the worst-case behavior of certain regexes. This type of attack, known as *regular expression denial-of-service* (REDoS) [79], has motivated a lot of recent research in methods for identifying vulnerable regexes [50, 89, 13, 81, 102].

2.3 Regular Expression Based Parsing

After reading the previous sections it should be clear that regex implementations make trade-offs between expressivity and predictable performance. These trade-offs are perfectly acceptable for situations like ad-hoc text manipulation and automation of non-critical tasks, but as we pointed out there are also scenarios where we need greater matching expressivity (capturing groups under the star operator) and/or hard performance guarantees (exponential time performance bugs in mission-critical systems). In this section we address the first issue by presenting a generalization of the language semantics for REs into one of *parsing*. The goal of formulating a new semantics for REs is to separate the specification of what we want to solve from its actual implementation, giving us freedom to try out different approaches. The second issue will then be addressed by finding new ways to implement the parsing semantics efficiently, which is the main topic of our contributions.

Terms and Types

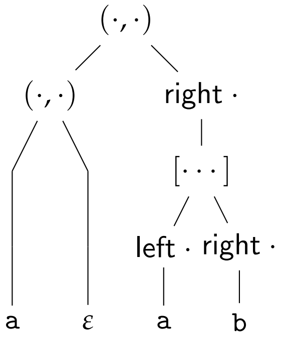
Before we can define the parsing semantics for REs, we need to introduce the notions of *terms* and *types*.

Terms act as generalizations of strings with extra information about how they are matched. They can be compactly described by the following generative grammar:

$$V ::= a \mid \varepsilon \mid (V_1, V_2) \mid \mathsf{left}\ V_1 \mid \mathsf{right}\ V_2 \mid [V_1, V_2, ..., V_n]$$

In other words, symbols a and the empty string ε are the smallest terms. Larger terms can be build as follows: if V_1, V_2 are terms, then (V_1, V_2) is a term; left V_1 is a term; and right V_2 is a term. Finally, if $V_1, V_2, ..., V_n$ are terms, then the list $[V_1, V_2, ..., V_n]$ is a term. In particular, the empty list [] is a term.

Terms have structure, and can easily be decomposed into smaller parts by a program. It is instructive to view them as upside-down trees, with the smallest terms at the bottom. For example, the term $V = ((\mathtt{a}, \varepsilon), \mathsf{left}\ [\mathsf{left}\ \mathtt{a}, \mathsf{right}\ \mathtt{b}])$ has the following tree structure:



This illustrates how terms are merely strings with more structure: by removing anything above the symbols at the bottom, we are left with the string $\mathtt{a}\varepsilon\mathtt{ab} = \mathtt{aab}$. This is also called the *flattening* of V, and we write $|V| = \mathtt{aab}$.

We write Term_Σ to refer to the infinite set of all terms over the alphabet Σ. We will call subsets T of terms for *types*, analogously to the way we called subsets of Σ^* languages in the previous sections. We also need to define analogous operations for concatenation, union and star.

Given two types T and U, write $T \times U$ for their *product* which is the set of all pairs whose first and second components are from T and U, respectively. Formally, $T \times U = \{(V_1, V_2) \mid V_1 \in T, V_2 \in U\}$. For example,

$$\begin{aligned}&\{\mathsf{left}\ \mathtt{a}, \mathsf{right}\ \mathtt{b}\} \times \{[], [\mathtt{a}], [\mathtt{a}, \mathtt{a}], \ldots\} \\ &\quad = \{(\mathsf{left}\ \mathtt{a}, []), (\mathsf{right}\ \mathtt{b}, []), (\mathsf{left}\ \mathtt{a}, [\mathtt{a}]), (\mathsf{right}\ \mathtt{b}, [\mathtt{a}]), \ldots\}\end{aligned}$$

The product operation is analogous to the concatenation operation AB for languages, but with the difference that it reveals whenever there are multiple ways of obtaining the same string. Consider the following example, which shows language concatenation in the top and type product in the bottom:

$$\begin{aligned}\{\mathtt{a}, \mathtt{ab}\}\{\mathtt{bc}, \mathtt{c}\} &= \{\mathtt{abc}, \mathtt{ac}, \mathtt{abbc}\} \\ \{\mathtt{a},\ (\mathtt{a}, \mathtt{b})\} \times \{(\mathtt{b}, \mathtt{c}),\ \mathtt{c}\} &= \{(\mathtt{a}, (\mathtt{b}, \mathtt{c})),\ (\mathtt{a}, \mathtt{c}),\ ((\mathtt{a}, \mathtt{b}), (\mathtt{b}, \mathtt{c})),\ ((\mathtt{a}, \mathtt{b}), \mathtt{c})\}\end{aligned}$$

The string abc can be formed either by picking a from the left language and bc from the right; or by picking ab and then c. The two are indistinguishable, so the result is a language with three strings instead of four. For products, the two cases can be distinguished, and we obtain a four-element type.

Given two types T and U, write $T + U$ for their *sum* which is the set all terms of the form left V_1 and right V_2, where V_1 is in T and V_2 is in U. Formally, $T + U = \{\mathsf{left}\ V_1 \mid V_1 \in T\} \cup \{\mathsf{right}\ V_2 \mid V_2 \in U\}$. For example,

$$\begin{aligned}&\{\mathsf{left}\ \mathtt{a}, \mathsf{right}\ \mathtt{b}\} + \{[], [\mathtt{a}], [\mathtt{a}, \mathtt{a}], \ldots\} \\ &\quad = \{\mathsf{left}\ (\mathsf{left}\ \mathtt{a}), \mathsf{left}\ (\mathsf{right}\ \mathtt{b}), \mathsf{right}\ [], \mathsf{right}\ [\mathtt{a}], \mathsf{right}\ [\mathtt{a}, \mathtt{a}], \ldots\}\end{aligned}$$

The sum operation is analogous to the union operation $A \cup B$ for languages, but with the difference that it records from which operand a given element comes from, ensuring that no two elements are conflated. Consider the following example, which shows language union on top and type sum in the bottom:

$$\begin{aligned}\{\mathtt{a}, \mathtt{ab}\} \cup \{\mathtt{ab}, \mathtt{ac}\} &= \{\mathtt{a}, \mathtt{ab}, \mathtt{ac}\} \\ \{\mathtt{a},\ (\mathtt{a}, \mathtt{b})\} + \{(\mathtt{a}, \mathtt{b}),\ (\mathtt{a}, \mathtt{c})\} &= \{\mathsf{left}\ \mathtt{a},\ \mathsf{left}\ (\mathtt{a}, \mathtt{b}),\ \mathsf{right}\ (\mathtt{a}, \mathtt{b}),\ \mathsf{right}\ (\mathtt{a}, \mathtt{c})\}\end{aligned}$$

The string ab is in both languages, but the union only contains the string once, resulting in a three-element language. For sums, the elements are tagged by the side they came from and are thus not conflated, resulting in a four-element type.

Finally, if T is a type, then $\mathsf{List}(T)$ is the set of all lists whose elements are from T. Formally, $\mathsf{List}(T) = \{[V_1, V_2, ..., V_n] \mid n \geq 0, \text{for all } i \leq n.\, V_i \in T\}$. For example, if T is as in the previous examples, then

$$\mathsf{List}(T) = \{[], [\mathsf{left}\ \mathtt{a}], [\mathsf{right}\ \mathtt{b}], [\mathsf{left}\ \mathtt{a}, \mathsf{left}\ \mathtt{a}], [\mathsf{left}\ \mathtt{a}, \mathsf{right}\ \mathtt{b}], [\mathsf{right}\ \mathtt{b}, \mathsf{left}\ \mathtt{a}], ...\}$$

The list operation is analogous to the star operator for languages. The difference between the two is similar to the difference between concatenation and product.

Languages can be understood as string patterns. Types are also a form of string patterns, but where we also care about *how* a given string is in the pattern—this is explained by the terms that flatten to the string. The correspondence between languages and types is summarized in the table below:

Strings & Languages		Terms & Types	
All strings	Σ^*	All terms	Term_Σ
Languages	$A \subseteq \Sigma^*$	Types	$T \subseteq T_\Sigma$
Concatenation	AB	Product	$T \times U$
Union	$A \cup B$	Sum	$T + U$
Star	A^*	List	$\mathsf{List}(T)$

Regular Expressions as Types

We are now ready to define a parsing semantics for REs. Using the framework of terms and types that we have set up in the previous, we associate every RE E with a type $\mathcal{T}[\![E]\!]$ as follows:

$$\begin{aligned} \mathcal{T}[\![a]\!] &= \{a\} & \mathcal{T}[\![\epsilon]\!] &= \{\varepsilon\} \\ \mathcal{T}[\![E_1^*]\!] &= \mathsf{List}(\mathcal{T}[\![E_1]\!]) & \mathcal{T}[\![E_1 E_2]\!] &= \mathcal{T}[\![E_1]\!] \times \mathcal{T}[\![E_2]\!] \\ \mathcal{T}[\![E_1 + E_2]\!] &= \mathcal{T}[\![E_1]\!] + \mathcal{T}[\![E_2]\!] \end{aligned}$$

There is a close connection between the traditional language interpretation and the type interpretation of REs. Namely, for every RE E, if u is a string in its language $\mathcal{L}[\![E]\!]$, then there is a term V in its type $\mathcal{T}[\![E]\!]$ with flattening u, that is $|V| = u$. Vice versa, for any term T in $\mathcal{T}[\![E]\!]$, its flattening $|V|$ can also be found in $\mathcal{L}[\![E]\!]$.

The benefit of this change of perspective is that the type interpretation of REs accounts for ambiguity, whereas this aspect is hidden in the language interpretation and only arises as a property of the concrete implementations. For example, consider the two REs $\mathtt{a}(\mathtt{a} + \mathtt{b})^*$ and $(\mathtt{ab} + \mathtt{a})(\mathtt{a} + \mathtt{b})^*$. They have the same languages, $\mathcal{L}[\![\mathtt{a}(\mathtt{a} + \mathtt{b})^*]\!] = \mathcal{L}[\![(\mathtt{ab} + \mathtt{a})(\mathtt{a} + \mathtt{b})^*]\!]$, but not the same types:

$$\mathcal{T}[\![\mathtt{a}(\mathtt{a} + \mathtt{b})^*]\!] \neq \mathcal{T}[\![(\mathtt{ab} + \mathtt{a})(\mathtt{a} + \mathtt{b})^*]\!].$$

The type interpretation captures the fact that there is only one term with flattening aba in the type of the first RE, while there are two such terms in the type of the second, as illustrated in Figure 2.1.

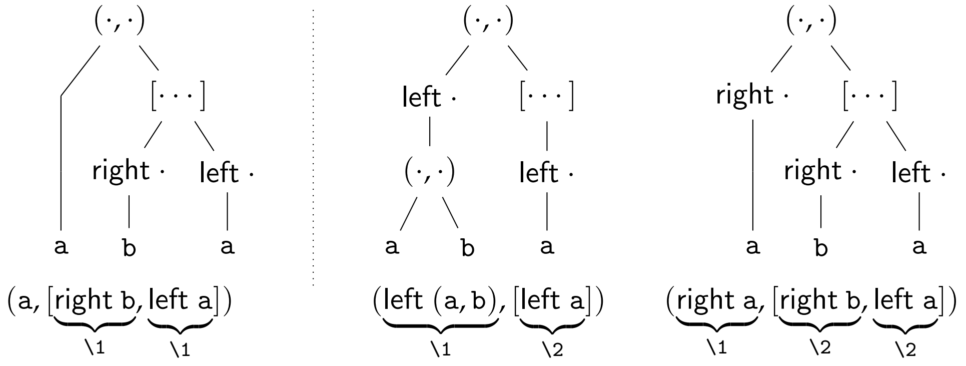

$(\mathsf{a}, [\underbrace{\mathsf{right}\ \mathsf{b}}_{\backslash 1}, \underbrace{\mathsf{left}\ \mathsf{a}}_{\backslash 1}])$
$(\underbrace{\mathsf{left}\ (\mathsf{a}, \mathsf{b})}_{\backslash 1}, [\underbrace{\mathsf{left}\ \mathsf{a}}_{\backslash 2}])$
$(\underbrace{\mathsf{right}\ \mathsf{a}}_{\backslash 1}, [\underbrace{\mathsf{right}\ \mathsf{b}}_{\backslash 2}, \underbrace{\mathsf{left}\ \mathsf{a}}_{\backslash 2}])$

Figure 2.1: Terms with flattening aba for REs $\mathtt{a}(\mathtt{a}+\mathtt{b})^*$ and $(\mathtt{ab}+\mathtt{a})(\mathtt{a}+\mathtt{b})^*$.

A term fully "explains" how its flattened string can be parsed according to the given RE, including information pertaining to the star operator. As such, we will therefore also refer to terms as *parse trees*. The information contained in parse trees transcends the information provided by capturing groups in a regex implementation. As the example illustrates, the groups \1 in the first RE and \2 in the second cannot be assigned unique substrings since they both occur under a star operator, whereas the parse tree just contains a list of all the possible assignments.

The recognition problem introduced in Section 2.1 can now be generalized to the *parsing problem*:

> Given an RE E and an input string u, is there a parse tree V in $\mathcal{T}[\![E]\!]$ such that $|V| = u$?

Unlike the recognition problem, the parsing problem has more than one solution due to the possibility of ambiguous choices for the parse tree V. Different strategies for picking such a parse tree are analogous to the solutions to the disambiguation problem for regex matching, and it is possible to give definitions that are compatible with both PCRE [39] and POSIX [91] disambiguation.

2.4 Parsing as Transduction

The finite automata used as the computational model for the language interpretation of REs are not expressive enough for the type interpretation. Since a finite automaton can only ever accept or reject an input, it does not support the construction of a parse tree. There is, however, another model called *finite transducers* [15, 68] which can. In order to connect the type interpretation to this machine model, we need to first introduce the concept of *bit-coding* [70].

Bit-Coding

Bit-coding can be seen as a way of writing down a parse tree for an RE as a flat string, but in a way such that the parse tree can easily be recovered again. The coding scheme is based on the observation that if an RE is of the form $E_1 + E_2$, then any of its parse trees must be of one of the two forms $\mathsf{left}\ V_1$ or $\mathsf{right}\ V_2$. A single number, say 0 or 1, can be used to specify which of the respective shapes the parse tree has, and we are left with the problem of finding a code for one of the subtrees V_1 or V_2. Similarly, every parse tree for an RE E_1^* is a list, and we can again use the symbols 0 and 1 to indicate whether the list is non-empty or empty, respectively. In the latter case there is only one possible list (the empty list), and in the first case, we are left with the problem of finding codes for the first element and the rest of the list. For a product E_1E_2, every parse tree is of the form (V_1, V_2), so all we have to do is find codes for the two subtrees. Similarly, for the remaining constructs and a, ϵ there is only one possible parse tree, so no coding is needed to specify which one it is.

Formally, for any parse tree V, we define $\mathsf{code}(V)$ as a string of bits, that is a string in $\{0,1\}^*$, as follows:

$$\begin{aligned}
\mathsf{code}(a) &= \varepsilon \\
\mathsf{code}(\varepsilon) &= \varepsilon \\
\mathsf{code}([V_1, V_2, ..., V_n]) &= 0\ \mathsf{code}(V_1)\ 0\ \mathsf{code}(V_2)\ ...\ 0\ \mathsf{code}(V_n)\ 1 \\
\mathsf{code}((V_1, V_2)) &= \mathsf{code}(V_1)\mathsf{code}(V_2) \\
\mathsf{code}(\mathsf{left}\ V_1) &= 0\ \mathsf{code}(V_1) \\
\mathsf{code}(\mathsf{right}\ V_2) &= 1\ \mathsf{code}(V_2)
\end{aligned}$$

For example, the term $(\mathsf{right}\ \mathtt{a}, [\mathsf{right}\ \mathtt{b}, \mathsf{left}\ \mathtt{a}])$ which is a parse tree for $(\mathtt{ab} + \mathtt{a})(\mathtt{a} + \mathtt{b})^*$ and depicted as the third tree in Figure 2.1 has the following bit-code:

$$\begin{aligned}
\mathsf{code}((\mathsf{right}\ \mathtt{a}, [\mathsf{right}\ \mathtt{b}, \mathsf{left}\ \mathtt{a}])) &= \mathsf{code}(\mathsf{right}\ \mathtt{a})\ \mathsf{code}([\mathsf{right}\ \mathtt{b}, \mathsf{left}\ \mathtt{a}]) \\
&= 1\ \mathsf{code}([\mathsf{right}\ \mathtt{b}, \mathsf{left}\ \mathtt{a}]) \\
&= 1\ 0\mathsf{code}(\mathsf{right}\ \mathtt{b})\ 0\mathsf{code}(\mathsf{left}\ \mathtt{a})\ 1 \\
&= 1\ 01\ 00\ 1
\end{aligned}$$

A code can easily be decoded again to obtain the original parse tree. That is, for every RE E there is also a function decode_E which takes a bit-code and returns the parse tree that we started out with. In other words, we have $\mathsf{decode}_E(\mathsf{code}(V)) = V$. We will not need the definition of decoding for this presentation, and refer to Nielsen and Henglein [70] for details.

Transductions

By treating parse trees as codes, we can now connect the type interpretation of REs with another type of finite automata called *finite transducers* [15]. These are finite automata extended such that every transition is now labeled by a pair a/c, where a is an *input label* and c is an *output label*. Input labels are symbols from Σ as before, or the empty string ε. Output labels are strings over some output alphabet Γ. A string u is accepted by a transducer if there is a path from the initial to the final state such that the concatenation of all the input labels along the path equals u. Furthermore, every such path is associated with a corresponding output string obtained by concatenating all the output labels in the same way. This justifies the name *transducers*, as they model a simple form of string translators. When transducers are used for parsing, we will have $\Gamma = \{0, 1\}$ as we will be translating input strings to bit-codes.

It can be shown that every RE has a transducer which accepts the strings in its language and furthermore outputs all the bit-codes of the corresponding parse trees with the same flattening [70]. For an example, see the transducer for the RE $(\mathtt{ab} + \mathtt{a})(\mathtt{a} + \mathtt{b})^*$ in Figure 2.2. It can be seen that this machine generates two codes for the input aba, corresponding to the two parse trees on the right in Figure 2.1.

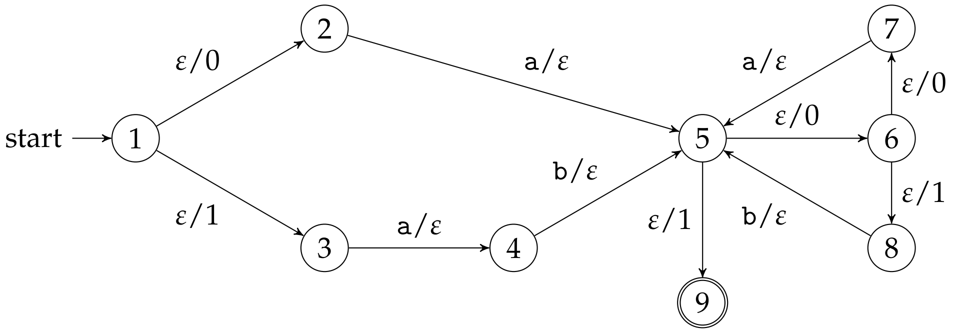


Figure 2.2: A bit-coded parsing transducer for the RE $(\mathtt{ab} + \mathtt{a})(\mathtt{a} + \mathtt{b})^*$.

Running a non-deterministic transducer is not as straightforward as using the pebble method for automata, since every pebble is now associated with the output string generated along its path. Since there can be an exponential number of different ways to get to a particular state, a pebble strategy will have to limit the number of active pebbles on each state to at most one in order to ensure linear running time. This corresponds to disambiguation of parses when more than one parse of a string is possible.

Bit-coded parsing transducers provide another perspective on the parsing problem which now becomes:

Given an RE E and a string u, is there an accepting path with input u and output v in the parsing transducer?

2.5 Recognition, Matching and Parsing Techniques

We have discussed three different problems pertaining to REs: recognition, matching and parsing. The first is formulated in terms of the language semantics of REs, the second arises in concrete implementations and the third is formulated in terms of the type interpretation. The answers to each problem provide increasing amounts of information, as summarized in the following table:

Problem	Solution
Recognition	Accept/Reject
Matching	Accept/Reject and disambiguated captures
Parsing	Accept/Reject and disambiguated parse tree

In this section we review the work that has been done on techniques for solving the above.

Recognition and Matching

For pure recognition, the NFA ("multi-pebble") and DFA ("single-pebble") based techniques described in this chapter are well-known [54, 1]. The construction of NFAs from REs is generally attributed to Thompson [94], and McNaughton and Yamada [64]. Instead of automata, one can also use Brzozowski [20] or Antimirov [9] derivatives. These are syntactic operators on REs which correspond to removing a single letter from all strings in the underlying language. The recognition problem can then be reduced to taking iterated derivatives and checking whether the resulting RE contains the empty string. Implementations of regex matching not based on automata or derivatives are generally based on backtracking which has already been covered earlier in this chapter.

The first implementation of regex matching appeared in Pike's `sam` editor [77]. The method was not based on backtracking, but tracked the locations of capturing groups during running of the NFA. According to Cox [24], Pike did not know that his technique was new and thus did not claim it as such. Laurikari [57] later rediscovered it and formalized it using an extension of NFAs with *tags* for tracking captures, and also gave a method for converting tagged NFAs to tagged DFAs.

While NFA based approaches ensure linear time, they are not as fast as DFAs, which on the other hand can get very big for certain REs. Cox [25] describes a way of constructing DFAs on the fly while running the NFA. The method obtains the performance benefits of DFAs without risking an exponential blowup of the number of states during conversion, and gracefully falls back to using only the NFA when too many DFA states are encountered. It is implemented in the RE2 library [26, 93] which also supports matching via capturing groups, although the fast DFA technique supports at most one group. Other approaches to augmenting NFAs with information about capturing groups exist [32, 43], with a particularly elegant one due to Fischer, Huch and Wilke [32], implemented in the Haskell [46] programming language. It avoids explicitly constructing an NFA by treating the nodes in the RE syntax tree as states in a Glushkov [40] automaton.

It is also possible to perform RE matching without the use of finite automata by applying RE derivatives. This is a popular approach for implementations in functional programming languages where the representation of finite automata can be cumbersome [75]. Sulzmann and Lu [90] give an RE matching algorithm by extending Brzozowski and Antimirov derivatives to keep track of partially matched capturing groups added to the syntax of REs, and they give variants of the method for both POSIX and PCRE disambiguation.

Parsing

Via General Parsing Techniques

The RE formalism is subsumed by more general language formalisms such as *context-free grammars* (CFG) which are capable of expressing non-regular languages such as $a^n b^n$. Methods for parsing with CFGs can therefore also be applied to solve the RE parsing problem, but due to their generality they cannot take advantage of the limited expressivity. The literature on CFG parsing algorithms is vast [42], but they can generally be divided into two categories: *deterministic* and *general* algorithms.

General CFG parsing algorithms include CYK [103], Earley [29] and GLR [96], and they can parse all CFGs regardless of ambiguity, including REs. The result is often a set of all possible parse trees, with disambiguation deferred to the consumer of the algorithm. The disadvantage of using general CFG algorithms for RE parsing is first of all that the worst-case running time is non-linear, a situation which is theoretically impossible to improve [58]. Furthermore, we are rarely interested in the set of all parse trees and would rather prefer disambiguation to be built in.

Deterministic CFG parsing algorithms include LR(k) [52] and LL(k) [60], and they guarantee linear time complexity at the expense of only working for a strict subset of CFGs which are *deterministic* (choices are resolved by looking at most k symbols ahead in the input) relative to the strategy employed by the respective algorithms. The result is always a single parse tree, as ambiguity is ruled out by the determinism restriction. This unfortunately also rules out all ambiguous REs, so deterministic CFG parsing will only work for unambiguous RE subclasses such as one-unambiguous REs [19].

Ostrand, Paull and Wcyuker [73] restrict themselves to regular CFGs and devise an algorithm for deciding whether such a grammar is FL(k) for some k, where FL(k) means that at most k next symbols have to be examined in order to resolve any choice. They give two linear time algorithms which can parse any FL(k) grammar while producing the parse tree on the fly. In the case where unbounded lookahead is required, the latter algorithm still works but may use non-linear time.

Another general language formalism is Ford's [36] *parsing expression grammars* (PEG), which can also express any RE [66]. Contrary to CFG parsing, PEG parsing can actually be done in linear time [2, 35] and always yields a unique parse tree consistent with the disambiguation policy of PCRE. The known linear time parsing algorithms use quite a lot of memory, however, which is again a consequence of the generality of PEGs.

Pure RE Parsing

Most automata-based RE parsing algorithms operate in two separate passes, where the first pass runs over the input string and the second runs over an auxiliary data structure produced during the first pass. We will classify such methods as being either "forwards-backwards" or "backwards-forwards" depending on the direction of these runs.

Kearns [49] devised the first known pure RE parsing algorithm which operates by running the NFA in reverse while journaling the sets of active states in each step. If the run succeeds then the journal is traversed again in order to construct the parse tree. It is thus a backwards-forwards algorithm.

Dubé and Feeley [28] gave the first method based on NFAs whose transitions are annotated with actions for constructing parse trees. Under this view, NFA runs also produce a parse tree whenever the machine accepts, but since many paths are explored at once in the forward simulation, the problem becomes finding the one that lead to acceptance. Their forwards-backwards algorithm builds a DFA without actions and runs it while journaling the sequence of visited states. If the DFA accepts, the journal can be traversed again to reconstruct a single NFA path using a precomputed lookup table. By executing the actions on this path, the corresponding parse tree is obtained.

Neither of the methods by Kearns or Dubé and Feeley are concerned with implementing a specific formal disambiguation policy. Kearns implements a policy which seems to resemble that of PCRE, but he never proves them equivalent. Dubé and Feeley encode disambiguation in the lookup table which is not uniquely characterized, and so disambiguation is left to the implementation. This situation was resolved by Frisch and Cardelli [39] who independently rediscovered the backwards-forwards method of Kearns, but also formalized PCRE disambiguation in terms of parse trees and proves that the method actually implements this policy. They also gave a satisfying solution to the problem of dealing with so-called *problematic REs* which cause naïve backtracking implementations to run forever, and thus also pose a problem for a formal account of PCRE disambiguation. Their solution also handles problematic REs, but in a way which gives the same results as backtracking search in all cases where it terminates.

The formalization by Frisch and Cardelli seems to be the first mention of the type interpretation of REs. This interpretation is further investigated by Henglein and Nielsen [44] who use it to give a sound and complete reasoning system for proving RE containment (is the language of one RE contained in another?). Their system has a computational interpretation as coercions of parse trees and also admits an encoding of other reasoning systems for RE containment [85, 53, 41], equipping them with a computational interpretation as well. They also introduce the *bit-coding* of parse trees described in Section 2.3. Nielsen and Henglein [70] show that the forwards-backwards method of Dubé and Feeley and the backwards-forwards method of Frisch and Cardelli can both be modified to emit bit-codes instead of materializing the parse trees.

Parsing with the POSIX "leftmost-longest" disambiguation policy is significantly more difficult than parsing with the PCRE "greedy" policy. Okui and Suzuki [71, 72] give the first forwards-backwards algorithm for disambiguated RE parsing using POSIX disambiguation. It runs in linear time, but with a constant that is quadratic in the size of the RE. The correctness proof of the algorithm is also significantly more intricate than the previous methods discussed here, which confirms the impression that the POSIX policy is in fact more difficult to implement than PCRE, at least for automata based techniques. Sulzmann and Lu [91] formulate POSIX disambiguation as an ordering relation on parse trees and give an alternative forwards-backwards parsing algorithm based on Brzozowski [20] derivatives, as well as a forwards parsing algorithm using only a single pass which produces a bit-code representation of the parse tree.

Borsotti, Breveglieri, Reghizzi and Morzenti [16, 17] recently gave a forwards-backwards parser based on an extension of the Berry-Sethi [14] algorithm for constructing DFAs from REs. The parser can be configured for both POSIX and PCRE disambiguation by only changing the choices made in the second backwards pass, giving a common framework which can accommodate both policies.

Connection to Transducers

It is remarkable that every automaton based method for RE parsing seems to operate in two passes. By applying the interpretation of parsing as transduction from Section 2.4, it seems that one should be able to obtain a single-pass parser by turning the non-deterministic parsing transducer into a deterministic one, just as an NFA can be converted to an equivalent DFA. This is however not possible in general, as non-deterministic transducers are strictly more powerful than deterministic ones [15, ex. IV.2.3]. This implies that any deterministic RE parsing machine must be more powerful than finite transducers.

One the other hand, an old result by Elgot and Mezei [30][15, Theorem 5.2] says that every unambiguous transducer can be run in two passes, where each pass is modeled by a deterministic transducer. The first pass runs in the forwards direction, producing an auxiliary string over an intermediate alphabet, and the second pass runs in the opposite direction over this string to produce the reversed output. This is exactly the forwards-backwards model employed by the automata based parsing methods, which can all be seen as rediscoveries of this old result.

Disambiguation Policies

Since most practical REs are ambiguous, any method for RE matching or parsing must employ a disambiguation policy, which furthermore must have a semantics that is transparent to the user. Defining a disambiguation policy which is both efficient to implement and easy to comprehend is not an easy task. The formal definitions by Vansummeren [97] of various common disambiguation policies, including those employed in PCRE and POSIX, provide a good comparison of their different qualities.

Myers, Oliva and Guimaraes [69] argue that the PCRE and POSIX disambiguation policies are not intuitive, since they are inherently tied to the structure of the underlying NFA instead of more meaningful semantic criteria such as "maximize the total length of substrings captured in all capturing groups". They give a method which in an NFA can select the path that corresponds to either maximizing the length of all captured substrings, or the individual lengths of the leftmost ones.

Although semantic policies are ostensibly more intuitive, they seem to have been largely ignored in most work on RE matching and parsing. Apart from the difficulty of obtaining efficient implementations (Myers' method runs in linear time, but with a large constant overhead), a possible hindrance to adoption could be that most users have familiarized themselves with REs through existing PCRE or POSIX tools, and so this is the behavior that they have come to expect.

2.6 Our Contributions

The first two papers of this dissertation are concerned with regular expression based parsing using a PCRE "greedy" disambiguation policy. Our approaches are both based on finite automata annotated with bit-codes à la Nielsen and Henglein [70] and offer, respectively, improved memory and time usage by lowered constant factors compared to existing methods, as well as a new streaming execution model for parsing.

In Paper A we present a new algorithm for RE parsing which operates in two passes similar to the forwards-backwards algorithms mentioned in the previous section, producing a reversed bit-code representation of the greedy parse tree in the second pass. The first pass runs the NFA in the forwards direction while maintaining an ordered *list* of active states instead of a set, where the ordering of states in the list denote their ranking according to the disambiguation policy. In each step of the forward run we save k bits of information in a *log*, where $k < \frac{1}{3}m$ and m is the number of states in the NFA. In the second pass the log is traversed in opposite order in order to reconstruct the greedy parse tree. Our algorithm is a variant of the method of Dubé and Feeley [28] with disambiguation, and using less storage for the log—we only save k bits per input character instead of the full set of active states which requires m bits. We also avoid having to build a DFA and thus avoid the risk of an exponential number of states. We compare the performance of a prototype C implementation with RE2 [26], Tcl [74], Perl [101], GNU `grep` as well as the implementations by Nielsen and Henglein [70] of the methods of Dubé and Feeley [28] and Frisch and Cardelli [39]. It performs well in practice, and is surprisingly competitive with tools that only perform matching such as RE2 and `grep`.

Paper B takes a new approach and presents a linear time parsing algorithm which also performs bit-coded PCRE disambiguated parsing, but using only a single forward pass. Furthermore, the parse is produced in an *optimally streaming* fashion—bits of the output is produced as early as is semtically possible, sometimes even before the corresponding input symbols have been seen. For REs where an unbounded amount of symbols need to be consumed in order to resolve a choice, such as the RE $\mathtt{a}^*\mathtt{b} + \mathtt{a}^*\mathtt{c}$, the algorithm automatically adapts to buffering as many `a`s as needed, and immediately outputs the bit-code as soon as a `b` or `c` symbol is encountered. In order to obtain optimal streaming a PSPACE-hard analysis is required, adding a worst-case $O(2^{m \log m})$ preprocessing phase to the algorithm, although this must only be done once for the RE and is independent of the input. The main idea of the method is to maintain a *path tree* from the initial state to all states that can be reached by reading the input read so far, where a branching node in the tree represents the latest point at which two paths diverge. The longest unary branch from the root of the tree thus represents the path prefix that must be followed by *all* viable paths reading a completion of the input seen so far. The path tree model was also used by Ostrand, Paull and Weyuker [73] in their FL(k) parser, albeit without support for linear-time parsing with unbounded lookahead and with a more primitive condition for resolving choices.

2.7 Conclusions and Perspectives

We will hope that by the end of reading this chapter, it has become clear that the area of regular expressions still contains interesting problems despite their well-understood language theory and long list of practical applications. By taking a step back to properly identify the core problem that is being solved in practical regex tools, namely *parsing*, we obtain a new perspective from which new and interesting solutions can be uncovered.

We present two new methods for regular expression based parsing. The first improves on previous methods, while the second appears to be the first streaming parsing algorithm for unrestricted regular expressions, and both methods follow a simple disambiguation policy consistent with that found in popular regex implementations such as Perl's. Our work paves the road for new tools with stronger guarantees and greater expressivity than current solutions, as well as new and interesting application areas. Furthermore, a connection is revealed between regular expression based parsing and finite-state transductions.

It would be a mistake to claim that our methods will replace all existing applications of regular expressions for search, extraction and manipulation of data. Existing tools are also appreciated for the features which we deliberately choose not to support, and there continue to be problem areas where the resulting trade-offs are acceptable. On the other hand, the two-pass and streaming regular expression parsing methods offer alternatives for those areas where the performance guarantee or increased expressivity is needed.

Our focus has mainly been on the theoretical aspects of regular expression parsing and little on practical applications, of which we believe there are many. Possible applications include parsing of data formats, streaming protocol implementation, advanced text editing and lexical analysis with *maximal munch* [83].

In order to enable any application, a considerable amount of effort has to be invested in tools and integration, including software libraries, command-line tools, programming language integration or the design of domain-specific languages. The development of a compiler for the latter is one of the topics of the next chapter, in which we develop a grammar-based programming language for high-performance streaming string processing, based on the streaming parsing algorithm presented in Paper B.

Chapter 3

Grammar Based Stream Processing

In this chapter we will consider two formalisms, *regular grammars* and *parsing expression grammars*, as foundations for specifications of string processing programs. It is our goal to be able to turn such specifications into efficient streaming programs which execute in time proportional to the length of the input string. "Streaming" in this context means that the resulting programs do not need access to the full input string at any time, but instead operate in a single pass from left to right, generating parts of the final result as they go along. Programs of this kind can be used to perform a range of useful tasks, including advanced text substitution, streaming filtering of log files, formatting of data to human-readable form, lexical analysis of programming languages, et cetera.

We first observe that the compact nature of REs cause large specifications to become unwieldy and hard to comprehend. A formalism that scales better is Chomsky's [21, 1] *context-free grammars* (CFG) for specifying linguistic structure using a set of production rules. CFGs have more expressive power than REs, so in order to use them as a replacement of the latter, a syntactic test must be used to discard those that use non-regular features. The regular CFGs have a natural notion of parse trees which is compatible with the transducer based view of RE parsing using greedy disambiguation.

In order to use regular CFGs as specifications of *programs*, we assign a semantics to the parse trees by translating every tree into a sequence of program statements to be executed. This type of specification is called a *syntax-directed translation scheme* (SDT) [59], and is obtained by allowing CFGs to contain program fragments, also called *semantic actions*, embedded within productions. The actions then show up in the parse trees which can be flattened to remove all structure except the sequence of program statements to be executed. This is the first formalism that will be considered by this chapter, and the goal is to apply the streaming RE parsing technique of Paper B.

The restriction to regular languages somewhat limit the possible applications, as it precludes the specification of programs that need to match parentheses or otherwise parse recursive language structures. For this purpose we want to base our program specifications on a more expressive formalism, but without giving up the strong performance guarantees provided by linear time parsing. A candidate for such as formalism is Ford's [36] *parsing expressing grammars* (PEG) for specifying recursive descent parses with limited backtracking. Every regular CFG parsed using greedy disambiguation corresponds to its interpretation as a PEG, but PEGs can additionally also express recursive parsing rules. This is the second formalism to be considered, and the challenge then becomes to generalize the streaming RE parsing methods to also apply to PEGs.

The rest of the chapter is structured as follows. We define context-free grammars in Section 3.1, and show how the regular subclass allows for a compact representation of non-deterministic finite automata. In Section 3.2 we describe syntax-directed translation schemes and give a few examples. In Section 3.3 we introduce parsing expression grammars as a generalization of regular translation schemes with greedy disambiguation. In Section 3.4, we discuss formalisms found in the literature for the specification of string processing programs and their evaluation on commodity hardware. We present our own contributions in Section 3.5, and offer our conclusions and perspectives for further work in Section 3.6.

3.1 Context-Free Grammars

REs are not a particularly compact way of specifying regular languages. Although REs and finite automata have the same expressive power, there are regular language whose smallest RE description is quadratically bigger than equivalent descriptions using DFAs [31, Theorem 23]. Furthermore, since the RE formalism does not include a systematic way of breaking up large REs into more manageable parts, they quickly become unwieldy and hard to comprehend for users. In this section we consider an alternative.

The *context-free grammars* (CFGs) introduced by Chomsky [21] is a formalism for systematically describing formal languages using a set of *production rules*. They are one step above REs in the *Chomsky hierarchy* of increasingly expressive generative language formalisms. Figure 3.1 shows an example of a CFG for a simple language. The underlined words in the grammar are called *nonterminal symbols* and act as "syntactic variables" in the specification. The letters (written in typewriter font) are called *terminal symbols*. Every line in the CFG is called a *production*, and is of the form $\underline{A} \rightarrow \alpha_0 \mid \alpha_1 \mid \ldots \mid \alpha_{n-1}$, where each α_i is a string (possibly empty) of terminals and nonterminals. The α_i strings are called *alternatives*, as they represent different choices for sentences described by the corresponding nonterminal. They are numbered from left to

$$\begin{aligned}
\underline{\textit{phrase}} &\rightarrow \underline{\textit{subject}}\ \underline{\textit{verb}}\ \underline{\textit{adjectives}} \\
\underline{\textit{subject}} &\rightarrow \texttt{he} \mid \texttt{she} \\
\underline{\textit{verb}} &\rightarrow \texttt{was} \mid \texttt{is} \\
\underline{\textit{adjectives}} &\rightarrow \underline{\textit{adverb}}\ \underline{\textit{adjective}} \mid \underline{\textit{adverb}}\ \underline{\textit{adjective}}\ \texttt{and}\ \underline{\textit{adjectives}} \\
\underline{\textit{adverb}} &\rightarrow \underline{\textit{verys}} \mid \texttt{not} \\
\underline{\textit{verys}} &\rightarrow \texttt{very}\ \underline{\textit{verys}} \mid \varepsilon \\
\underline{\textit{adjective}} &\rightarrow \texttt{happy} \mid \texttt{hungry} \mid \texttt{tall}
\end{aligned}$$

Figure 3.1: A context-free grammar.

right starting from zero. The nonterminal to the left of the arrow in the first production is a designated *starting symbol*.

The language described by the CFG contains the following sentences:

```
he is tall,
she was very hungry and not happy,
he is tall and very very happy,
she was happy and hungry and tall
```

This language can also be described by an RE. However, it is quite big, spanning two lines, and is not very readable:

$$\begin{aligned}
&(\texttt{he} + \texttt{she})(\texttt{was} + \texttt{is})((\texttt{very})^* + \texttt{not})(\texttt{happy} + \texttt{hungry} + \texttt{tall}) \\
&(\texttt{and}((\texttt{very})^* + \texttt{not})(\texttt{happy} + \texttt{hungry} + \texttt{tall}))^*
\end{aligned}$$

In particular, note that we have to include duplicate occurrences of most of the words in order to correctly specify that a list of adjectives is separated by the word and. On the other hand, the CFG is self-documenting by having the names of nonterminals describe what kind of sentence structure they define.

The language described by a CFG is determined as the set of all strings of terminal symbols that can be *derived* from the starting symbol. A derivation is a sequence of rewritings of strings containing terminal and nonterminal symbols. For any string α of the form $\beta\underline{A}\delta$, where β and δ are strings of terminals and nonterminals and $\underline{A}$ is a nonterminal with production $\underline{A} \rightarrow \gamma_0 \mid \gamma_1 \mid ... \mid \gamma_{n-1}$, we can rewrite α as follows:

$$\alpha \Rightarrow \beta\gamma_i\delta$$

where the alternative γ_i is chosen freely among the alternatives in the production for $\underline{A}$. If a string can be rewritten several times, $\alpha_0 \Rightarrow \alpha_1 \Rightarrow ... \Rightarrow \alpha'$, we also write just $\alpha \Rightarrow \alpha'$. Rewriting is a highly non-deterministic process, since neither the expanded non-terminal $\underline{A}$ or the chosen alternative are uniquely

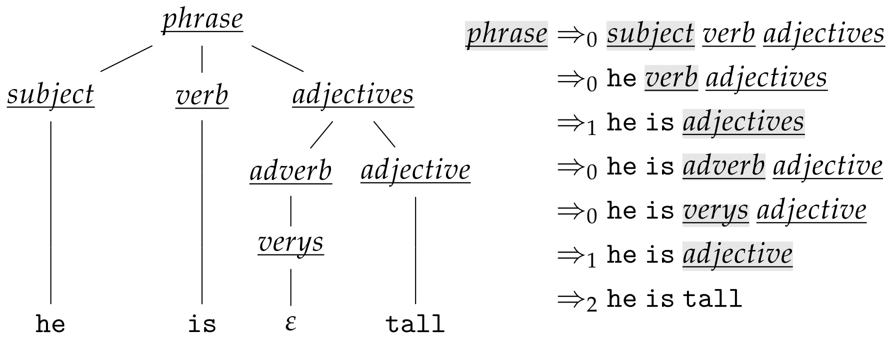


Figure 3.2: A parse tree and the corresponding derivation.

determined in each step. Figure 3.2 shows an example of how to derive a sentence in the grammar from Figure 3.1, starting from the starting symbol $\underline{\textit{phrase}}$, and with the expanded nonterminal highlighted in each step.

We denote the language of a grammar G with start symbol $\underline{S}$ by $\mathcal{L}[\![G]\!]$, and define it formally as the set of all terminal strings that can be derived from $\underline{S}$:

$$\mathcal{L}[\![G]\!] = \{u \in \Sigma^* \mid \text{there is a derivation } \underline{S} \Rightarrow u \text{ in } G\}$$

Parse Trees and Codes

Parse trees for CFGs are naturally defined as pictorial descriptions of how a given string is derived from the starting symbol. As such, a parse tree consists of labeled nodes, where internal nodes are labeled by nonterminals and leaf nodes are labeled by terminals or the empty string ε. The root is always labeled by the starting symbol, and for each internal node with label $\underline{A}$ and child nodes with labels $L_0, L_1, ..., L_{m-1}$, there must be a production $\underline{A} \to \alpha_0 \mid \alpha_1 \mid ... \mid \alpha_{n-1}$ such that $\alpha_i = L_0 L_1 ... L_{m-1}$ for some number i. If we assume that no production contains two equal alternatives, then every parse tree uniquely guides the choice of alternatives in derivations, although the order of expanded nonterminals is still nondeterministic. See Figure 3.2 for an example.

We can obtain a true one-to-one correspondence between parse trees and derivations by only considering derivations which choose the next nonterminal to expand in a particular order. For this purpose we only consider *leftmost* derivations, which always expand the leftmost nonterminal before others. The derivation in Figure 3.2 is leftmost, and thus uniquely determines the corresponding parse tree and vice versa.

Every parse tree can be given a serialized code in the same way as we did for RE parse trees in Chapter 2. Since a parse tree corresponds to a leftmost derivation which performs a series of deterministic expansions, the code can

$$\begin{aligned} \underline{S} &\to \mathtt{a}\underline{T} \mid \mathtt{a}\underline{L} \\ \underline{T} &\to \mathtt{b}\underline{L} \mid \mathtt{b}\underline{S} \\ \underline{L} &\to \mathtt{a}\underline{L} \mid \mathtt{b}\underline{L} \mid \varepsilon \end{aligned}$$

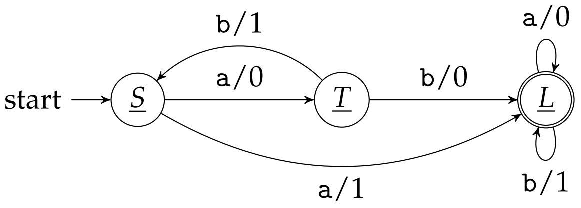


Figure 3.3: Regular CFG and its parsing transducer.

simply be defined as the sequence of numbers which determine the alternatives in each expansion step. For example, the derivation in Figure 3.2 has been annotated with the choice of alternative in each expansion, which leads to the code 0010012.

From CFGs to Transducers

The motivation for introducing CFGs were as a replacement for REs, allowing us to apply methods specific to RE parsing to parse CFGs. Not every CFG has a corresponding RE describing the same language, as the CFG formalism is significantly more expressive. For example, the simple grammar with only one production $\underline{S} \to \mathtt{a}\underline{S}\mathtt{b} \mid \varepsilon$ describes the non-regular language consisting of strings of as followed by exactly the same number of bs. We will have to rule out such grammars to ensure that we only consider the regular CFGs. There is no computer program which can determine for any CFG whether it is regular or not [12], but there are simple tests we can use which can verify most regular CFGs as such, but which returns false negatives for some [8].

It can be shown that every regular CFG can be rewritten such that for every production $\underline{A} \to \alpha_0 \mid \alpha_1 \mid \ldots \mid \alpha_{n-1}$, each α_i is either of the form $a_i\underline{B_i}$ where each a_i is a terminal symbol and $\underline{B_i}$ is a nonterminal symbol, or $\alpha_i = \varepsilon$. Grammars on this form are called *right-regular*, and have a natural interpretation as finite state transducers where each nonterminal identifies a state. If the production for a nonterminal $\underline{A}$ has an alternative $\alpha_i = \varepsilon$, then its state is accepting. For every alternative where $\alpha_i = a_i\underline{B_i}$, there is an outgoing transition $\underline{A} \overset{a_i/i}{\to} \underline{B_i}$. See Figure 3.3 for an example.

Disambiguation

The problem of ambiguity also arises for CFGs, since there can be more than one parse tree/leftmost derivation for a given string. The problem can be solved in a similar way as for REs by specifying a policy for selecting a single parse tree from a set of candidates. We consider here the greedy disambiguation policy introduced in the previous chapter generalized to CFGs: If there is more than one leftmost derivation for a terminal string, identify the first step at which they made different expansion choices, and pick the one that chose the earliest alternative.

For example, the grammar in Figure 3.3 is ambiguous since the string `aba` has three leftmost derivations:

$$\underline{S} \Rightarrow_0 \mathtt{a}\underline{T} \Rightarrow_0 \mathtt{ab}\underline{L} \Rightarrow_0 \mathtt{aba}\underline{L} \Rightarrow_1 \mathtt{aba} \tag{3.1}$$

$$\underline{S} \Rightarrow_0 \mathtt{a}\underline{T} \Rightarrow_1 \mathtt{ab}\underline{S} \Rightarrow_1 \mathtt{aba}\underline{L} \Rightarrow_1 \mathtt{aba} \tag{3.2}$$

$$\underline{S} \Rightarrow_1 \mathtt{a}\underline{L} \Rightarrow_1 \mathtt{ab}\underline{L} \Rightarrow_0 \mathtt{aba}\underline{L} \Rightarrow_0 \mathtt{aba} \tag{3.3}$$

We see that (3.1) differs from (3.2) and (3.3) in the first and second step, respectively. In both cases, (3.1) chooses an earlier alternative than the other, so this is the unique greedy derivation.

3.2 Syntax-Directed Translation Schemes

In a *syntax-directed translation scheme* (SDT) [1], we allow program fragments, also called *semantic actions*, to occur inside the alternatives of each production. The program fragments can be anything that can be executed on the underlying machine, such as manipulation of stateful variables or execution of side-effects. In order to distinguish semantic actions from the terminal and nonterminal symbols, we will write them in braces and highlight them. For example, the action which sets a variable `x` to value `"a"` is written `{x:="a"}`.

A derivation for an SDT is finished when it has reached a string which only contains interleaved terminal symbols and semantic actions. The parsed string is the substring of terminal symbols, and the substring of semantic actions forms a sequence of program statements to be executed.

We illustate SDTs by an example. Consider the SDT in Figure 3.4 which reads an English noun phrase, reformulates it, and prints the result. If we parse the string `a man who was happy` using the SDT, we obtain the (greedy) parse tree depicted in Figure 3.5. The subsequence of program statements in the leaves forms the following program, which when executed prints the string `a formerly happy man`:

```
d := "a";
n := "man";
v := "formerly";
a := "happy";
p := d+v+a+n;
print(p);
```

Many useful string processing programs can be conveniently specified using SDTs. For example, if we needed to collect statistics from a large log of web requests, an SDT could easily be used to parse the log entries and update a database based on the extracted data. We could also use an SDT to read data

```
phrase → det noun wp verb adj { p := d+v+a+n; print(p); }
   det → the { d := "the"; }
       | a { d := "a"; }
  noun → man { n := "man"; }
       | woman { n := "woman"; }
    wp → who
  verb → is { v := ""; }
       | was { v := "formerly"; }
   adj → happy { a := "happy"; }
       | tall { a := "tall"; }
```

Figure 3.4: Example SDT for reformulating simple English phrases.

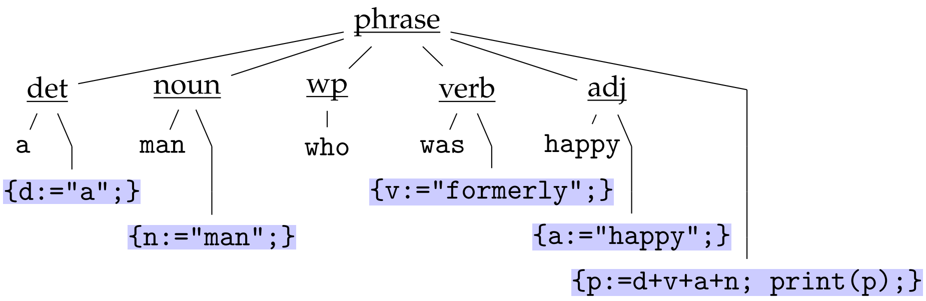


Figure 3.5: Greedy parse of the string `a man who was happy`, using SDT from Figure 3.4.

in a format which is hard to read for humans and automatically format it in a readable report.

For some applications such as implementations of protocols where data arrives in a stream whose total length is unknown, we want to start executing semantic actions as soon a possible, since we may not have enough memory to store the complete stream. Under this execution model, we have to be careful not to execute semantic actions "prematurely": if after seeing a prefix of the input stream we decide to execute an action, then that action must be guaranteed to be executed for every complete parse of the input.

This leads us to the first problem that we wish to address in this chapter:

> How to evaluate the SDT on an input string in a streaming fashion, using at most time proportional to the length of the input?

$$\begin{aligned}
\underline{sum} &\leftarrow \underline{factor}\ \texttt{+}\ \underline{sum} / \underline{factor} \\
\underline{factor} &\leftarrow \texttt{0} / \underline{digit}\ \underline{digits} / \texttt{(}\ \underline{sum}\ \texttt{)} \\
\underline{digits} &\leftarrow \underline{digit}\ \underline{digits} / \varepsilon \\
\underline{digit} &\leftarrow \texttt{0} / \texttt{1} / \ldots / \texttt{9}
\end{aligned}$$

Figure 3.6: Example of a simple PEG.

3.3 Parsing Expression Grammars

With the restriction to regular SDTs, we lose the ability to express a large number of interesting string processing programs. Regular languages cannot contain unbounded nesting, so this precludes processing languages such as arithmetic expressions, languages containing matching parentheses and nested data formats.

As we pointed out in the previous chapter, we cannot allow specifications based on unrestricted CFGs without losing the guarantee of linear time parsing [58], and we would like to avoid restricting ourselves to deterministic CFGs such as LR(k) [52] since they are difficult to write. It seems to be hard to come up with a suitable relaxation of the regular SDTs, so in this section we will step outside the Chomsky hierarchy and instead consider Ford's *parsing expression grammars* (PEG) [36].

A PEG is specified as a set of production rules, each of the form $\underline{A} \leftarrow e$, where $\underline{A}$ is a nonterminal as before, and e is a *parsing expression* (PE) generated by the following grammar:

$$e ::= \underline{A} \mid a \mid e_1 e_2 \mid e_1 / e_2 \mid !e_1$$

A PE can either be a nonterminal $\underline{A}$, a terminal symbol a in Σ, the empty string ε, a product $e_1 e_2$, an *ordered sum* e_1 / e_2, or a *negated expression* $!e_1$, where in all of the previous, e_1 and e_2 stand for PEs. The rules for associativity of parentheses are the same as for REs, and we write $e_1 e_2 e_3$ and $e_1 / e_2 / e_3$ for the PEs $e_1(e_2 e_3)$ and $e_1 / (e_2 / e_3)$, respectively. See Figure 3.6 for an example PEG which parses simple arithmetic expressions with parentheses.

PEG Semantics

Although on the surface PEGs resemble CFGs, their semantics are quite different. PEGs do not have a notion of derivations, but instead every parsing expression specifies a recursive backtracking parser which searches for a greedy parse of the input.

The result of a parse is either *success*, in which case zero or more input symbols are consumed, or *failure*, in which case exactly zero input symbols

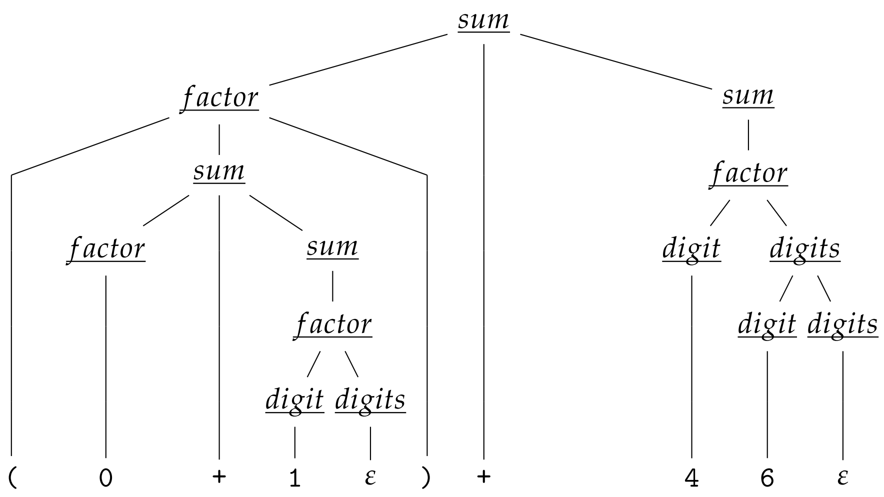


Figure 3.7: A PEG parse tree for the string `(0+1)+46`.

are consumed. If the PE being parsed is a terminal symbol, then the parse succeeds and consumes one symbol if the first symbol in the input matches; otherwise it fails. If the PE is a nonterminal, then parsing proceeds with the PE associated with that nonterminal in the PEG. For sequences $e_1 e_2$, the expression e_1 is parsed first, and if it succeeds, e_2 is parsed with the remainder of the input; otherwise $e_1 e_2$ fails. For ordered sums e_1 / e_2, the expression e_1 is parsed first, and if it succeeds, the whole sum succeeds, disregarding e_2. Only if e_1 fails is e_2 tried. A negation $!e_1$ fails if e_1 succeeds; if e_1 fails, then $!e_1$ succeeds, but consumes zero symbols.

The behavior for ordered sums means that in the PEG in Figure 3.6, parsing $\underline{factor}$ with input `0123` will fail: since the first alternative consumes `0`, the other alternatives are disregarded, leaving the suffix `123` unhandled. This illustrates the difference with CFGs, where the second alternative would have lead to a successful parse. The backtracking behavior is in this case intentionally used to reject numbers with leading zeros. See Figure 3.7 for the parse tree resulting from parsing the input `(0+1)+46`.

Although the semantics of PEGs are formulated as a backtracking parsing process, every PEG can be parsed in time proportional to the input length. One can either apply a dynamic programming approach [2, Theorem 6.4] or apply the memoizing Packrat algorithm due to Ford [35]. None of these algorithms operate in a streaming fashion, however.

Expressivity

PEGs are equivalent in power to the formalisms TDPL and GTDPL [36] due to Aho and Ullman [2], albeit a lot easier to read.

The expressive power of PEGs and CFGs is incomparable. The negation operator allows PEGs to parse languages which cannot be described by any CFG. An example of such a language is the following:

$$\underbrace{\mathtt{aa}\cdots\mathtt{a}}_{n \text{ times}}\underbrace{\mathtt{bb}\cdots\mathtt{b}}_{n \text{ times}}\underbrace{\mathtt{cc}\cdots\mathtt{c}}_{n \text{ times}}$$

That is, the strings consisting of as followed by bs followed by cs, in equal numbers. The PEG recognizing this language crucially depends on the negation operator in order to look ahead in the input string [36, Section 3.4]:

$$\begin{aligned}
\underline{D} &\leftarrow !!(\underline{A}\ !\mathtt{b})\ \underline{S}\ \underline{B}\ !(\mathtt{a}/\mathtt{b}/\mathtt{c})\\
\underline{A} &\leftarrow \mathtt{a}\ \underline{A}\ \mathtt{b}/\varepsilon\\
\underline{B} &\leftarrow \mathtt{b}\ \underline{B}\ \mathtt{c}/\varepsilon\\
\underline{S} &\leftarrow \mathtt{a}\ \underline{S}/\varepsilon
\end{aligned}$$

On the other hand, since every PEG can be parsed in linear time, then due to the non-linear lower bound of general CFG parsing [58], there must exist a CFG describing a language which cannot be parsed by any PEG.[1] However, every *deterministic* CFG can be simulated by PEG [2, Theorem 6.1], including all LL(k) and LR(k) grammars.

Although PEGs are incomparable to general CFGs, they *do* have a close connection to the right-regular CFGs. For every right-regular CFG, replace all productions of the form $\underline{A} \to \alpha_0 \mid \alpha_1 \mid \ldots \mid \alpha_{n-1}$ by PEG rules $\underline{A} \leftarrow \alpha_0/\alpha_1/\ldots/\alpha_{n-1}$, and then replace every occurrence of ε by a special end-of-input marker #. It can be shown that for every input string u, the greedy leftmost derivation for u in the original CFG will yield the same parse tree as the PEG on input u#. The reason for this is that no ordered sum in the PEG will finish parsing before all of the string has been processed, so all alternatives will be exhausted, resulting in a simulation of the search for the greedy leftmost derivation in the CFG. PEGs can thus be seen as direct generalizations of regular CFGs with greedy leftmost semantics.

It is straightforward to extend PEGs with semantic actions in the same way as we did for CFGs to obtain a generalization of the regular syntax-directed translation schemes. By applying one of the linear time PEG parsing algorithms, we can evaluate such a PEG-based SDT in a non-streaming fashion. This leads to the second problem to be addressed in this chapter:

> How to evaluate a PEG-based SDT on an input string in a streaming fashion, using at most time proportional to the length of the input?

[1] To the best of our knowledge, finding an example of such a language is an open problem.

3.4 String Processing Methods

We discuss formalisms for the specification of string processing programs and methods for evaluating such specifications on commodity hardware.

Line-Oriented Stream Processing

Several methods and tools for streaming text processing rely on a delimiters such as newline symbols to chunk the input stream. Each chunk is processed independently of the following ones, and can be discarded once the next chunk starts processing. The UNIX operating system adopted this model by treating text files as arrays of strings separated by newlines, and as a result all popular UNIX tools for streaming text processing, such as the regex based tools `sed` [92] and `awk/gawk` [78], operate using the chunking model. The advantage of this model is that each chunk can be assumed to be small, often a single line in a text file, which means that further pattern matching inside chunks do not have to be streaming. The disadvantage is, as noted by Pike [76], that *"[...] if the interesting quantum of information isn't a line, most of the tools [...] don't help"*, and as a consequence, processing data formats which are not line-oriented is complicated[2].

Automata Based Methods

There are several different methods for specification of streaming string processing programs using finite automata and their generalizations. The state machine compiler Ragel [95] allows users to specify NFAs whose transitions are annotated by arbitrary program statements from a host programming language. The annotated NFAs are converted to DFAs, and in the case of ambiguity the DFA will simultaneously perform actions from several NFA transitions upon transitioning from one state to the next, even if one of these transitions turns out not to be on a viable path. By contrast, a syntax-directed translation scheme will only perform the actions that occur in the unique final parse tree. For this reason, Ragel is most useful for processing mostly deterministic specifications or for pure recognition.

Methods based on the more expressive transducer model include the Microsoft Research languages Bek[3] [45] and Bex[4] [98], both of which are based on *symbolic transducers* [99, 27], a compact notation for representing transducers with many similar transitions which can be described using logical theories. Both languages are formalisms for expressing string sanitizers and encoders commonly found in web programming, supporting both synthesis of fast programs as well as automatic checking of common correctness criteria

[2] `sed`, `awk` are Turing-complete and can parse any decidable language, but not without pain.

[3] `http://rise4fun.com/Bek`

[4] `http://rise4fun.com/Bex`

of such specifications. Due to the focus on a limited application domain, both languages are restricted to expressing deterministic transducers only. This trivially ensures linear time execution, but also limits their expressivity.

Streaming string transducers (SSTs) [3, 4] is another extension of DFAs which upon transitioning from one state to another can perform a set of simultaneous copy-free updates to a finite number of string variables. SSTs are deterministic, but are powerful enough to be able express any function describable by non-deterministic transducers, as well as some functions which cannot, such as string reversal. Since they can be run in linear time, they are an interesting model of computation to target for string processing languages. DReX [5] is a domain specific string processing language based on a combinatory language [7] which can express all string functions describable by SSTs. In order for DReX programs to be evaluated in time proportional to the input length, they must be restricted to an unambiguous subset.

Domain-Specific Languages

There is an abundance of less general solutions which operate within restricted application domains. These include languages for specifying steaming processors for binary [11] and textual [33, 34] data formats, network packets [63, 61, 18] and wireless protocols [88]. Many of these require domain-specific features which are outside the scope of the general grammar based model of SDTs.

A system which comes close to the SDT model is PADS [33, 34], a domain-specific language for writing specifications of the physical and textual layouts of ad-hoc data formats from which parsers, statistical tools and streaming string translators to other textual formats or databases can be derived. PADS can be seen as regular SDTs with greedy disambiguation, but extended with extra features such as *data dependencies*—grammar alternatives can be resolved based on semantic predicates on previously parsed data. The parsers generated by a PADS specification operate via backtracking.

Parsing Expression Grammars

Streaming evaluation of PEG-based SDTs will have to rely on a streaming top-down parsing method for PEG. Current practical methods are either based on backtracking [65], recursive descent with memoization [35], or some variant of these using heuristics for optimization [82, 56].

There is only one known parsing method which is *streaming* [67], but it relies on the programmer to manually annotate the grammar with *cut points* to help the parsing algorithm figure out when parts of the parse tree can be written to the output.

For a more in-depth discussion on methods for streaming PEG parsing, we also refer to Section D.7 of Paper D.

3.5 Our Contributions

In Paper C we present *Kleenex*, a language for expressing high-performance streaming string processing programs as regular grammars with embedded semantic actions for string manipulation, and its compilation to efficient C code. Its underlying theory is based on transducer decomposition into oracle and action machines, where an oracle machine corresponds to a bit-coded RE parsing transducer of Chapter 2, and an action machine is a deterministic transducer which translates bit-codes into sequences of semantic actions to be executed. Based on the optimally streaming RE parsing algorithm of Paper B, the oracle machine, which is non-deterministic and ambiguous, is disambiguated using the greedy policy and converted into a deterministic streaming string transducer, the same machine model employed by DReX. Unlike DReX, we allow unrestricted ambiguity in Kleenex specifications which makes programming in Kleenex easier. By letting the set of semantic actions in Kleenex be copy-free string variable updates, it appears that Kleenex programs are equivalent to the full set of non-deterministic streaming string transducers [6], and thus equivalent in expressive power with DReX.

The generated transducers are translated to efficient C programs which achieve sustained high throughput in the 1Gbps range on practical use cases. The high performance is obtained by avoiding having to compute path trees at run-time—the most expensive part of the streaming algorithm of Paper B—by fully encoding the current path tree structure in the control mechanism of the streaming string transducer. Furthermore, having translated a Kleenex specification to a restricted machine model allows a range of optimizations to be applied, including standard compiler optimizations such as constant propagation [10] as well as model-specific optimizations such as symbolic representation [99].

In Paper D we present a new linear time parsing algorithm for parsing expression grammars. The algorithm is based on a well-known bottom-up tabulation strategy by Aho and Ullman [2] which is reformulated using least fixed points. Using the method of *chaotic iteration* [23] for computing least fixed points, we can compute approximations of the parse table, one for each prefix of the input, in an incremental top-down fashion. The approximated parse tables provide enough information for a simple dynamic analysis to predict a prefix of the control flow of all viable parses accepting a completion of the input prefix read so far. The result is a streaming parser which can be used to schedule semantic actions during the parsing process in the same fashion as Kleenex. We evaluate a prototype of the method on selected examples which shows that it automatically adapts to use practically constant space for grammars that do not require lookahead. We also point out directions for further improvements which must be addressed before the algorithm can be used as a basis for an efficient streaming implementation of parsing expression grammars. In particular, the algorithm fails to obtain streaming behavior for strictly right-regular grammars, and it also performs a large amount of superfluous computation, adding a large constant to the time complexity.

3.6 Conclusions and Perspectives

In this chapter, we have illustated how syntax-directed translation schemes provide a restricted but expressive formalism which programmers can use to specify streaming string processing programs without having to explicitly deal with orthogonal technical issues related to buffering and disambiguation.

With the Kleenex language, we have demonstrated that streaming regular expression parsing can be used to obtain high-performance implementations of regular syntax-directed translation schemes with greedy disambiguation. Kleenex provides a concise and convenient language for rapid development of streaming string processing programs with predictable high performance. These programs can be used to process many of the common ad-hoc data formats that can be described or approximated by regular grammars, including web request logs, CSV files, HTML documents, JSON files, and more. Kleenex is distinguished from other tools in the same category by allowing unrestricted ambiguity in specifications which are automatically disambiguated using a predictable policy, thus making it easier to combine and reuse Kleenex program fragments without having to worry about compiler errors.

For the cases where the expressivity of Kleenex is not adequate, we show that the foundation of regular grammars can be conservatively extended to the more expressive formalism of parsing expression grammars, thus allowing a larger range of translation schemes to be specified while preserving the input/output-semantics of the regular ones. This however leaves the question of how to evaluate parsing expression grammars in a streaming fashion. We address this issue by providing a streaming linear time algorithm which automatically adapts to constant memory usage in practical use cases, paving the way for a more expressive dialect of Kleenex.

There are several directions for future work on the Kleenex language and its compilation:

Data-parallel execution Veanes, Molnar and Mytkowics [100] show how to implement the symbolic tranducers of Bek and Bex on multi-core hardware in order to hide I/O latencies by processing separate partitions of the input string in parallel. By virtue of also being based on finite state transducers, a similar approach might be applicable to enable Kleenex to run on multi-core hardware as well.

Reducing state complexity Certain Kleenex specifications have a tendency to result in very large SSTs, which negatively affects both the compile times and the sizes of the produced binary programs. Perhaps we can apply a similar hybrid runtime simulation/compilation technique as used in the RE2 [24] library in order to materialize only the SST states reached during processing of a particular input stream.

We should also point out a result of Roche [84], who shows that the number of states in the forwards-backwards decomposition of a transducer can be exponentially smaller than the equivalent representation using a *bimachine* [87, 15], another deterministic transducer model. It is future work to see if this also applies to SSTs, and whether it can account for the blowups observed in practice, but if it turns out to be the case then a streaming variant of the forwards-backwards parsing algorithm of Paper A might serve as an alternative, more space economical execution model for Kleenex.

As we also point out in Paper D, there are still some issues that need to be addressed before the streaming parsing algorithm for parsing expression grammars can be used as a high-performance execution model in Kleenex:

Regular grammar parsing The algorithm fails to be streaming for the purely right-regular grammars, but works as expected for grammars using the non-regular features of parsing expression grammars. This is due to the fact that streaming regular expression parsing relies on orthogonal criteria for detecting when parts of the parse tree can be written to the output, which suggests that we might be able to find a hybrid method which can handle both types of grammars.

Time complexity overhead In its current form, the algorithm has been optimized for simplicity and performs a large number of computations which are never needed, adding a constant time overhead to the processing of each input symbol. This should be avoidable by integration with a runtime analysis, but requires further study.

Machine models Can we find a deterministic machine model which can simulate the streaming parsing algorithm such that parts of the expensive computations can be encoded in the control mechanism of the machine? Such a model would necessarily have to generalize the deterministic pushdown automata [1, 42] used for parsing deterministic context-free languages, but could potentially yield significant speedups.

Bibliography

[1] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Pearson Education, 2006.

[2] A. V. Aho and J. D. Ullman. *The Theory of Parsing, Translation, and Compiling*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1972.

[3] R. Alur and P. Černý. Expressiveness of streaming string transducers. In *Proc. Foundations of Software Technology and Teoretical Computer Science (FSTTCS)*, 2010.

[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, 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.

[6] R. Alur and J. Deshmukh. Nondeterministic streaming string transducers. *Automata, Languages and Programming*, 2011.

[7] 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.

[8] M. Anselmo, D. Giammarresi, and S. Varricchio. Finite automata and non-self-embedding grammars. In *Implementation and Application of Automata*, pages 47–56. Springer, 2003.

[9] V. Antimirov. Partial derivatives of regular expressions and finite automaton constructions. *Theor. Comput. Sci.*, 155(2):291–319, 1996.

[10] A. W. Appel. *Modern Compiler Implementation in ML*. Cambridge University Press, 1998.

[11] G. Back. DataScript- A Specification and Scripting Language for Binary Data. In D. Batory, C. Consel, and W. Taha, editors, *Generative Programming and Component Engineering*, number 2487 in Lecture Notes in Computer Science, pages 66–77. Springer Berlin Heidelberg, Oct. 2002. DOI: 10.1007/3-540-45821-2_4.

[12] Y. Bar-Hillel, M. Perles, and E. Shamir. On formal properties of simple phrase structure grammars. *Zeitschrift für Phonetik, Sprachwissenschaft und Kommunikationsforschung*, 14:143–172, Jan. 1961.

[13] M. Berglund, F. Drewes, and B. van der Merwe. Analyzing Catastrophic Backtracking Behavior in Practical Regular Expression Matching. *Electronic Proceedings in Theoretical Computer Science*, 151:109–123, May 2014. arXiv: 1405.5599.

[14] G. Berry and R. Sethi. From regular expressions to deterministic automata. *Theoretical Computer Science*, 48:117 – 126, 1986.

[15] J. Berstel. *Transductions and Context-Free Languages*. Teubner, 1979.

[16] A. Borsotti, L. Breveglieri, S. C. Reghizzi, and A. Morzenti. BSP: A parsing tool for ambiguous regular expressions. In *Implementation and Application of Automata*, pages 313–316. Springer, 2015.

[17] A. Borsotti, L. Breveglieri, S. C. Reghizzi, and A. Morzenti. From ambiguous regular expressions to deterministic parsing automata. In *Implementation and Application of Automata*, pages 35–48. Springer, 2015.

[18] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker. P4: Programming Protocol-independent Packet Processors. *SIGCOMM Comput. Commun. Rev.*, 44(3):87–95, July 2014.

[19] A. Brüggemann-Klein and D. Wood. One-unambiguous regular languages. *Information and computation*, 140(2):229–253, 1998.

[20] J. A. Brzozowski. Derivatives of regular expressions. *J. ACM*, 11(4):481–494, 1964.

[21] N. Chomsky. Three models for the description of language. *IRE Transactions on Information Theory*, 2(3):113–124, Sept. 1956.

[22] C. L. A. Clarke and G. V. Cormack. On the Use of Regular Expressions for Searching Text. *ACM Trans. Program. Lang. Syst.*, 19(3):413–426, May 1997.

[23] P. Cousot and R. Cousot. Automatic synthesis of optimal invariant assertions: Mathematical foundations. *SIGPLAN Notices*, 12(8):1–12, Aug 1977.

[24] R. Cox. Regular expression matching can be simple and fast (but is slow in Java, Perl, PHP, Python, Ruby, ...). `https://swtch.com/~rsc/regexp/regexp1.html`, January 2007.

[25] R. Cox. Regular expression matching: the virtual machine approach, December 2009.

[26] R. Cox. Regular expression matching in the wild, March 2010.

[27] L. D'Antoni and M. Veanes. Static Analysis of String Encoders and Decoders. In *VMCAI 2013*, volume 7737 of *LNCS*, pages 209–228. Springer Verlag, 2013.

[28] D. Dubé and M. Feeley. Efficiently Building a Parse Tree From a Regular Expression. *Acta Informatica*, 37(2):121–144, 2000.

[29] J. Earley. An Efficient Context-free Parsing Algorithm. *Commun. ACM*, 13(2):94–102, Feb. 1970.

[30] C. C. Elgot and J. E. Mezei. On Relations Defined by Generalized Finite Automata. *IBM J. Res. Dev.*, 9(1):47–68, Jan. 1965.

[31] 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.

[32] S. Fischer, F. Huch, and T. Wilke. A Play on Regular Expressions: Functional Pearl. In *Proc. of the 15th ACM SIGPLAN International Conference on Functional Programming*, ICFP '10, pages 357–368, New York, NY, USA, 2010. ACM.

[33] K. Fisher and R. Gruber. PADS: a domain-specific language for processing ad hoc data. *ACM Sigplan Notices*, 40(6):295–304, 2005.

[34] K. Fisher and D. Walker. The PADS project: an overview. In *Proceedings of the 14th International Conference on Database Theory*, pages 11–17. ACM, 2011.

[35] B. Ford. Packrat parsing: Simple, Powerful, Lazy, Linear Time. In *ACM SIGPLAN Notices*, volume 37, pages 36–47. ACM, Sept. 2002.

[36] B. Ford. Parsing Expression Grammars: A Recognition-Based Syntactic Foundation. *ACM SIGPLAN Notices*, 39(1):111–122, Jan. 2004.

[37] G. Fowler. An interpretation of the POSIX regex standard. `http://www2.research.att.com/~astopen/testregex/re-interpretation.html`, January 2003. Inaccessible as of September 2016. Copies are provided upon request to the author of this dissertation.

[38] J. Friedl. *Mastering Regular Expressions—Powerful Techniques for Perl and Other Tools*. O'Reilly, 1997.

[39] 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 (LNCS)*, pages 618–629. Springer, July 2004.

[40] V. M. Glushkov. On a synthesis algorithm for abstract automata. *Ukr. Matem. Zhurnal*, 12(2):147–156, 1960.

[41] C. Grabmayer. Using proofs by coinduction to find "traditional" proofs. In *Proc. 1st Conference on Algebra and Coalgebra in Computer Science (CALCO)*, number 3629 in Lecture Notes in Computer Science (LNCS). Springer, September 2005.

[42] D. Grune and C. J. Jacobs. *Parsing Techniques: A Practical Guide - Second Edition*. Monographs in Computer Science. Springer, 2008.

[43] S. Haber, W. Horne, P. Manadhata, M. Mowbray, and P. Rao. Efficient Submatch Extraction for Practical Regular Expressions. In A.-H. Dediu, C. Martín-Vide, and B. Truthe, editors, *Language and Automata Theory and Applications*, number 7810 in Lecture Notes in Computer Science, pages 323–334. Springer Berlin Heidelberg, Apr. 2013. DOI: 10.1007/978-3-642-37064-9_29.

[44] F. Henglein and L. Nielsen. Regular expression containment: Coinductive axiomatization and computational interpretation. In *Proc. 38th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL)*, volume 46 of *SIGPLAN Notices*, pages 385–398. ACM Press, January 2011.

[45] P. Hooimeijer, B. Livshits, D. Molnar, P. Saxena, and M. Veanes. Fast and Precise Sanitizer Analysis with BEK. In *Proceedings of the 20th USENIX Conference on Security*, SEC'11, pages 1–1, Berkeley, CA, USA, 2011. USENIX Association.

[46] P. Hudak, J. Hughes, S. Peyton Jones, and P. Wadler. A History of Haskell: Being Lazy with Class. In *Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages*, HOPL III, pages 12–1–12–55, New York, NY, USA, 2007. ACM.

[47] IEEE Computer Society. *Standard for Information Technology - Portable Operating System Interface (POSIX), Base Specifications, Issue 7*. IEEE, 2008. IEEE Std 1003.1.

[48] W. L. Johnson, J. H. Porter, S. I. Ackley, and D. T. Ross. Automatic Generation of Efficient Lexical Processors Using Finite State Techniques. *Commun. ACM*, 11(12):805–813, Dec. 1968.

[49] S. Kearns. Extending regular expressions with context operators and parse extraction. *Software - Practice and Experience*, 21(8):787–804, 1991.

[50] J. Kirrage, A. Rathnayake, and H. Thielecke. Static Analysis for Regular Expression Denial-of-Service Attacks. In J. Lopez, X. Huang, and R. Sandhu, editors, *Network and System Security*, number 7873 in Lecture Notes in Computer Science, pages 135–148. Springer Berlin Heidelberg, June 2013. DOI: 10.1007/978-3-642-38631-2_11.

[51] S. C. Kleene. Representation of Events in Nerve Nets and Finite Automata. In C. E. Shannon and J. McCarthy, editors, *Automata Studies*, pages 3–41. Princeton University Press, 1956.

[52] D. E. Knuth. On the translation of languages from left to right. *Information and Control*, 8(6):607–639, Dec. 1965.

[53] D. Kozen. A completeness theorem for Kleene algebras and the algebra of regular events. *Infor. and Comput.*, 110(2):366–390, 1994.

[54] D. Kozen. *Automata and computability*. Springer Verlag, 1997.

[55] C. Kuklewicz. Regex Posix - HaskellWiki. `https://wiki.haskell.org/Regex_Posix`. Accessed September 2016.

[56] K. Kuramitsu. Packrat Parsing with Elastic Sliding Window. *Journal of Information Processing*, 23(4):505–512, 2015.

[57] V. Laurikari. NFAs with tagged transitions, their conversion to deterministic automata and application to regular expressions. In *Seventh International Symposium on String Processing and Information Retrieval, 2000. SPIRE 2000. Proceedings*, pages 181–187, 2000.

[58] L. Lee. Fast Context-free Grammar Parsing Requires Fast Boolean Matrix Multiplication. *J. ACM*, 49(1):1–15, Jan. 2002.

[59] P. M. Lewis, D. J. Rosenkrantz, and R. E. Stearns. Attributed Translations. In *Proceedings of the Fifth Annual ACM Symposium on Theory of Computing*, STOC '73, pages 160–171, New York, NY, USA, 1973. ACM.

[60] P. M. Lewis, II and R. E. Stearns. Syntax-Directed Transduction. *J. ACM*, 15(3):465–488, July 1968.

[61] A. Madhavapeddy, A. Ho, T. Deegan, D. Scott, and R. Sohan. Melange: Creating a "Functional" Internet. In *Proceedings of the 2Nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, EuroSys '07, pages 101–114, New York, NY, USA, 2007. ACM.

[62] K. Mahalingam and O. Bagasra. Bioinformatics Tools: Searching for Markers in DNA/RNA Sequences. In *BIOCOMP*, pages 612–615, 2008.

[63] P. J. McCann and S. Chandra. Packet Types: Abstract Specification of Network Protocol Messages. In *Proceedings of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '00, pages 321–333, New York, NY, USA, 2000. ACM.

[64] R. McNaughton and H. Yamada. Regular expressions and state graphs for automata. *IRE Trans. on Electronic Comput.*, EC-9(1):38–47, 1960.

[65] S. Medeiros and R. Ierusalimschy. A Parsing Machine for PEGs. In *Proceedings of the 2008 Symposium on Dynamic Languages*, DLS '08, pages 2:1–2:12, New York, NY, USA, 2008. ACM.

[66] S. Medeiros, F. Mascarenhas, and R. Ierusalimschy. From regexes to parsing expression grammars. *Science of Computer Programming*, 93, Part A:3–18, Nov. 2014.

[67] K. Mizushima, A. Maeda, and Y. Yamaguchi. Packrat Parsers Can Handle Practical Grammars in Mostly Constant Space. In *Proceedings of the 9th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, PASTE '10, pages 29–36, New York, NY, USA, 2010. ACM.

[68] M. Mohri. Finite-state transducers in language and speech processing. *Computational linguistics*, 23(2):269–311, 1997.

[69] E. Myers, P. Oliva, and K. Guimarães. Reporting exact and approximate regular expression matches. In *Combinatorial Pattern Matching*, pages 91–103. Springer, 1998.

[70] L. Nielsen and F. Henglein. Bit-coded Regular Expression Parsing. In *Proc. 5th Int'l Conf. on Language and Automata Theory and Applications (LATA)*, volume 6638 of *Lecture Notes in Computer Science (LNCS)*, pages 402–413. Springer, May 2011.

[71] S. Okui and T. Suzuki. Disambiguation in regular expression matching via position automata with augmented transitions. In M. Domaratzki and K. Salomaa, editors, *Implementation and Application of Automata*, volume 6482 of *Lecture Notes in Computer Science*, pages 231–240. Springer Berlin Heidelberg, 2011.

[72] 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.

[73] T. J. Ostrand, M. C. Paull, and E. J. Weyuker. Parsing regular grammars with finite lookahead. *Acta Informatica*, 16(2):125–138, 1981.

[74] J. Ousterhout. Tcl: An Embeddable Command Language. In *Proc. USENIX Winter Conference*, pages 133–146, January 1990.

[75] S. Owens, J. Reppy, and A. Turon. Regular-expression derivatives reexamined. *Journal of Functional Programming*, 19(2):173–190, Mar. 2009.

[76] R. Pike. Structural regular expressions. In *Proceedings of the EUUG Spring 1987 Conference*, pages 21–28, Helsinki, May 1987.

[77] R. Pike. The text editor sam. *Software: Practice and Experience*, 17(11):813–845, Nov. 1987.

[78] T. G. project. `https://www.gnu.org/software/gawk/`, 2016.

[79] T. O. W. A. S. Project. Regular expression Denial of Service - ReDoS, 2015.

[80] M. O. Rabin and D. Scott. Finite Automata and Their Decision Problems. *IBM Journal of Research and Development*, 3:114–125, 1959.

[81] A. Rathnayake and H. Thielecke. Static analysis for regular expression exponential runtime via substructural logics. *CoRR*, abs/1405.7058, 2014.

[82] R. R. Redziejowski. Mouse: From parsing expressions to a practical parser. In *Concurrency Specification and Programming Workshop*. Citeseer, 2009.

[83] T. Reps. "maximal-munch" tokenization in linear time. *ACM Trans. Program. Lang. Syst.*, 20(2):259–273, 1998.

[84] E. Roche. Factorization of finite-state transducers. *Mitsubishi Electric Research Laboratories*, pages 1–13, 1995.

[85] A. Salomaa. Two complete axiom systems for the algebra of regular events. *J. ACM*, 13(1):158–169, 1966.

[86] N. Schwarz. *Scaleable Code Clone Detection*. PhD thesis, PhD thesis, University of Bern. 569, 2014.

[87] M. P. Schützenberger. A remark on finite transducers. *Information and Control*, 4(2-3):185–196, Sept. 1961.

[88] G. Stewart, M. Gowda, G. Mainland, B. Radunovic, D. Vytiniotis, and C. L. Agulló. Ziria: A DSL for wireless systems programming. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 415–428. ACM, 2015.

[89] S. Sugiyama and Y. Minamide. Checking time linearity of regular expression matching based on backtracking. In *IPSJ Transactions on Programming*, number 3 in 7, pages 1–11, 2014.

[90] M. Sulzmann and K. Z. M. Lu. Regular expression sub-matching using partial derivatives. In *Proc. 14th symposium on Principles and practice of declarative programming*, PPDP '12, pages 79–90, New York, NY, USA, 2012. ACM.

[91] 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.

[92] The GNU project. GNU SED. `http://www.gnu.org/software/sed/`, 2015.

[93] The RE2 authors. RE2. `https://github.com/google/re2`, 2015.

[94] K. Thompson. Programming techniques: Regular expression search algorithm. *Commun. ACM*, 11(6):419–422, 1968.

[95] A. Thurston. Ragel state machine compiler. `https://www.colm.net/files/ragel/ragel-6.7.pdf`, 2003.

[96] M. Tomita. An Efficient Augmented-context-free Parsing Algorithm. *Comput. Linguist.*, 13(1-2):31–46, Jan. 1987.

[97] S. Vansummeren. Type inference for unique pattern matching. *ACM Trans. Program. Lang. Syst.*, 28(3):389–428, 2006.

[98] M. Veanes. Symbolic String Transformations with Regular Lookahead and Rollback. In *Ershov Informatics Conference (PSI'14)*. Springer Verlag, 2014.

[99] 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.

[100] 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.

[101] L. Wall et al. The Perl programming language. `http://www.perl.org`, 2015.

[102] N. Weideman, B. v. d. Merwe, M. Berglund, and B. Watson. Analyzing Matching Time Behavior of Backtracking Regular Expression Matchers by Using Ambiguity of NFA. In Y.-S. Han and K. Salomaa, editors, *Implementation and Application of Automata*, number 9705 in Lecture Notes in Computer Science, pages 322–334. Springer International Publishing, July 2016. DOI: 10.1007/978-3-319-40946-7_27.

[103] D. H. Younger. Recognition and parsing of context-free languages in time n^3. *Information and Control*, 10(2):189–208, Feb. 1967.

Paper A

Two-Pass Greedy Regular Expression Parsing

This paper has been published in the following:

> Niels Bjørn Bugge Grathwohl, Fritz Henglein, Lasse Nielsen and Ulrik Terp Rasmussen. "Two-Pass Greedy Regular Expression Parsing". In *Proceedings 18th International Conference on Implementation and Application of Automata (CIAA)*, pages 60-71. Springer, 2013. DOI: 10.1007/978-3-642-39274-0_7.

The enclosed version contains minor revisions of the published paper in the form of corrections of typos and reformatting to fit the layout of this dissertation. The presentation in Sections A.4 and A.5 has also been improved, but no contributions have been added or removed.

Two-Pass Greedy Regular Expression Parsing[1]

Niels Bjørn Bugge Grathwohl, Fritz Henglein, Lasse Nielsen and Ulrik Terp Rasmussen

Department of Computer Science, University of Copenhagen (DIKU)

Abstract

We present new algorithms for producing greedy parses for regular expressions (REs) in a semi-streaming fashion. Our lean-log algorithm executes in time $O(mn)$ for REs of size m and input strings of size n and outputs a compact bit-coded parse tree representation. It improves on previous algorithms by: operating in only 2 passes; using only $O(m)$ words of random-access memory (independent of n); requiring only kn bits of sequentially written and read log storage, where $k < \frac{1}{3}m$ is the number of alternatives and Kleene stars in the RE; processing the input string as a symbol stream and not requiring it to be stored at all. Previous RE parsing algorithms do not scale linearly with input size, or require substantially more log storage and employ 3 passes where the first consists of reversing the input, or do not or are not known to produce a greedy parse. The performance of our unoptimized C-based prototype indicates that the superior performance of our lean-log algorithm can also be observed in practice; it is also surprisingly competitive with RE tools not performing full parsing, such as Grep.

A.1 Introduction

Regular expression (RE) parsing is the problem of producing a parse tree for an input string under a given RE. In contrast to most regular-expression based tools for programming such as Grep, RE2 and Perl, RE parsing returns not only whether the input is accepted, where a substring matching the RE and/or sub-REs are matched, but a full parse tree. In particular, for Kleene stars it returns *a list* of all matches, where each match again can contain such lists depending on the star depth of the RE.

An RE parser can be built using Perl-style backtracking or general context-free parsing techniques. What the backtracking parser produces is the *greedy* parse amongst potentially many parses. General context-free parsing and backtracking parsing are not scalable since they have cubic, respectively exponential worst-case running times. REs can be and often are grammatically ambiguous and can require arbitrary much look-ahead, making limited look-ahead context-free parsing techniques inapplicable. Kearns [7] describes the first linear-time algorithm for RE parsing. In a streaming context it consists

[1] The order of authors is insignificant.

of 3 passes: reverse the input, perform backward NFA-simulation, and construct parse tree. Frisch and Cardelli [5] formalize greedy parsing and use the same strategy to produce a greedy parse. Dubé and Feeley [2] and Nielsen and Henglein [9] produce parse trees in linear time for fixed RE, the former producing internal data structures and their serialized forms, the latter parse trees in bit-coded form; neither produces a greedy parse.

In this paper we make the following contributions:

1. Specification and construction of symmetric nondeterministic finite automata (NFA) with maximum in- and out-degree 2, whose paths from initial to final state are in one-to-one correspondence with the parse trees of the underlying RE; in particular, the greedy parse for a string corresponds to the lexicographically least path accepting the string.

2. NFA simulation with *ordered state sets*, which gives rise to a 2-pass greedy parse algorithm using $\lceil m \lg m \rceil$ bits per input symbol in the original input string, with m the size of the underlying RE. No input reversal is required.

3. NFA simulation optimized to require only $k \leq \lceil 1/3m \rceil$ bits per input symbol, where the input string need not be stored at all and the 2nd pass is simplified. Remarkably, this *lean-log algorithm* requires fewest log bits, and neither state set nor even the input string need to be stored.

4. An empirical evaluation, which indicates that our prototype implementation of the optimized 2-pass algorithm outperforms also in practice previous RE parsing tools and is sometimes even competitive with RE tools performing limited forms of RE matching.

In the remainder, we introduce REs as types to represent parse trees, define greedy parses and their bit-coding, introduce NFAs with bit-labeled transitions, describe NFA simulation with ordered sets for greedy parsing and finally the optimized algorithm, which only logs join state bits. We conclude with an empirical evaluation of a straightforward prototype to gauge the competitiveness of full greedy parsing with regular-expression based tools yielding less information for Kleene-stars.

A.2 Symmetric NFA Representation of Parse Trees

REs are finite terms of the form $0, 1, a, E_1 \times E_2, E_1 + E_2$ or E_1^*, where E_1, E_2 are REs.

Proviso: For simplicity and brevity we henceforth assume REs that do not contain sub-REs of the form E^*, where E is nullable (can generate the empty string). All results reported here can be and have been extended to such problematic REs in the style of Frisch and Cardelli [5]. In particular, our implementation BitC handles problematic REs.

REs can be interpreted as types built from singleton, product, sum, and list type constructors [5, 6]:

$$\begin{aligned}
\mathcal{T}[\![0]\!] &= \emptyset \\
\mathcal{T}[\![1]\!] &= \{()\}, \\
\mathcal{T}[\![a]\!] &= \{\mathtt{a}\}, \\
\mathcal{T}[\![E_1 \times E_2]\!] &= \{(V_1, V_2) \mid V_1 \in \mathcal{T}[\![E_1]\!], V_2 \in \mathcal{T}[\![E_2]\!]\}, \\
\mathcal{T}[\![E_1 + E_2]\!] &= \{\mathsf{inl}\ V_1 \mid V_1 \in \mathcal{T}[\![E_1]\!]\} \cup \{\mathsf{inr}\ V_2 \mid V_2 \in \mathcal{T}[\![E_2]\!]\}, \\
\mathcal{T}[\![E_0^\star]\!] &= \{[V_1, \ldots, V_n] \mid n \geq 0 \wedge \forall 1 \leq i \leq n. V_i \in \mathcal{T}[\![E_0]\!]\}
\end{aligned}$$

Its structured values $\mathcal{T}[\![E]\!]$ represent the *parse trees* for E such that the regular language $\mathcal{L}[\![E]\!]$ coincides with the strings obtained by flattening the parse trees:

$$\mathcal{L}[\![E]\!] = \{\mathsf{flat}(V) \mid V \in \mathcal{T}[\![E]\!]\},$$

where the flattening function erases all structure but the leaves:

$$\begin{aligned}
\mathsf{flat}(()) &= \epsilon \\
\mathsf{flat}(a) &= \mathtt{a} \\
\mathsf{flat}((V_1, V_2)) &= \mathsf{flat}(V_1)\mathsf{flat}(V_2) \\
\mathsf{flat}(\mathsf{inl}\ V_1) &= \mathsf{flat}(V_1) \\
\mathsf{flat}(\mathsf{inr}\ V_2) &= \mathsf{flat}(V_2) \\
\mathsf{flat}([V_1, \ldots, V_n]) &= \mathsf{flat}(V_1) \ldots \mathsf{flat}(V_n)
\end{aligned}$$

We recall bit-coding from Nielsen and Henglein [9]. The bit code $\mathsf{code}(V)$ of a parse tree $V \in \mathcal{T}[\![E]\!]$ is a sequence of bits uniquely identifying V within $\mathcal{T}[\![E]\!]$; that is, there exists a function decode_E such that for all $V \in \mathcal{T}[\![E]\!]$, we have $\mathsf{decode}_E(\mathsf{code}(V)) = V$:

$$\begin{aligned}
\mathsf{code}(()) &= \epsilon \\
\mathsf{code}(a) &= \epsilon \\
\mathsf{code}((V_1, V_2)) &= \mathsf{code}(V_1)\,\mathsf{code}(V_2) \\
\mathsf{code}([V_1, \ldots, V_n]) &= \mathsf{0}\,\mathsf{code}(V_1) \ldots \mathsf{0}\,\mathsf{code}(V_n)\,\mathsf{1} \\
\mathsf{code}(\mathsf{inl}\ V_1) &= \mathsf{0}\ \mathsf{code}(V_1) \\
\mathsf{code}(\mathsf{inr}\ V_2) &= \mathsf{1}\ \mathsf{code}(V_2)
\end{aligned}$$

The definition of decode_E is omitted for brevity, but is straightforward.

We write $\mathcal{B}[\![\ldots]\!]$ instead of $\mathcal{T}[\![\ldots]\!]$ whenever we want to refer to the bit codings, rather than the parse trees. We use subscripts to discriminate parses with a specific flattening: $\mathcal{T}_s[\![E]\!] = \{V \in \mathcal{T}[\![E]\!] \mid \mathsf{flat}(V) = s\}$. We extend the notation $\mathcal{B}_s[\![\ldots]\!]$ similarly.

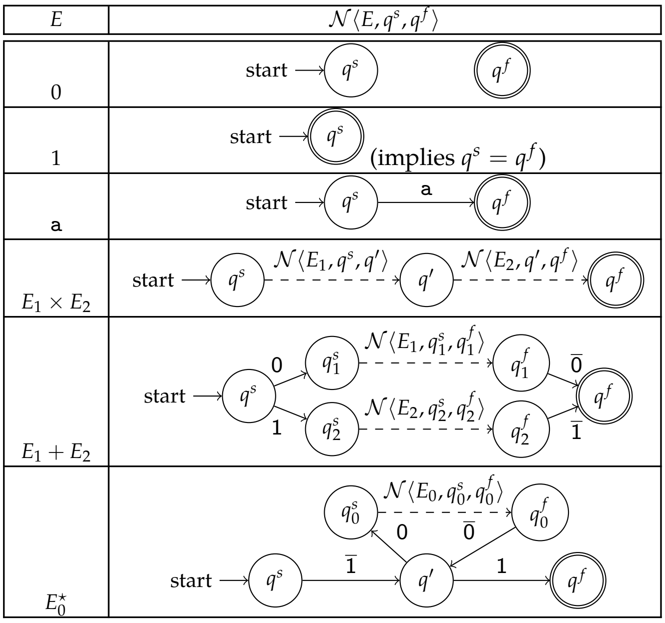


Figure A.1: aNFA construction schema.

Note that a bit string by itself does not carry enough information to deduce which parse tree it represents. Indeed this is what makes bit strings a compact representation of strings where the underlying RE is statically known.

The set $\mathcal{B}[\![E]\!]$ for an RE E can be compactly represented by an *augmented nondeterministic finite automaton (aNFA)*, a variant of enhanced NFAs [9] that has in- and outdegree at most 2 and carries a label on each transition.

Definition 1 (Augmented NFA)**.** An *augmented NFA* (aNFA) is a 5-tuple $M = (Q, \Sigma, \Delta, q^s, q^f)$ where Q is the set of states, Σ is the input alphabet, and q^s, q^f are the start and final states, respectively. The transition relation $\Delta \subseteq Q \times (\Sigma \cup \{0, 1, \overline{0}, \overline{1}\}) \times Q$ contains directed, labeled transitions: $(q, \gamma, q') \in \Delta$ is a transition from q to q' with label γ, written $q \xrightarrow{\gamma} q'$.

We call transition labels in Σ *input labels*; labels in $\{0, 1\}$ *output labels*; and labels in $\{\overline{0}, \overline{1}\}$ *log labels*.

We write $q \overset{p}{\rightsquigarrow} q'$ if there is a path labeled p from q to q'. The sequences $\mathsf{read}(p)$, $\mathsf{write}(p)$, and $\mathsf{log}(p)$ are the subsequences of input labels, output labels, and log labels of p, respectively.

We write: J_M for the *join states* $\{q \in Q \mid \exists q_1, q_2.\ (q_1, \overline{0}, q), (q_2, \overline{1}, q) \in \Delta\}$; S_M for the *symbol sources* $\{q \in Q \mid \exists q' \in Q, a \in \Sigma.\ (q, a, q')\}$; and C_M for the *choice states* $\{q \in Q \mid \exists q_1, q_2.\ (q, 0, q_1), (q, 1, q_2) \in \Delta\}$.

If M is an aNFA, then $\overline{M}$ is the aNFA obtained by *flipping* all transitions and exchanging the start and finishing states, that is reverse all transitions and interchange output labels with the corresponding log labels. □

Our algorithm for constructing an aNFA from an RE is a standard Thompson-style NFA generation algorithm modified to accomodate output and log labels:

Definition 2 (aNFA construction)**.** We write $M = \mathcal{N}\langle E, q^s, q^f\rangle$ when M is an aNFA constructed according to the rules in Figure A.1.

Augmented NFAs are dual under reversal; that is, flipping produces the augmented NFA for the reverse of the regular language.

Proposition A.2.1. *Let $\overline{E}$ be canonically constructed from E to denote the reverse of $\mathcal{L}[\![E]\!]$, i.e. $\overline{E_1 \times E_2} = \overline{E_2} \times \overline{E_1}$. Let $M = \mathcal{N}\langle E, q^s, q^f\rangle$. Then $\overline{M} = \mathcal{N}\langle \overline{E}, q^f, q^s\rangle$.*

This is useful since we will be running aNFAs in both forward and backward (reverse) directions.

Well-formed aNFAs—and Thompson-style NFAs in general—are canonical representations of REs in the sense that they not only represent their language interpretation, but their type interpretation:

Theorem A.2.2 (Representation)**.** *Given an aNFA $M = \mathcal{N}\langle E, q^s, q^f\rangle$, M outputs the bit-codings of E:*

$$\mathcal{B}_s[\![E]\!] = \{\mathsf{write}(p) \mid q^s \overset{p}{\rightsquigarrow} q^f \wedge \mathsf{read}(p) = s\}.$$

A.3 Greedy parsing

The *greedy parse* of a string s under an RE E is what a backtracking parser returns that tries the left operand of an alternative first and backtracks to try the right alternative only if the left alternative does not yield a successful parse. The name comes from treating the Kleene star $E^\star$ as $E \times E^\star + 1$, which "greedily" matches E against the input as many times as possible. A "lazy" matching interpretation of $E^\star$ corresponds to treating $E^\star$ as $1 + E \times E^\star$. (In practice, multiple Kleene-star operators are allowed to make both interpretations available; e.g. $E*$ and $E**$ in PCRE.)

Greedy parsing can be formalized by an order $\lessdot$ on parse trees, where $V_1 \lessdot V_2$ means that V_1 is "more greedy" than V_2. The following is adapted from Frisch and Cardelli [5].

Definition 3 (Greedy order)**.** The binary relation $\lessdot$ is defined inductively on the structure of values as follows:

$$\begin{array}{rcll}
(V_1, V_2) & \lessdot & (V_1', V_2') & \text{if} \quad V_1 \lessdot V_1' \vee (V_1 = V_1' \wedge V_2 \lessdot V_2') \\
\mathsf{inl}\ V_0 & \lessdot & \mathsf{inl}\ V_0' & \text{if} \quad V_0 \lessdot V_0' \\
\mathsf{inr}\ V_0 & \lessdot & \mathsf{inr}\ V_0' & \text{if} \quad V_0 \lessdot V_0' \\
\mathsf{inl}\ V_0 & \lessdot & \mathsf{inr}\ V_0' & \\
[V_1, \ldots] & \lessdot & [] & \\
[V_1, \ldots] & \lessdot & [V_1', \ldots] & \text{if} \quad V_1 \lessdot V_1' \\
[V_1, V_2, \ldots] & \lessdot & [V_1, V_2', \ldots] & \text{if} \quad [V_2, \ldots] \lessdot [V_2', \ldots]
\end{array}$$

The relation $\lessdot$ is not a total order; consider for example the incomparable elements $(\mathsf{a}, \mathsf{inl}\ ())$ and $(\mathsf{b}, \mathsf{inr}\ ())$. The parse trees of any particular RE are totally ordered, however:

Proposition A.3.1. *For each E, the order $\lessdot$ is a strict total order on $\mathcal{T}[\![E]\!]$.*

In the following, we will show that there is a correspondence between the structural order on values and the lexicographic order on their bit-codings.

Definition 4. For bit sequences $d, d' \in \{0, 1\}^\star$ we write $d \prec d'$ if d is lexicographically strictly less than d'; that is, $\prec$ is the least relation satisfying

1. $\epsilon \prec d$ if $d \neq \epsilon$
2. $b\ d \prec b'\ d'$ if $b < b'$ or $b = b'$ and $d \prec d'$.

Theorem A.3.2. *For all REs E and values $V, V' \in \mathcal{T}[\![E]\!]$ we have $V \lessdot V'$ iff* $\mathsf{code}(V) \prec \mathsf{code}(V')$.

Corollary A.3.3. *For any RE E with aNFA $M = \mathcal{N}\langle E, q^s, q^f\rangle$, and for any string s, $\min_{\lessdot} \mathcal{T}_s[\![E]\!]$ exists and*

$$\min_{\lessdot} \mathcal{T}_s[\![E]\!] = \mathsf{decode}_E \left(\min_{\prec} \left\{ \mathsf{write}(p) \,\middle|\, q^s \overset{p}{\rightsquigarrow} q^f \wedge \mathsf{read}(p) = s \right\} \right).$$

Proof. Follows from Theorems A.2.2 and A.3.2. □

We can now characterize greedy RE parsing as follows: Given an RE E and string s, find bit sequence b such that there exists a path $q^s \overset{p}{\rightsquigarrow} q^f$ from start to finishing state in the aNFA for E such that:

1. $\mathsf{read}(p) = s$,
2. $\mathsf{write}(p) = b$,
3. b is lexicographically least among all paths satisfying 1 and 2.

This is easily done by a backtracking algorithm that tries 0-labeled transitions before 1-labeled ones. It is atrociously slow in the worst case, however: exponential time. How to do it faster?

A.4 NFA-Simulation with Ordered State Sets

Our first algorithm is basically an NFA-simulation. For reasons of space we only sketch its key idea, which is the basis for the more efficient algorithm in the following section.

A standard NFA-simulation consists of computing $\mathsf{Reach}(S,s)$ where

$$\begin{array}{rcl}
\mathsf{Reach}(S,\epsilon) & = & S \\
\mathsf{Reach}(S,a\,s') & = & \mathsf{Reach}(\mathsf{Close}(\mathsf{Step}(S,a)),s') \\
\mathsf{Step}(S,a) & = & \{q' \mid q \in S, q \xrightarrow{a} q'\} \\
\mathsf{Close}(S') & = & \{q'' \mid q' \in S', q' \overset{p}{\rightsquigarrow} q'', \mathsf{read}(p) = \epsilon\}
\end{array}$$

Checking $q^f \in \mathsf{Reach}(S_0,s)$ where $S_0 = \mathsf{Close}(\{q^s\})$ determines whether s is accepted or not. But how to construct an accepting *path* and in particular the one corresponding to the greedy parse?

We can *log* the sequence of NFA state sets reached during forward NFA-simulation over an input string $s = a_1 \ldots a_n$. The log thus consists of a list of state sets $S_0, S_1, ..., S_n$, where S_0 is defined above, and for each $0 \leq i \leq n-1$, we have $S_{i+1} = \mathsf{Close}(\mathsf{Step}(S_i, a_{i+1}))$.

It is easy to check that every path $q^s \overset{p}{\rightsquigarrow} q^f$ with $\mathsf{read}(p) = s$ is of the form

$$q^s \overset{p_0}{\rightsquigarrow} q_0 \xrightarrow{a_1} q_0' \overset{p_1}{\rightsquigarrow} q_1 \cdots q_i \xrightarrow{a_{i+1}} q_i' \overset{p_{i+1}}{\rightsquigarrow} q_{i+1} \cdots q_{n-1} \xrightarrow{a_n} q_{n-1}' \overset{p_n}{\rightsquigarrow} q^f$$

where $p = p_0 a_1 p_1 a_2 p_2 ... a_n p_n$ and $\mathsf{read}(p_i) = \epsilon$ for all $0 \leq i \leq n$. By definition, each q_i is in the state set S_i, so the set of all paths $\{p \mid q^s \overset{p}{\rightsquigarrow} q^f \wedge \mathsf{read}(p) = s\}$ can be recovered only from the log, and equals the set $\mathsf{Paths}(n, q^f)$ defined as follows:

$$\begin{array}{rl}
\mathsf{Paths}(0,q'') = & \{p \mid q^s \overset{p}{\rightsquigarrow} q'' \mid \mathsf{read}(p) = \epsilon\} \\
\mathsf{Paths}(i+1,q'') = & \{p'p \mid \exists q \in S_i.\, \exists a, q'.\, q \xrightarrow{a} q' \overset{p}{\rightsquigarrow} q'' \\
 & \qquad \wedge\ \mathsf{read}(p) = \epsilon \\
 & \qquad \wedge\ p' \in \mathsf{Paths}(i,q)\}
\end{array} \tag{A.1}$$

Using this definition, any single path $p \in \mathsf{Paths}(n, q^f)$ can be recovered in linear time by processing the log in reverse order. In each step $i > 0$, we pick some $q \in S_i$ such that the condition in A.1 is satisfied, which can be checked by computing the preimage of the ϵ-closure of q''. Note in particular that we do not need the input string for this. $\mathsf{write}(p)$ gives a bit-coded parse tree, though not necessarily the lexicographically least. We need a way to locally choose $q \in S_i$ such that the lexicographically least path is constructed without backtracking.

We can adapt the NFA-simulation by keeping each state set S_i in a particular order: If $\mathsf{Reach}(\{q^s\}, a_1 \ldots a_i) = \{q_{i1}, \ldots q_{ij_i}\}$ then order the q_{ij} according to the lexicographic order of the paths reaching them. Intuitively, the highest ranked state in S_i is on the greedy path if the remaining input is accepted from this state; if not, the second-highest ranked is on the greedy path, if the remaining input is accepted; and so on. Using this, we can resolve the choice of q in (A.1) and define a function which recovers the lexicographically least bit-code $\mathsf{Path}(n, q^f)$ from the log:

$$\begin{aligned}
\mathsf{Path}(0, q'') &= \min_{\prec}\{p \mid q^s \overset{p}{\rightsquigarrow} q'' \mid \mathsf{read}(p) = \epsilon\} \\
\mathsf{Path}(i+1, q'') &= \mathsf{Path}(i, q)\mathsf{write}(p) \\
&\qquad \text{where } q \in S_i \text{ is highest ranked such that} \\
&\qquad \exists a, q'.\, q \xrightarrow{a} q' \overset{p}{\rightsquigarrow} q'' \wedge \mathsf{read}(p) = \epsilon
\end{aligned}$$

The NFA-simulation can be refined to construct properly ordered state sequences instead of sets without asymptotic slow-down. The log, however, is adversely affected by this. We need $\lceil m \lg m \rceil$ bits per input symbol, for a total of $\lceil mn \lg m \rceil$ bits.

The key property for allowing us to list a state at most once in an ordered state sequence is this:

Lemma A.4.1. *Let s, t_1, t_2, and t be states in an aNFA M, and let p_1, p_2, q_1, q_2 be paths in M such that $s \overset{p_1}{\rightsquigarrow} t_1$, $s \overset{p_2}{\rightsquigarrow} t_2$, and $t_1 \overset{q_1}{\rightsquigarrow} t$, $t_2 \overset{q_2}{\rightsquigarrow} t$, where p_1 is not a prefix of p_2. If* $\mathsf{write}(p_1) \prec \mathsf{write}(p_2)$ *then* $\mathsf{write}(p_1 q_1) \prec \mathsf{write}(p_2 q_2)$

Proof. Application of the lexicographical ordering on paths. □

A.5 Lean-log Algorithm

We can do better than saving a log where each element is a full sequence of NFA states. Since the join states J_M of an aNFA M become the choice states $C_{\overline{M}}$ of the reverse aNFA $\overline{M}$ we only need to construct one "direction" bit for each join state at each input string position. It is not necessary to record any states in the log at all. This results in an algorithm that requires only k bits per input symbol for the log, where k is the number of Kleene-stars and alternatives occurring in the RE. It can be shown that $k \leq \frac{1}{3}m$; in practice we can observe $k << m$.

Instead of writing down state sequences, we write down *log frames* which are partial maps $L : J_M \rightarrow \{\overline{0}, \overline{1}\}$. The subset of J_M on which L is defined is denoted $\mathrm{dom}(L)$. The empty log frame is $\emptyset$, and the disjoint union of two log frames L, L' is written as $L \cup L'$. The set of all log frames is Frame_M. A modified closure algorithm computes both a state sequence and a log frame:

$$\mathsf{Close}(q, L) : Q_M \times \mathsf{Frame}_M \to Q_M^* \times \mathsf{Frame}_M$$

$$\mathsf{Close}(q, L) = \begin{cases} (\vec{q}\vec{q}', L'') & \text{if } q \xrightarrow{0} q_0 \wedge q \xrightarrow{1} q_1 \\ & \quad \wedge \mathsf{Close}(q_0, L) = (\vec{q}, L') \\ & \quad \wedge \mathsf{Close}(q_1, L') = (\vec{q}', L'') \\ \mathsf{Close}(q', L \cup \{q' \mapsto t\}) & \text{if } q \xrightarrow{t} q' \wedge t \in \{\overline{0}, \overline{1}\} \\ & \quad \wedge q' \notin \mathrm{dom}(L) \\ (\epsilon, L) & \end{cases}$$

Computing $\mathsf{Close}(q, \emptyset) = (\vec{q}, L)$ results in the sequence of states $\vec{q}$ in the "frontier" of the ϵ-closure of q, ordered according to their lexicographic order, and a log frame L which uniquely identifies the lexicographically least ϵ-path from q to any state in $\vec{q}$. Note that the algorithm works by backtracking and stops when a join state has previously been encountered. This is sound since the previous encounter must have been via a higher ranked path, and since any extension of the path continues to have higher rank by Lemma A.4.1.

The closure algorithm is extended to state sequences by applying the statewise closure algorithm in ranking order, using the same log frame:

$$\begin{aligned} \mathsf{Close}^* &: Q_M^* \times \mathsf{Frame}_M \to Q_M^* \times \mathsf{Frame}_M \\ \mathsf{Close}^*(\epsilon, L) &= L \\ \mathsf{Close}^*(q\,\vec{q}, L) &= (\vec{q}'\vec{q}'', L'') \\ &\quad \text{where } (\vec{q}', L') = \mathsf{Close}(q, L) \\ &\quad \text{and } (\vec{q}'', L'') = \mathsf{Close}^*(\vec{q}, L') \end{aligned}$$

The modified algorithm $\mathsf{Step} : Q_M^* \times \Sigma \to Q_M^*$ is defined on single states $q \in Q_M$ by

$$\mathsf{Step}(q, a) = \begin{cases} q' & \text{if } q \xrightarrow{a} q' \\ \epsilon & \text{otherwise} \end{cases}$$

and extended homomorphically to sequences Q_M^*. The forward simulation algorithm is essentially the same process as before, but now explicitly maintains a sequence of log frames $\vec{L}$:

$$\begin{aligned} \mathsf{Reach} &: Q_M^* \times \Sigma^* \to Q_M^* \times \mathsf{Frame}_M^* \\ \mathsf{Reach}(\vec{q}, \epsilon) &= (\vec{q}, \epsilon) \\ \mathsf{Reach}(\vec{q}, a\,s') &= (\vec{q}'', L\vec{L}) \\ &\quad \text{where } (\vec{q}', L) = \mathsf{Close}^*(\mathsf{Step}(\vec{q}, a), \emptyset) \\ &\quad \text{and } (\vec{q}'', \vec{L}) = \mathsf{Reach}(\vec{q}', s') \end{aligned}$$

Let $s = a_1 \ldots a_n$. Computing $\mathsf{Reach}(\vec{q_0}, s)$ where $(\vec{q_0}, L_0) = \mathsf{Close}(q^s, \emptyset)$ results in a pair $(q_1 q_2 ... q_m, L_1 ... L_n)$. If for any $1 \leq k \leq m$ we have $q_k = q^f$, then the the lexicographically least path $q^s \overset{p}{\rightsquigarrow} q^f$ with $\mathsf{read}(p) = s$ exists, and the sequence $L_0 L_1 ... L_n$ can be used to effectively reconstruct its bit-code $\mathsf{Path}(q^f, n)$:

$$\begin{aligned} \mathsf{Path} : Q_M \times \{0, 1, ..., n\} &\to \{0, 1\}^* \\ \mathsf{Path}(q^s, 0) &= \epsilon \\ \mathsf{Path}(q', i) &= \begin{cases} \mathsf{Path}(q, i-1) & \text{if } \exists a \in \Sigma .\, q \xrightarrow{a} q' \\ \mathsf{Path}(q_{L_i(q)}, i) & \text{if } q_{\bar{0}} \xrightarrow{\bar{0}} q' \text{ and } q_{\bar{1}} \xrightarrow{\bar{1}} q' \\ \mathsf{Path}(q, i) b & \text{if } q \xrightarrow{b} q' \text{ and } b \in \{0, 1\} \end{cases} \end{aligned}$$

The forward Reach algorithm keeps the aNFA and the current character in working memory, requiring $O(m)$ words of random access memory (RAM), writing nk bits to the log, and discarding the input string. The backward Path algorithm also requires $O(m)$ words of RAM and reads from the log in reverse write order. The log is thus a 2-phase stack: In the first pass it is only pushed to, in the second pass popped from.

Both Close^* and Step run in time $O(m)$ per input symbol, hence the forward pass requires time $O(mn)$. Likewise, the backward pass requires time $O(mn)$.

A.6 Evaluation

We have implemented the optimized algorithms in C and in Haskell, and we compare the performance of the C implementation with the following existing RE tools:

RE2: Google's RE implementation, available from [11].

Tcl: The scripting language Tcl [10].

Perl: The scripting language Perl [13].

Grep: The UNIX tool `grep`.

Rcp: The implementation of the algorithm *"DFASIM"* from [9]. It is based on Dubé and Feeley's method [2], but altered to produce a bit-coded parse tree.

FrCa: The implementation of the algorithm*"FrCa"* algorithm used in [9]. It is based on Frisch and Cardelli's method from [5].

In the subsequent plots, our implementation of the lean-log algorithm is referred to as *BitC*.

The tests have been performed on an Intel Xeon 2.5 GHz machine running GNU/Linux 2.6.

Pathological Expressions

To get an indication of the "raw" throughput for each tool, $\mathtt{a}^\star$ was run on sequences of as (Figure A.2a). (Note that the plots use log scales on both axes, so as to accommodate the dramatically varying running times.) Perl outperforms the rest, likely due to a strategy where it falls back on a simple scan of the input. FrCa stores each position in the input string from which a match can be made, which in this case is every position. As a result, FrCa uses significantly more memory than the rest, causing a dramatic slowdown.

The expression $(\mathtt{a}|\mathtt{b})^\star\mathtt{a}(\mathtt{a}|\mathtt{b})^n$ with the input $(\mathtt{ab})^{n/2}$ is a worst-case for DFA-based methods, as it results in a number of states exponential in n. Perl has been omitted from the plots, as it was prohibitively slow. Tcl, Rcp, and Grep all perform orders of magnitude slower than FrCa, RE2, and BitC (Figure A.2b), indicating that Tcl and Grep also use a DFA for this expression. If we fix n to 25, it becomes clear that FrCa is slower than the rest, likely due to high memory consumption as a result of its storing all positions in the input string (Figure A.2c). The asymptotic running times of the others appear to be similar to each other, but with greatly varying constants.

For the backtracking worst-case expression $(\mathtt{a?})^n\mathtt{a}^n$ in Figure A.3a, BitC performs roughly like RE2.[2] In contrast to Rcp and FrCa, which are both highly sensitive to the *direction* of non-determinism, BitC has the same performance for both $(\mathtt{a?})^n\mathtt{a}^n$ and $\mathtt{a}^n(\mathtt{a?})^n$ (Figure A.3b).

Practical Examples

We have run the comparisons with various "real-life" examples of REs taken from [12], all of which deal with expressions matching e-mail addresses. In Figure A.4b, BitC is significantly slower than in the other examples. This can likely be ascribed to heavy use of bounded repetitions in this expression, as they are currently just rewritten into concatenations and alternations in our implementation.

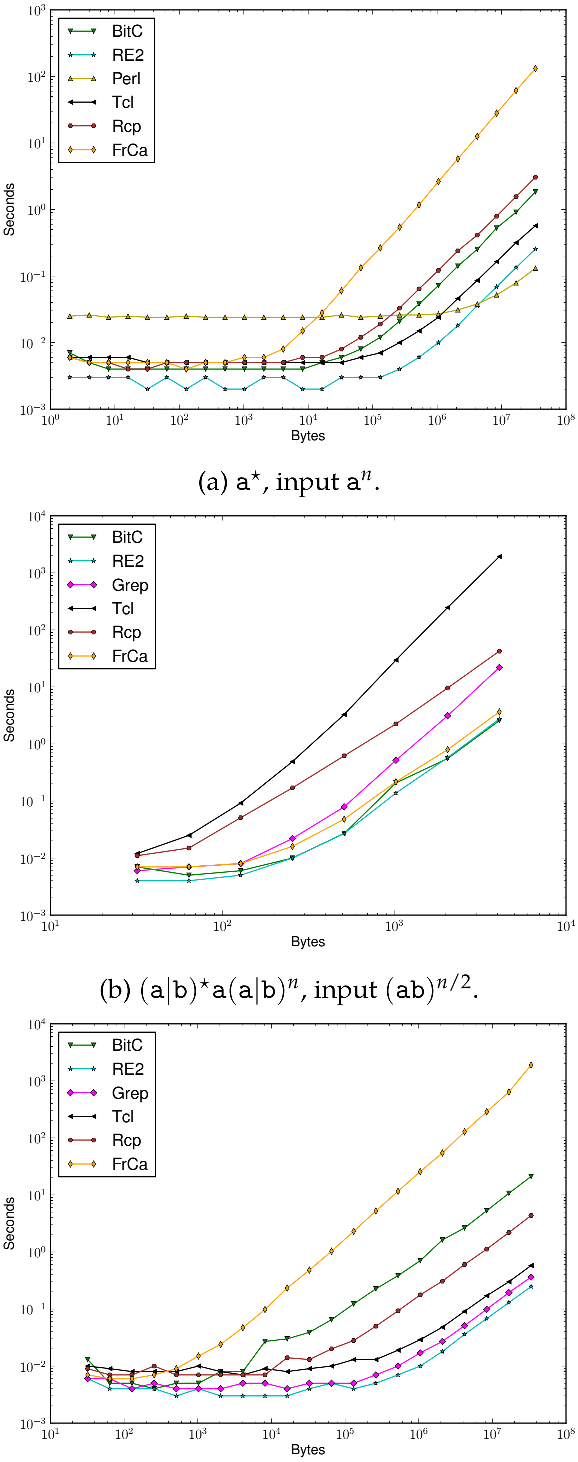
In the other two cases, BitC's performance is roughly like that of Grep. This is promising for BitC since Grep performs only RE *matching*, not full *parsing*. RE2 is consistently ranked as the fastest program in our benchmarks, presumably due to its aggressive optimizations and ability to dynamically choose between several strategies. Recall that RE2 performs greedy leftmost subgroup matching, not full parsing. Our present prototype of BitC is coded

[2] The expression parser in BitC failed for the largest expressions, which is why they are not on the plot.

in less than 1000 lines of C. It uses only standard libraries and performs no optimizations such as NFA-minimization, DFA-construction, cached or parallel NFA-simulation, etc. This is future work.

A.7 Related work

The known RE parsing algorithms can be divided into four categories. The first category is Perl-style backtracking used in many tools and libraries for RE subgroup matching [1]; it has an exponential worst case running time, but always produces the greedy parse and enables some extensions to REs such as backreferences. Another category consists of context-free parsing methods, where the RE is first translated to a context-free grammar, before a general context-free parsing algorithm such as Earley's [3] using cubic time is applied. An interesting CFG method is derivatives-based parsing [8]. While efficient parsers exist for subsets of unambiguous context-free languages, this restriction propagates to REs, and thus these parsers can only be applied for subsets of unambiguous REs. The third category contains RE scalable parsing algorithms that do not always produce the greedy parse. This includes NFA and DFA based algorithms provided by Dubé and Feeley [2] and Nielsen and Henglein [9], where the RE is first converted to an NFA with additional information used to parse strings or to create a DFA preserving the additional information for parsing. This category also includes the algorithm by Fischer, Huch and Wilke [4]; it is left out of our tests since its Haskell-based implementation often turned out not to be competitive with the other tools. The last category consists of the algorithms that scale well and always produce greedy parse trees. Kearns [7] and Frisch and Cardelli [5] reverse the input; perform backwards NFA-simulation, building a log of NFA-states reached at each input position; and construct the greedy parse tree in a final forward pass over the input. They require storing the input symbol plus m bits per input symbol for the log. This can be optimized to storing bits proportional to the number of NFA-states reached at a given input position [9], although the worst case remains the same. Our lean log algorithm uses only 2 passes, does not require storing the input symbols and stores only $k < \frac{1}{3}m$ bits per input symbol in the string.

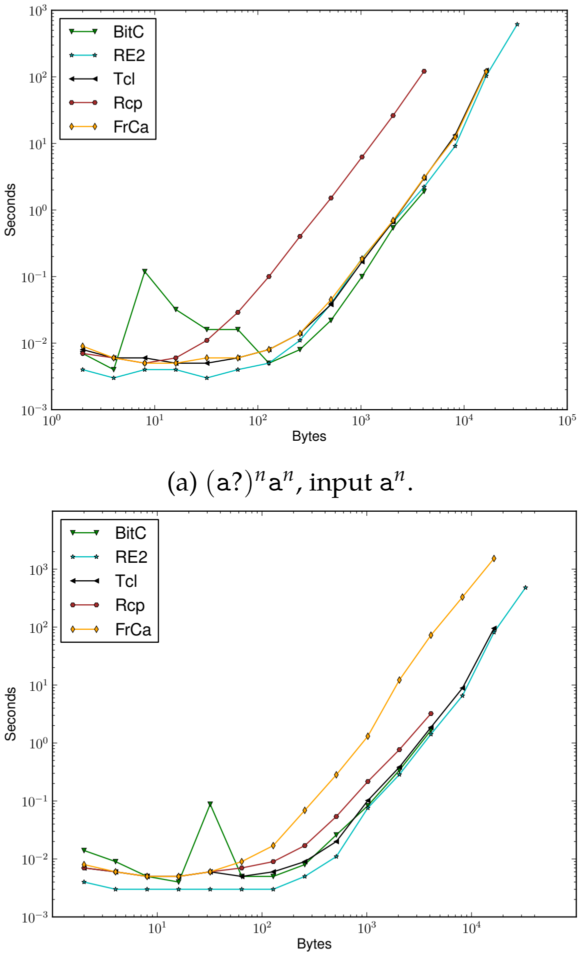


(a) $\mathtt{a}^\star$, input $\mathtt{a}^n$.

(b) $(\mathtt{a}|\mathtt{b})^\star\mathtt{a}(\mathtt{a}|\mathtt{b})^n$, input $(\mathtt{ab})^{n/2}$.

(c) $(\mathtt{a}|\mathtt{b})^\star\mathtt{a}(\mathtt{a}|\mathtt{b})^{25}$, input $(\mathtt{ab})^{n/2}$.

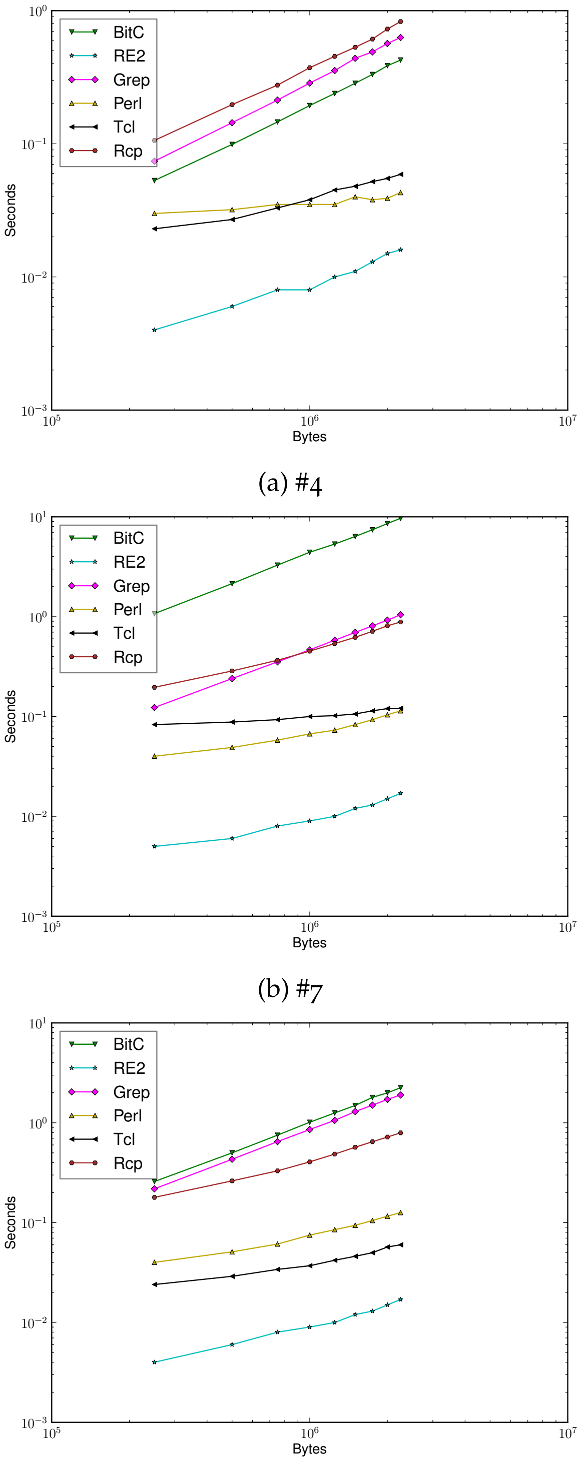
Figure A.2: Comparisons using very simple iteration expressions.



(a) (a?)^{n}a^n, input a^n.

(b) a^n(a?)^n, input a^n.

Figure A.3: Comparison using a backtracking worst case expression, and its reversal.



(a) #4

(b) #7

(c) #8

Figure A.4: Comparison using various e-mail expressions.

Bibliography

[1] R. Cox. Regular Expression Matching can be Simple and Fast. http://swtch.com/ rsc/regexp/regexp1.html.

[2] D. Dubé and M. Feeley. Efficiently Building a Parse Tree From a Regular Expression. *Acta Informatica*, 37(2):121–144, 2000.

[3] J. Earley. An Efficient Context-Free Parsing Algorithm. *Communications of the ACM*, 13(2):94–102, 1970.

[4] S. Fischer, F. Huch, and T. Wilke. A Play on Regular Expressions: Functional Pearl. In *Proc. of the 15th ACM SIGPLAN International Conference on Functional Programming*, ICFP '10, pages 357–368, New York, NY, USA, 2010. ACM.

[5] 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 (LNCS)*, pages 618–629. Springer, July 2004.

[6] F. Henglein and L. Nielsen. Declarative Coinductive Axiomatization of Regular Expression Containment and its Computational Interpretation (Preliminary Version). TOPPS D-Report 612, Department of Computer Science, University of Copenhagen (DIKU), February 2010.

[7] S. M. Kearns. *Extending Regular Expressions*. PhD thesis, Columbia University, 1990.

[8] M. Might, D. Darais, and D. Spiewak. Parsing with derivatives: A functional pearl. In *ACM SIGPLAN Notices*, volume 46, pages 189–195. ACM, 2011.

[9] L. Nielsen and F. Henglein. Bit-coded Regular Expression Parsing. In *Proc. 5th Int'l Conf. on Language and Automata Theory and Applications (LATA)*, volume 6638 of *Lecture Notes in Computer Science (LNCS)*, pages 402–413. Springer, May 2011.

[10] J. Ousterhout. Tcl: An Embeddable Command Language. In *Proc. USENIX Winter Conference*, pages 133–146, January 1990.

[11] The RE2 authors. RE2. `https://github.com/google/re2`, 2015.

[12] M. V. M. Veanes, P. de Halleux, and N. Tillmann. Rex: Symbolic Regular Expression Explorer. In *Proc. 3d Int'l Conf. on Software Testing, Verification and Validation*, Paris, France, April 6-10 2010. IEEE Computer Society Press.

[13] L. Wall, T. Christiansen, and J. Orwant. *Programming Perl*. O'Reilly Media, Incorporated, 2000.

Paper B

Optimally Streaming Greedy Regular Expression Parsing

This paper has been published in the following:

> Niels Bjørn Bugge Grathwohl, Fritz Henglein and Ulrik Terp Rasmussen. "Optimally Streaming Greedy Regular Expression Parsing". In *Proceedings 11th International Colloquium on Theoretical Aspects of Computing (ICTAC)*. Pages 224-240. Springer, 2014. DOI: 10.1007/978-3-319-10882-7_14.

The enclosed version has been reformatted to fit the layout of this dissertation.

Optimally Streaming Greedy Regular Expression Parsing[1]

Niels Bjørn Bugge Grathwohl, Fritz Henglein and Ulrik Terp Rasmussen

Department of Computer Science, University of Copenhagen (DIKU)

Abstract

We study the problem of *streaming* regular expression parsing: Given a regular expression and an input stream of symbols, how to output a serialized syntax tree representation as an output stream *during* input stream processing.

We show that *optimally streaming* regular expression parsing, outputting bits of the output as early as is semantically possible for any regular expression of size m and any input string of length n, can be performed in time $O(2^{m \log m} + mn)$ on a unit-cost random-access machine. This is for the wide-spread *greedy* disambiguation strategy for choosing parse trees of grammatically ambiguous regular expressions. In particular, for a fixed regular expression, the algorithm's run-time scales linearly with the input string length. The exponential is due to the need for preprocessing the regular expression to analyze state coverage of its associated NFA, a PSPACE-hard problem, and tabulating all reachable *ordered* sets of NFA-states.

Previous regular expression parsing algorithms operate in multiple phases, *always* requiring processing or storing the whole input string before outputting the first bit of output, not only for those regular expressions and input prefixes where reading to the end of the input is strictly *necessary*.

B.1 Introduction

In programming, regular expressions are often used to extract information from an input, which requires an intensional interpretation of regular expressions as denoting parse trees, and not just their ordinary language-theoretic interpretation as denoting strings.

This is a nontrivial change of perspective. We need to deal with grammatical ambiguity—*which* parse tree to return, not just that it has one—and memory requirements become a critical factor: Deciding whether a string belongs to the language denoted by $(\mathtt{ab})^\star + (\mathtt{a}+\mathtt{b})^\star$ can be done in constant space, but outputting the first bit, whether the string matches the first alternative or only the second, may require buffering the whole input string. This is an instructive case of deliberate grammatical ambiguity to be resolved by the prefer-the-left-alternative policy of greedy disambiguation: Try to match the

[1] The order of authors is insignificant.

left alternative; if that fails, return a match according to the right alternative as a fallback. Straight-forward application of automata-theoretic techniques does not help: $(\mathtt{ab})^\star + (\mathtt{a} + \mathtt{b})^\star$ denotes the same *language* as $(\mathtt{a} + \mathtt{b})^\star$, which is unambiguous and corresponds to a small DFA, but is also useless: it doesn't represent any more when a string consists of a sequence of *ab*-groups.

Previous parsing algorithms [9, 3, 5, 10, 13, 6] require at least one full pass over the input string before outputting any output bits representing the parse tree. This is the case even for regular expressions requiring only bounded lookahead such as one-unambiguous regular expressions [1].

In this paper we study the problem of *optimally streaming* parsing. Consider

$$(\mathtt{ab})^\star + (\mathtt{a} + \mathtt{b})^\star,$$

which is ambiguous and in general requires unbounded input buffering, and consider the particular input string

$$\mathtt{ab} \ldots \mathtt{abaabababab} \ldots .$$

An *optimally* streaming parsing algorithm needs to buffer the prefix $\mathtt{ab} \ldots \mathtt{ab}$ in some form because the complete parse might match either of the two alternatives in the regular expression, but once encountering aa, only the right alternative is possible. At this point it outputs this information and the output representation for the buffered string as parsed by the second alternative. After this, it outputs a bit for each input symbol read, with no internal buffering: input symbols are discarded before reading the next symbol. Optimality means that output bits representing the eventual parse tree must be produced *earliest possible*: as soon as they are semantically determined by the input processed so far under the assumption that the parse will succeed.

Outline. In Section B.2 we recall the *type interpretation* of regular expressions, where a regular expression denotes parse trees, along with the *bit-coding* of parse trees.

In Section B.3 we introduce a class of Thompson-style augmented nondeterministic finite automata (aNFAs). Paths in such an aNFA naturally represent *complete* parse trees, and paths to intermediate states represent *partial* parse trees for prefixes of an input string.

We recall the greedy disambiguation strategy in Section B.4, which specifies a deterministic mapping of accepted strings to NFA-paths.

Section B.5 contains a definition of what it means to be an optimally streaming implementation of a parsing function.

We define what it means for a set of aNFA-states to *cover* another state in Section B.6, which constitutes the computationally hardest part needed in our algorithm.

Section B.7 contains the main results. We present *path trees* as a way of organizing partial parse trees, and based on these we present our algorithm for an optimally streaming parsing function and analyze its asymptotic runtime complexity.

Finally, in Section B.8, the algorithm is demonstrated by illustrative examples alluding to its expressive power and practical utility.

B.2 Preliminaries

In the following section, we recall definitions of regular expressions and their interpretation as types [10].

Definition 5 (Regular expression). A regular expression (RE) over a finite alphabet Σ is an expression E generated by the grammar

$$E ::= \mathbf{0} \mid \mathbf{1} \mid a \mid E_1 E_2 \mid E_1 + E_2 \mid E_1^\star$$

where $a \in \Sigma$.

Concatenation (juxtaposition) and alternation ($+$) associates to the right; parentheses may be inserted to override associativity. Kleene star ($\star$) binds tightest, followed by concatenation and alternation.

The standard interpretation of regular expressions is as descriptions of regular languages.

Definition 6 (Language interpretation). Every RE E denotes a language $\mathcal{L}[\![E]\!] \subseteq \Sigma^\star$ given as follows:

$$\begin{array}{lll} \mathcal{L}[\![\mathbf{0}]\!] = \emptyset & \mathcal{L}[\![E_1 E_2]\!] = \mathcal{L}[\![E_1]\!]\mathcal{L}[\![E_2]\!] & \mathcal{L}[\![a]\!] = \{a\} \\ \mathcal{L}[\![\mathbf{1}]\!] = \{\epsilon\} & \mathcal{L}[\![E_1 + E_2]\!] = \mathcal{L}[\![E_1]\!] \cup \mathcal{L}[\![E_2]\!] & \mathcal{L}[\![E_1^\star]\!] = \bigcup_{n \geq 0} \mathcal{L}[\![E_1]\!]^n \end{array}$$

where we have $A_1 A_2 = \{w_1 w_2 \mid w_1 \in A_1, w_2 \in A_2\}$, and $A^0 = \{\epsilon\}$ and $A^{n+1} = AA^n$.

Proviso: Henceforth we shall restrict ourselves to REs E such that $\mathcal{L}[\![E]\!] \neq \emptyset$.

For regular expression parsing, we consider an alternative interpretation of regular expressions as types.

Definition 7 (Type interpretation). Let the syntax of *values* be given by

$$v ::= () \mid \mathsf{inl}\, v_1 \mid \mathsf{inr}\, v_1 \mid \langle v_1, v_2 \rangle \mid [v_1, v_2, ..., v_n]$$

Every RE E can be seen as a *type* describing a set $\mathcal{T}[\![E]\!]$ of well-typed values:

$$\begin{array}{ll} \mathcal{T}[\![\mathbf{0}]\!] = \emptyset & \mathcal{T}[\![E_1 E_2]\!] = \{\langle v_1, v_2 \rangle \mid v_1 \in \mathcal{T}[\![E_1]\!], v_2 \in \mathcal{T}[\![E_2]\!]\} \\ \mathcal{T}[\![\mathbf{1}]\!] = \{()\} & \mathcal{T}[\![E_1 + E_2]\!] = \{\mathsf{inl}\, v \mid v \in \mathcal{T}[\![E_1]\!]\} \cup \{\mathsf{inr}\, v \mid v \in \mathcal{T}[\![E_2]\!]\} \\ \mathcal{T}[\![a]\!] = \{a\} & \mathcal{T}[\![E_1^\star]\!] = \{[v_1, \ldots, v_n] \mid n \geq 0 \wedge \forall 1 \leq i \leq n. v_i \in \mathcal{T}[\![E_1]\!]\} \end{array}$$

We write $|v|$ for the *flattening* of a value, defined as the word obtained by doing an in-order traversal of v and writing down all the symbols in the order they are visited. We write $\mathcal{T}_w[\![E]\!]$ for the restricted set $\{v \in \mathcal{T}[\![E]\!] \mid |v| = w\}$. Regular expression *parsing* is a generalization of the acceptance problem of determining whether a word w belongs to the language of some RE E, where additionally we produce a parse tree from $\mathcal{T}_w[\![E]\!]$. We say that an RE E is *ambiguous* iff there exists a w such that $|\mathcal{T}_w[\![E]\!]| > 1$.

Any well-typed value can be serialized into a sequence of bits.

Definition 8 (Bit-coding)**.** Given a value $v \in \mathcal{T}[\![E]\!]$, we denote its *bit-code* by $\ulcorner v \urcorner \subseteq \{0, 1\}^\star$, defined as follows:

$$\begin{array}{rcr} \ulcorner () \urcorner = \epsilon & \ulcorner a \urcorner = \epsilon & \ulcorner \mathsf{inl}\ v \urcorner = 0 \ulcorner v \urcorner \\ \ulcorner \langle v_1, v_2 \rangle \urcorner = \ulcorner v_1 \urcorner \ulcorner v_2 \urcorner & \ulcorner [v_1, ..., v_n] \urcorner = 0 \ulcorner v_1 \urcorner ... 0 \ulcorner v_n \urcorner 1 & \ulcorner \mathsf{inr}\ v \urcorner = 1 \ulcorner v \urcorner \end{array}$$

We write $\mathcal{B}[\![E]\!]$ for the set $\{\ulcorner v \urcorner \mid v \in \mathcal{T}[\![E]\!]\}$ and $\mathcal{B}_w[\![E]\!]$ for the set restricted to bit-codes for values with a flattening w. Note that for any RE E, bit-coding is an isomorphism when seen as a function $\ulcorner \cdot \urcorner_E : \mathcal{T}[\![E]\!] \to \mathcal{B}[\![E]\!]$.

B.3 Augmented Automata

In this section we recall from an earlier paper [6] the construction of finite automata from regular expressions. Our construction is similar to that of Thompson [15], but augmented with extra annotations on non-deterministic ϵ-transitions. The resulting state machines can be seen as non-deterministic transducers which for each accepted input string in the language of the underlying regular expression outputs the bit-codes for the corresponding parse trees.

Definition 9 (Augmented non-deterministic finite automaton)**.** An *augmented non-deterministic finite automaton* (aNFA) is a tuple $(\mathsf{State}, \delta, q^{\mathsf{in}}, q^{\mathsf{fin}})$, where State is a finite set of *states*, $q^{\mathsf{in}}, q^{\mathsf{fin}} \in \mathsf{State}$ are *initial* and *final* states, respectively, and $\delta \subseteq \mathsf{State} \times \Gamma \times \mathsf{State}$ is a labeled transition relation with labels $\Gamma = \Sigma \uplus \{0, 1, \epsilon\}$.

Transition labels are divided into the disjoint sets Σ (symbol labels); $\{0, 1\}$ (bit-labels); and $\{\epsilon\}$ (ϵ-labels). Σ-transitions can be seen as input actions, and bit-transitions as output actions.

Definition 10 (aNFA construction)**.** Let E be an RE and define an aNFA $M_E = (\mathsf{State}_E, \delta_E, q^{\mathsf{in}}_E, q^{\mathsf{fin}}_E)$ by induction on E. We give the definition diagrammatically by cases:

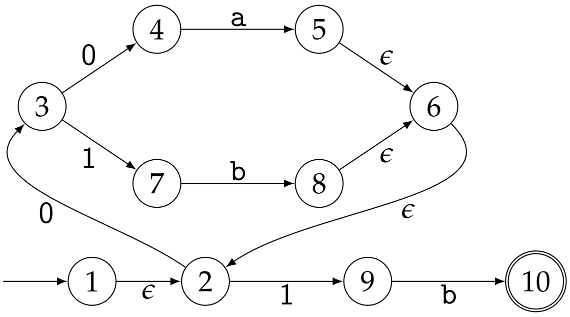
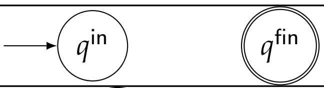
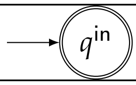
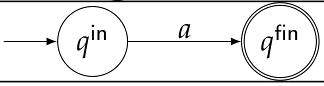
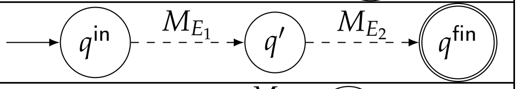
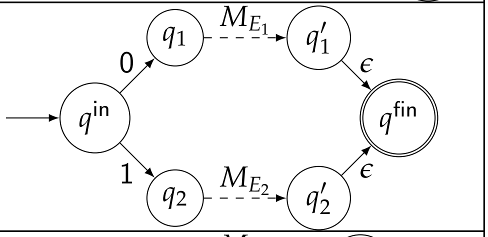
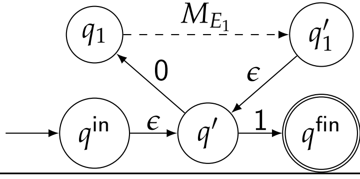


Figure B.1: Example automaton for the RE $(\mathtt{a} + \mathtt{b})^\star\mathtt{b}$

E	M_E
$\mathbf{0}$	
$\mathbf{1}$	 ($q^{\text{in}} = q^{\text{fin}}$)
a	
E_1E_2	
$E_1 + E_2$	
$E_1^\star$	

In the above, the notation $q_1 \overset{M}{\dashrightarrow} q_2$ means that q_1, q_2 are initial and final states, respectively, in some (sub-)automaton M.

See Figure B.1 for an example.

Definition 11 (Path). A *path* in an aNFA is a finite and non-empty sequence $\alpha \in \mathsf{State}^\star$ of the form $\alpha = p_0\, p_1 \ldots p_{n-1}$ such that for each $i < n$, we have $(p_i, \gamma_i, p_{i+1}) \in \delta_E$ for some γ_i. As a shorthand for this fact we might write $p_0 \overset{\alpha}{\rightsquigarrow} p_{n-1}$ (note that a single state is a path to itself).

Each path α is associated with a (possibly empty) sequence of labels $\mathsf{lab}(\alpha)$: we let $\mathsf{read}(\alpha)$ and $\mathsf{write}(\alpha)$ refer to the corresponding subsequences of $\mathsf{lab}(\alpha)$ filtered by Σ and $\{0,1\}$, respectively. An automaton *accepts* a word w iff we have $q^{\mathsf{in}} \overset{\alpha}{\rightsquigarrow} q^{\mathsf{fin}}$ for some α where $\mathsf{read}(\alpha) = w$. There is a one-to-one correspondence between bit-codes and accepting paths:

Proposition B.3.1. *For any RE E with aNFA M_E, we have for each $w \in \mathcal{L}[\![E]\!]$ that*

$$\{\mathsf{write}(\alpha) \mid q^{\mathsf{in}} \overset{\alpha}{\rightsquigarrow} q^{\mathsf{fin}} \wedge \mathsf{read}(\alpha) = w\} = \mathcal{B}_w[\![E]\!].$$

Determinization. Given a state set Q, define its *closure* as the set

$$\mathsf{closure}(Q) = \{q' \mid q \in Q \wedge \exists\alpha.\mathsf{read}(\alpha) = \epsilon \wedge q \overset{\alpha}{\rightsquigarrow} q'\}.$$

For any aNFA $M = (\mathsf{State}, \delta, q^{\mathsf{in}}, q^{\mathsf{fin}})$, let $D(M) = (\mathsf{DState}_M, I_M, F_M, \Delta_M)$ be the deterministic automaton obtained by applying the standard subset sum construction: Here, $I_M = \mathsf{closure}(\{q^{\mathsf{in}}\})$ is the *initial state*, and $\mathsf{DState}_M \subseteq \mathbf{2}^{\mathsf{State}}$ is the set of states, defined to be the smallest set containing I_M and closed under the transition function: for all $a \in \Sigma$ and $Q \in \mathsf{DState}_M$, we have $\Delta_M(Q, a) \in \mathsf{DState}_M$, where

$$\Delta_M(Q, a) = \mathsf{closure}(\{q' \mid (q, a, q') \in \delta, q \in Q\}).$$

The set of *final states* is $F_M = \{Q \in \mathsf{DState}_M \mid q^{\mathsf{fin}} \in Q\}$.

B.4 Disambiguation

A regular expression parsing algorithm has to produce a parse tree for an input word whenever the word is in the language for the underlying RE. In the case of ambiguous REs, the algorithm has to choose one of several candidates. We do not want the choice to be arbitrary, but rather a parse tree which is uniquely identified by a *disambiguation policy*. Since there is a one-to-one correspondence between words in the language of an RE E and accepting paths in M_E, a disambiguation policy can be seen as a deterministic choice between aNFA paths recognizing the same string.

We will focus on greedy disambiguation, which corresponds to choosing the first result that would have been found by a backtracking regular expression parsing algorithm such as the one found in the Perl programming language [16]. The greedy strategy has successfully been implemented in previous work [5, 6], and is simpler to define and implement than other strategies such as POSIX [8, 4] whose known parsing algorithms are technically more complicated [11, 13, 14].

Greedy disambiguation can be seen as picking the accepting path with the lexicographically least bitcode. A well-known problem with backtracking parsing is non-termination in the case of regular expressions with nullable subexpressions under Kleene star, which means that the lexicographically least path is not always well-defined. This problem can easily be solved by not considering paths with non-productive loops, as in [5].

B.5 Optimal Streaming

In this section we specify what it means to be an *optimally streaming* implementation of a function from sequences to sequences.

We write $w \sqsubseteq w''$ if w is a *prefix* of w'', that is $ww' = w''$ for some w'. Note that $\sqsubseteq$ is a partial order with greatest lower bounds for nonempty sets: $\bigsqcap L = w$ if $w \sqsubseteq w''$ for all $w'' \in L$ and $\forall w'.(\forall w'' \in S.w' \sqsubseteq w'') \Rightarrow w' \sqsubseteq w$. $\bigsqcap L$ is the longest common prefix of all words in L.

Definition 12 (Completions)**.** The set of *completions* $C_E(w)$ of w in E is the set of all words in $\mathcal{L}[\![E]\!]$ that have w as a prefix:

$$C_E(w) = \{w'' \mid w \sqsubseteq w'' \wedge w'' \in \mathcal{L}[\![E]\!]\}.$$

Note that $C_E(w)$ may be empty.

Definition 13 (Extension)**.** For nonempty $C_E(w)$ the unique *extension* $\hat{w}_E$ of w under E is the longest extension of w with a suffix such that all successful extensions of w to an element of $\mathcal{L}[\![E]\!]$ are also extensions of $\hat{w}$:

$$\hat{w}_E = \bigsqcap C_E(w).$$

Word w is *extended* under E if $w = \hat{w}$; otherwise it is unextended.

Extension is a closure operation: $\hat{\hat{w}} = \hat{w}$; in particular, extensions are extended.

Definition 14 (Reduction)**.** For empty $C_E(w)$ the unique *reduction* $\bar{w}_E$ of w under E is the longest prefix w' of w such that $C_E(w') \neq \emptyset$.

Given parse function $\mathsf{P}_E(\cdot) : \mathcal{L}[\![E]\!] \to \mathcal{B}[\![E]\!]$ for complete input strings, we can now define what it means for an implementation of it to be optimally streaming:

Definition 15 (Optimally streaming)**.** The *optimally streaming* function corresponding to $\mathsf{P}_E(\cdot)$ is

$$O_E(w) = \begin{cases} \bigsqcap\{\mathsf{P}_E(w'') \mid w'' \in C_E(w)\} & \text{if } C_E(w) \neq \emptyset \\ (\bigsqcap O_E(\bar{w}))\sharp & \text{if } C_E(w) = \emptyset. \end{cases}$$

The first condition expresses that after seeing prefix w the function must output *all* bits that are a common prefix of all bit-coded parse trees of words in $\mathcal{L}[\![E]\!]$ that w can be extended to. The second condition expresses that as soon as it is clear that a prefix has no extension to an element of $\mathcal{L}[\![E]\!]$, an indicator $\sharp$ of failure must be emitted, with no further output after that. In this sense O_E is *optimally* streaming: It produces output bits at the semantically earliest possible time during input processing.

It is easy to check that O_E is a streaming function:

$$w \sqsubseteq w' \Rightarrow O_E(w) \sqsubseteq O_E(w')$$

The definition has the, at first glance, surprising consequence that O_E may output bits for parts of the input it has not even read yet:

Proposition B.5.1. $O_E(w) = O_E(\hat{w})$

E.g. for $E = (\mathsf{a} + \mathsf{a})(\mathsf{a} + \mathsf{a})$ we have $O_E(\epsilon) = 00$; that is, O_E outputs 00 off the bat, before reading any input symbols, in anticipation of aa being the only possible successful extension. Assume the input is ab. After reading a it does not output anything, and after reading b it outputs $\sharp$ to indicate a failed parse, the total output being $00\sharp$.

B.6 Coverage

Our algorithm is based on simulating aNFAs in lock-step, maintaining a set of partial paths reading the prefix w of the input that has been consumed so far. In order to be optimally streaming, we have to identify partial paths which are guaranteed not to be prefixes of a greedy parse for a word in $C_E(w)$.

In this section, we define a *coverage relation* which our parsing algorithm relies on in order to detect the aforementioned situation. In the following, fix an RE E and its aNFA $M_E = (\mathsf{State}_E, \delta_E, q^{\mathsf{in}}_E, q^{\mathsf{fin}}_E)$.

Definition 16 (Coverage). Let $p \in \mathsf{State}_E$ be a state and $Q \subseteq \mathsf{State}_E$ a state set. We say that Q *covers* p, written $Q \sqsupseteq p$, iff

$$\{\mathsf{read}(\alpha) \mid q \overset{\alpha}{\rightsquigarrow} q^{\mathsf{fin}}, q \in Q\} \supseteq \{\mathsf{read}(\beta) \mid p \overset{\beta}{\rightsquigarrow} q^{\mathsf{fin}}\} \tag{B.1}$$

Coverage can be seen as a slight generalization of language inclusion. That is, if $Q \sqsupseteq p$, then every word suffix read by a path from p to the final state can also be read by a path from one of the states in Q to the final state.

Let $\overline{M_e}$ refer to the automaton obtained by reversing the direction of all transitions and swapping the initial and final states. It can easily be verified that if (B.1) holds for some Q, p, then the following property also holds in the *reverse* automaton $\overline{M_E}$:

$$\{\mathsf{read}(\alpha) \mid q^{\mathsf{in}} \overset{\alpha}{\rightsquigarrow} q, q \in Q\} \supseteq \{\mathsf{read}(\beta) \mid q^{\mathsf{in}} \overset{\alpha}{\rightsquigarrow} p\} \tag{B.2}$$

If we consider $D(\overline{M_E})$, the deterministic automaton generated from $\overline{M_E}$, then we see that (B.2) is satisfied iff

$$\forall S \in \mathsf{DState}_{\overline{M_E}}.\ p \in S \Rightarrow Q \cap S \neq \emptyset \tag{B.3}$$

This is true since a DFA state S is reachable by reading a word w in $D(\overline{M_E})$ iff every $q \in S$ is reachable by reading w in $\overline{M_E}$. Since a DFA accepts the same language as the underlying aNFA, this implies that condition (B.2) must hold iff Q has a non-empty intersection with *all* DFA states containing p.

The equivalence of (B.1) and (B.3) gives us a method to decide $\sqsupseteq$ in an aNFA M, provided that we have computed $D(\overline{M})$ beforehand. Checking (B.3) for a particular Q and p can be done by intersecting all states of $\mathsf{DState}_{\overline{M_E}}$ with Q, using time $O(|Q||\mathsf{DState}_{\overline{M_E}}|) = O(|Q|2^{O(m)})$, where m is the size of the RE E.

The exponential cost appears to be unavoidable – the problem of deciding coverage is inherently hard to compute:

Proposition B.6.1. *The problem of deciding coverage, that is the set* $\{(E, Q, p) \mid Q \subseteq \mathsf{State}_E \wedge Q \sqsupseteq p\}$*, is PSPACE-hard.*

Proof. We can reduce regular expression equivalence to coverage: Given regular expressions E and F, produce an aNFA M_{E+F} for $E + F$ and observe that M_E and M_F are subautomata. Now observe that there is a path $q^{\mathsf{in}}_{E+F} \overset{\alpha}{\rightsquigarrow} q^{\mathsf{fin}}_E$ (respectively $q^{\mathsf{in}}_{E+F} \overset{\beta}{\rightsquigarrow} q^{\mathsf{fin}}_F$) in M_{E+F} iff there is a path $q^{\mathsf{in}}_E \overset{\alpha'}{\rightsquigarrow} q^{\mathsf{fin}}_E$ with $\mathsf{read}(\alpha) = \mathsf{read}(\alpha')$ in M_E (respectively $q^{\mathsf{in}}_F \overset{\beta'}{\rightsquigarrow} q^{\mathsf{fin}}_F$ with $\mathsf{read}(\beta) = \mathsf{read}(\beta')$ in M_F). Hence, we have $\{q^{\mathsf{in}}_F\} \sqsupseteq q^{\mathsf{in}}_E$ in M_{E+F} iff $\mathcal{L}[\![E]\!] \subseteq \mathcal{L}[\![F]\!]$. Since regular expression containment is PSPACE-complete [12] this shows that coverage is PSPACE-hard. □

Even after having computed a determinized automaton, the decision version of the coverage problem is still NP-complete, which we show by reduction to and from Min-Cover, a well-known NP-complete problem. Let State-Cover refer to the problem of deciding membership for the language

$$\{(M, D(M), p, k) \mid \exists Q.\ |Q| = k \wedge p \notin Q \wedge Q \sqsupseteq p \text{ in } M\}.$$

Recall that Min-Cover is the problem of deciding membership for the language $\{(X, \mathcal{F}, k) \mid \exists \mathcal{C} \subseteq \mathcal{F}. |\mathcal{C}| = k \wedge X = \bigcup \mathcal{C}\}$.

Proposition B.6.2. *State-Cover is NP-complete.*

Proof. State-Cover $\Rightarrow$ Min-Cover: Let $(M, D(M), p, k)$ be given. Define $X = \{S \in \mathsf{DState}_M \mid p \in S\}$ and $\mathcal{F} = \{R_q \mid q \in \bigcup X\}$ where $R_q = \{S \in X \mid q \in S\}$. Then any k-sized set cover $\mathcal{C} = \{R_{q_1}, ..., R_{q_k}\}$ gives a state cover $Q = \{q_1, ..., q_k\}$ and vice-versa.

MIN-COVER ⇒ STATE-COVER: Let $(X, \mathcal{F}, k)$ be given, where $|X| = m$ and $|\mathcal{F}| = n$. Construct an aNFA $M_{X,\mathcal{F}}$ over the alphabet $\Sigma = X \uplus \{\$\}$. Define its states to be the set $\{q^{\mathsf{in}}, q^{\mathsf{fin}}, p\} \cup \{F_1, ..., F_n\}$, and for each F_i, add transitions $F_i \overset{\$}{\rightarrow} q^{\mathsf{fin}}$ and $q^{\mathsf{in}} \overset{x_{ij}}{\rightarrow} F_i$ for each $x_{ij} \in F_i$. Finally add transitions $p \overset{\$}{\rightarrow} q^{\mathsf{fin}}$ and $q^{\mathsf{in}} \overset{x}{\rightarrow} p$ for each $x \in X$.

Observe that $D(M_{X,\mathcal{F}})$ will have states $\{\{q^{\mathsf{in}}\}, \{q^{\mathsf{fin}}\}\} \cup \{S_x \mid x \in X\}$ where $S_x = \{F \in \mathcal{F} \mid x \in F\} \cup \{p\}$, and $\Delta(\{q^{\mathsf{in}}\}, x) = S_x$. Also, the time to compute $D(M_{X,\mathcal{F}})$ is bounded by $O(|X||\mathcal{F}|)$. Then any k-sized state cover $Q = \{F_1, ..., F_k\}$ is also a set cover. □

B.7 Algorithm

Our parsing algorithm produces a bit-coded parse tree from an input string w for a given RE E. We will simulate M_E in lock-step, reading a symbol from w in each step. The simulation maintains a set of all partial paths that read the prefix of w that has been consumed so far; there are always only finitely many paths to consider, since we restrict ourselves to paths without non-productive loops. When a path reaches a non-deterministic choice, it will "fork" into two paths with the same prefix. Thus, the path set can be represented as a tree of states, where the root is the initial state, the edges are transitions between states, and the leaves are the reachable states.

Definition 17 (Path trees). A *path tree* is a rooted, ordered, binary tree with *internal nodes* of outdegrees 1 or 2. Nodes are labeled by aNFA-states and edges by $\Gamma = \Sigma \cup \{0, 1\} \cup \{\epsilon\}$. Binary nodes have a pair of 0- and 1-labeled edges (in this order only), respectively.

We use the following notation:

- $\mathsf{root}(T)$ is the root node of path tree T.
- $\mathsf{path}(n, c)$ is the path from n to c, where c is a descendant of n.
- $\mathsf{init}(T)$ is the path from the root to the first binary node reachable or to the unique leaf of T if it has no binary node.
- $\mathsf{leaves}(T)$ is the *ordered list* of leaf nodes.
- $\mathsf{Tr}_{\mathsf{empty}}$ is the empty tree.

As a notational convenience, the tree with a root node labeled q and no children is written $q\langle\cdot\rangle$, where q is an aNFA-state. Similarly, a tree with a root labeled q with children l and r is written $q\langle 0 : l, 1 : r\rangle$, where q is an aNFA-state and l and r are path trees and the edges from q to l and r are labeled 0 and 1, respectively. Unary nodes are labelled by $\Sigma \cup \{\epsilon\}$ and are written $q\langle \ell : c\rangle$, denoting a tree rooted at q with only one ℓ-labelled child c.

In the following we shall use T_w to refer to a path tree created after processing input word w and T to refer to path trees in general, where the input string giving rise to the tree is irrelevant.

Definition 18 (Path tree invariant)**.** Let T_w be a path tree and w a word. Define $I(T_w)$ as the proposition that *all* of the following hold:

(i) The $\mathsf{leaves}(T_w)$ have pairwise distinct node labels; all labels are *symbol sources*, that is states with a single symbol transition, or the accept state.

(ii) All paths from the root to a leaf read w:

$$\forall n \in \mathsf{leaves}(T_w).\ \mathsf{read}(\mathsf{path}(\mathsf{root}(T_w), n)) = w.$$

(iii) For each leaf $n \in \mathsf{leaves}(T_w)$ there exists $w'' \in C_E(w)$ such that the bit-coded parse of w'' starts with $\mathsf{write}(\mathsf{path}(\mathsf{root}(T_w), n))$.

(iv) For each $w'' \in C_E(w)$ there exists $n \in \mathsf{leaves}(T_w)$ such that the bit-coded parse of w'' starts with $\mathsf{write}(\mathsf{path}(\mathsf{root}(T_w), n))$.

The path tree invariant is maintained by Algorithm 2: line 2 establishes part (i); line 3 establishes part (ii); and lines 4–7 establishes part (iii) and (iv).

Algorithm 1 Optimally streaming parsing algorithm.

Input: An aNFA M, a coverage relation $\sqsupseteq$, and an input stream S.
Output: Greedy leftmost parse tree, emitted in optimally-streaming fashion.

```
function STREAM-PARSE($M$, $\sqsupseteq$, $S$)
    $w \leftarrow \epsilon$
    $(T_\epsilon, \_\_) \leftarrow$ CLOSURE$(M, \emptyset, q^{\mathsf{in}})$   ▷ Initial path tree as output of CLOSURE
    while $S$ has another input symbol $a$ do
        if $C_E(wa) = \emptyset$ then
            return $\mathsf{write}(\mathsf{init}(T_w))$ followed by $\sharp$ and exit.
        $T_{wa} \leftarrow$ ESTABLISH-INVARIANT$(T_w, a, \sqsupseteq)$
        Output new bits on the path to the first binary node in $T_{wa}$, if any.
        $w \leftarrow wa$
    if $q^{\mathsf{fin}} \in \mathsf{leaves}(T_w)$ then
        return $\mathsf{write}(\mathsf{path}(\mathsf{root}(T_w), q^{\mathsf{fin}}))$
    else
        return $\mathsf{write}(\mathsf{init}(T_w))$ followed by $\sharp$
```

Theorem B.7.1 (Optimal streaming property)**.** *Assume extended w, $C_E(w) \neq \emptyset$. Consider the path tree T_w after reading w upon entry into the while-loop of the algorithm in Algorithm 1. Then* $\mathsf{write}(\mathsf{init}(T_w)) = O_E(w)$.

Algorithm 2 Establishing invariant $I(T_{wa})$

Input: A path tree T_w satisfying $I(T_w)$, a character a, and coverage relation $\sqsupseteq$.
Output: A path tree T_{wa} satisfying invariant $I(T_{wa})$.

```
function ESTABLISH-INVARIANT($T_w$, $a$, $\sqsupseteq$)
    Remove leaves from $T_w$ that do not have a transition on $a$.
    Extend $T_w$ to $T_{wa}$ by following all $a$-transitions.
    for each leaf $n$ in $T_{wa}$ do
        $(T', \_\_) \leftarrow$ CLOSURE$(M, \emptyset, n)$.
        Replace the leaf $n$ with the tree $T'$ in $T_{wa}$.
    return PRUNE$(T_{wa}, \sqsupseteq)$
```

Algorithm 3 Pruning algorithm.

Input: A path tree T and a covering relation $\sqsupseteq$.
Output: A pruned path tree T' where all leaves are alive.

```
function PRUNE$(T, \sqsupseteq)$
    for each $l$ in reverse(leaves$(T)$) do
        $S \leftarrow \{n \mid n$ comes before $l$ in leaves$(T)\}$
        if $S \sqsupseteq l$ then
            $p \leftarrow$ parent$(l)$
            Delete $l$ from $T$
            $T \leftarrow$ CUT$(T, p)$
    return $T$
function CUT$(T, n)$                                  ▷ Cuts a chain of 1-ary nodes.
    if |children$(n)$| $= 0$ then
        $p \leftarrow$ parent$(n)$
        $T' \leftarrow T$ with $n$ removed
        return CUT$(T', p)$
    else
        return $T$
```

In other words, the initial path from the root of T_w to the first binary node in T_w is the longest common prefix of all paths accepting an extension of w. Operationally, whenever that path gets longer by pruning branches, we output the bits on the extension.

Proof. Assume w extended, that is $w = \hat{w}$; assume $C_E(w) \neq \emptyset$, that is there exists w'' such that $w \sqsubseteq w''$ and $w'' \in \mathcal{L}[\![E]\!]$.

Claim: $|\mathsf{leaves}(T_w)| \geq 2$ or the unique node in $\mathsf{leaves}(T_w)$ is labeled by the accept state. Proof of claim: Assume otherwise, that is $|\mathsf{leaves}(T_w)| = 1$, but its node is not the accept state. By (i) of $I(T_w)$, this means the node must have a symbol transition on some symbol a. In this case, all accepting paths

Algorithm 4 ϵ-closure with path tree construction.

```
Require: An aNFA $M$, a set of visited states $V$, and a state $q$
Ensure: A path tree $T$ and a set of visited states $V'$
function CLOSURE($M, V, q$)
    if $q \xrightarrow{0} q_l$ and $q \xrightarrow{1} q_r$ then
        $(T^l, V_l) \leftarrow$ CLOSURE($M, V \cup \{q\}, q_l$)        ▷ Try left option first.
        $(T^r, V_{lr}) \leftarrow$ CLOSURE($M, V_l, q_r$)        ▷ Use $V_l$ to skip already-visited nodes.
        return $(q\langle T^l : T^r \rangle, V_{lr})$
    if $q \xrightarrow{\epsilon} p$ then
        if $p \in V$ then        ▷ Stop loops.
            return $(\mathsf{Tr}_{\mathsf{empty}}, V)$
        else
            $(T', V') \leftarrow$ CLOSURE($M, V \cup \{q\}, p$)
            return $(q\langle \epsilon : T' \rangle, V')$
    else        ▷ $q$ is a symbol source or the final state.
        return $(q\langle \cdot \rangle, V)$
```

$C_E(wa) = C_E(w)$ and thus $\hat{w} = \widehat{wa}$; in particular $\hat{w} \neq w$, which, however, is a contradiction to the assumption that w is extended.

This means we have two cases. The case $|\mathsf{leaves}(T_w)| = 1$ with the sole node being labeled by the accept state is easy: It spells a single path from initial to accept state. By (ii) and (iii) of $I(T_w)$ we have that that path is correct for w. By (iv) and since the accept state has no outgoing transitions, we have $C_E(w) = \{w\}$, and the theorem follows for this case.

Let us consider the case $|\mathsf{leaves}(T_w)| \geq 2$ then. Recall that $C_E(w) \neq \emptyset$ by assumption. By (iv) of $I(T_w)$ the accepting path of every $w'' \in C_E(w)$ starts with $\mathsf{path}(\mathsf{root}(T_w), n)$ for some $n \in \mathsf{leaves}(T_w)$, and by (iii) each path from the root to a leaf is the start of some accept path. Since $|\mathsf{leaves}(T_w)| \geq 2$ we know that there exists a binary node in T_w. Consider the first on the path from the root to a leaf. It has both 0- and 1-labeled out-edges. Thus the longest common prefix of $\{\mathsf{write}(p) \mid n \in \mathsf{leaves}(T_w), p \in \mathsf{path}(\mathsf{root}(T_w), n)\}$ is $\mathsf{write}(\mathsf{init}(T_w))$, the bits on the initial path from the root of T_w to its first binary node. □

The algorithm, as given, is only optimally streaming for extended prefixes. It can be made to work for all prefixes by enclosing it in an outer loop that for each prefix w computes $\hat{w}$ and calls the given algorithm with $\hat{w}$. The outer loop then checks that subsequent symbols match until $\hat{w}$ is reached. By Proposition B.5.1 the resulting algorithm gives the right result for all input prefixes, not only extended ones.

Theorem B.7.2. *The optimally streaming algorithm can be implemented to run in time $O(2^{m \log m} + mn)$, where $m = |E|$ and $n = |w|$.*

Sketch. As shown in Section B.6, we can decide coverage in time $O(m2^{O(m)})$. The set of ordered lists $\mathsf{leaves}(T)$ for any T reachable from the initial state can be precomputed and covered states marked in it. (This requires unit-cost random access since there are $O(2^{m \log m})$ such lists.) The ϵ-closure can be computed in time $O(m)$ for each input symbol, and pruning can be amortized over ϵ-closure computation by charging each edge removed to its addition to a tree path. □

For fixed regular expression E this is linear time in n and thus asymptotically optimal. An exponential in m as an additive preprocessing cost appears practically unavoidable since we require the coverage relation, which is inherently hard to compute (Proposition B.6.1).

B.8 Example

Consider the RE $(\mathtt{aaa} + \mathtt{aa})^\star$. A simplified version of its symmetric position automaton is shown in Figure B.2. The following two observations are requirements for an earliest parse of this expression:

- After one a has been read, the algorithm *must* output a 0 to indicate that one iteration of the Kleene star has been made, but:
- *five* consecutive as determine that the leftmost possibility in the Kleene star choice was taken, meaning that the first *three* as are consumed in that branch.

The first point can be seen by noting that any parse of a non-zero number of as must follow a path through the Kleene star. This guarantees that *if* a successful parse is eventually performed, it must be the case that at least one iteration was made.

The second point can be seen by considering the situation where only four input as have been read: It is not known whether these are the only four or more input symbols in the stream. In the former case, the correct (and only) parse is two iterations with the right alternative, but in the latter case, the first three symbols are consumed in the left branch instead.

These observations correspond intuitively to what "earliest" parsing is; as soon as it is impossible that an iteration was *not* made, a bit indicating this fact is emitted, and as soon as the first three symbols must have been parsed in the left alternative, this fact is output. Furthermore, a 0-bit is emitted to indicate that (at least) another iteration is performed.

Figure B.2 shows the evolution of the path tree during execution with the RE $(\mathtt{aaa} + \mathtt{aa})^\star$ on the input aaaaa.

By similar reasoning as above, after five as it is safe to commit to the left alternative after every third a. Hence, for the inputs $\mathtt{aaaaa}(\mathtt{aaa})^n$, $\mathtt{aaaaa}(\mathtt{aaa})^n\mathtt{a}$, and $\mathtt{aaaaa}(\mathtt{aaa})^n\mathtt{aa}$ the "commit points" are placed as follows ($\cdot$ indicate end-of-input):

$$\underset{0}{\mathtt{a}} \mid \underset{00}{\mathtt{aaaa}} \mid \underbrace{\left(\underset{00}{\mathtt{aaa}} \mid \cdots \mid \underset{00}{\mathtt{aaa}}\right)}_{n \text{ times}} \mid \underset{11}{\cdot} \qquad \underset{0}{\mathtt{a}} \mid \underset{00}{\mathtt{aaaa}} \mid \underbrace{\left(\underset{00}{\mathtt{aaa}} \mid \cdots \mid \underset{00}{\mathtt{aaa}}\right)}_{n \text{ times}} \mid \underset{01}{\mathtt{a}\cdot}$$

$$\underset{0}{\mathtt{a}} \mid \underset{00}{\mathtt{aaaa}} \mid \underbrace{\left(\underset{00}{\mathtt{aaa}} \mid \cdots \mid \underset{00}{\mathtt{aaa}}\right)}_{n \text{ times}} \mid \underset{1011}{\mathtt{aa}\cdot}$$

Complex coverage. The previous example does not exhibit any non-trivial coverage, i.e., situations where a state n is covered by $k > 1$ other states. One can construct an expression that contains non-trivial coverage relations by observing that if each symbol source s in the aNFA is associated with the RE representing the language recognized from s, coverage can be expressed as a set of (in)equations in Kleene algebra. Thus, the coverage $\{n_0, n_1\} \sqsupseteq n$ becomes $RE(n_0) + RE(n_1) \geq RE(n)$ in KA, where $RE(\cdot)$ is the function that yields the RE from a symbol source in an aNFA.

Any expression of the form $x_1zy_1 + x_2zy_2 + x_3z(y_1 + y_2)$ satisfies the property that two subterms cover a third. If the coverage is to play a role in the algorithm, however, the languages denoted by x_1 and x_2 must not subsume that of x_3, otherwise the part starting with x_3 would never play a role due to greedy leftmost disambiguation.

Choose $x_1 = x_2 = (\mathtt{aa})^\star$, $x_3 = \mathtt{a}^\star$, $y_1 = \mathtt{a}$, and $y_2 = \mathtt{b}$. Figure B.3 shows the expression

$$(\mathtt{aa})^\star\mathtt{za} + \mathtt{aa}^\star\mathtt{zb} + \mathtt{a}^\star\mathtt{za} + \mathtt{b} = (\mathtt{aa})^\star(\mathtt{za} + \mathtt{zb}) + a^\star\mathtt{z}(\mathtt{a} + \mathtt{b}).$$

The earliest point where any bits can be output is when the z is reached. Then it becomes known whether there was an even or odd number of as. Due to the coverage $\{8, 13\} \sqsupseteq 20$ state 20 is pruned away on the input aazb, thereby causing the path tree to have a large trunk that can be output.

CSV files. The expression $((\mathtt{a} + \mathtt{b})^\star(;(\mathtt{a} + \mathtt{b})^\star)^\star\mathtt{n})^\star$ defines the format of a simple semicolon-delimited data format, with data consisting of words over $\{\mathtt{a}, \mathtt{b}\}$ and rows separated by the newline character, n. Our algorithm emits the partial parse trees after each letter has been parsed, as illustrated on the example input below:

```
a;ba;a
b;;b
```

$$\underset{000}{\mathtt{a}} \mid \underset{10}{;} \mid \underset{01}{\mathtt{b}} \mid \underset{00}{\mathtt{a}} \mid \underset{10}{;} \mid \underset{00}{\mathtt{a}} \mid \underset{11}{\mathtt{n}} \mid \underset{001}{\mathtt{b}} \mid \underset{10}{;} \mid \underset{10}{;} \mid \underset{00}{\mathtt{a}} \mid \underset{11}{\mathtt{n}} \mid \underset{1}{\cdot}$$

Due to the star-height of three, many widespread implementations would not be able to meaningfully handle this expression using only the RE engine. Capturing groups under Kleene stars return either the first or last match, but

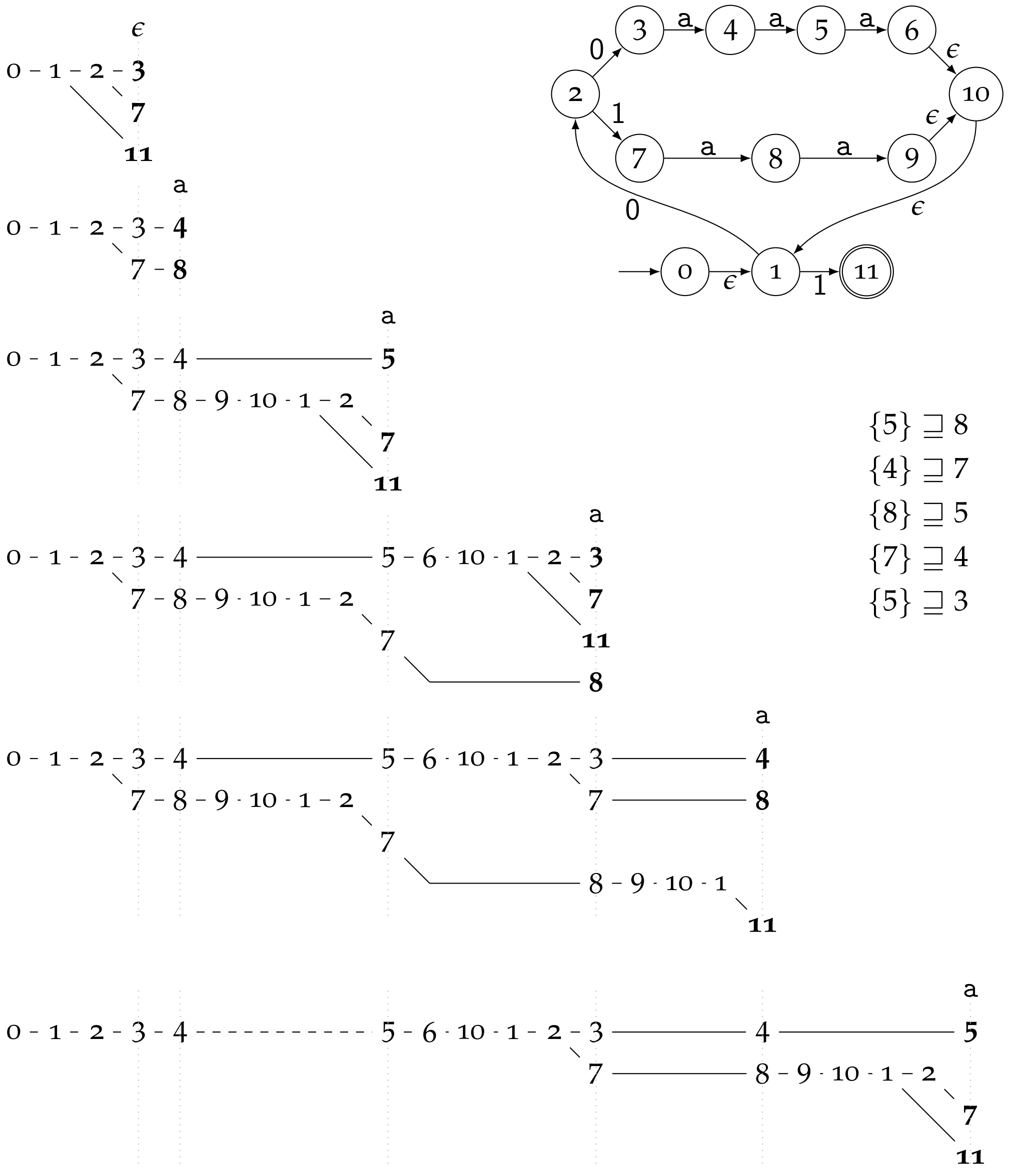


Figure B.2: Example run of the algorithm on the regular expression $E = (\mathtt{aaa} + \mathtt{aa})^\star$ and the input string aaaaa. The dashed edges represent the partial parse trees that can be emitted: thus, after one a we can emit a 0, and after five as we can emit 00 because the bottom "leg" of the tree has been removed in the pruning step. The automaton for E and its associated minimal covering relation are shown in the inset.

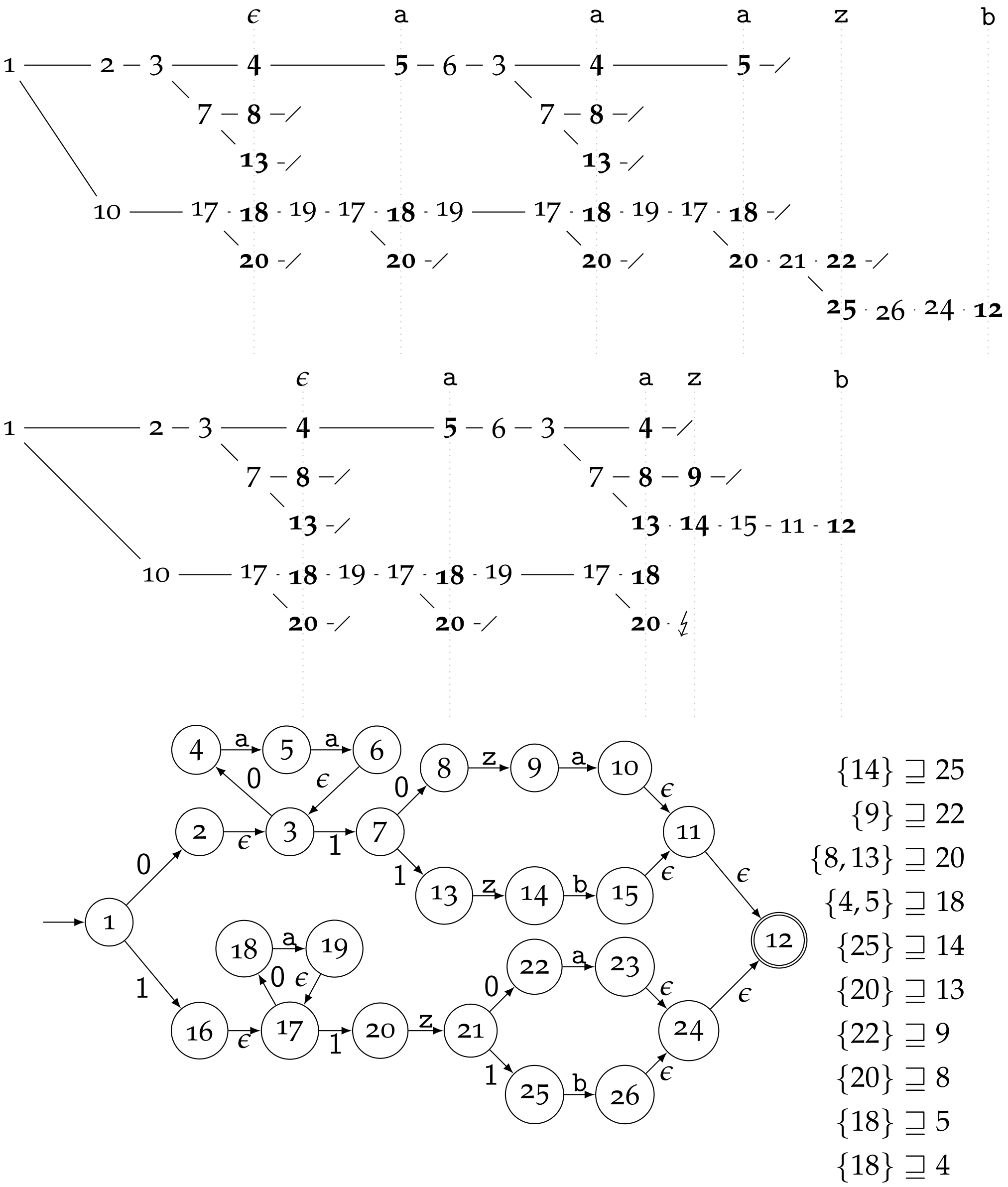


Figure B.3: Example run of the algorithm on $E = (\mathtt{aa})^\star(\mathtt{za}+\mathtt{zb}) + \mathtt{a}^\star\mathtt{z}(\mathtt{a}+\mathtt{b})$. Note that state 20 is covered by *the combination of* states 8 and 13. The earliest time the algorithm can commit is when a z is encountered, which determines if the number of as is even or odd. The top shows the path tree on the input aaazb. There is a "trunk" from state 1 to state 21 after reading z, as the rest of the branches have been pruned (not shown). This path corresponds to choosing the right top-level alternative. In the second figure, we see that if the z appears after an even number of as, a binary-node-free path from 1 to 7 emerges. Due to the cover $\{8,13\} \sqsupseteq 20$, the branch starting from 20 is not expanded further, even though there could be a z-transition on it. This is indicated with $\lightning$. Overall, the resulting parse tree corresponds to the leftmost option in the sum.

not a *list* of matches—and certainly not a list of lists of matches! Hence, if using an implementation like Perl's [16], one is forced to rewrite the expression by removing the iteration in the outer Kleene star and reintroduce it as a looping construct in Perl.

B.9 Related and Future Work

Parsing regular expressions is not new [6, 5, 3, 10, 14], and streaming parsing of XML documents has been investigated for more than a decade in the context of XQuery and XPath—see, e.g., [2, 7, 17]. However, *streaming regular expression* parsing appears to be new.

In earlier work [6] we described a compact "lean log" format for storing intermediate information required for two-phase regular expression parsing. The algorithm presented here may degenerate to two passes, but requires often just one pass in the sense being effectively streaming, using only $O(m)$ work space, independent of n. The preprocessing of the regular expression and the intermediate data structure during input string processing are more complex, however. It may be possible to merge the two approaches using a tree of lean log frames with associated counters, observing that edges in the path tree that are *not* labeled 0 or 1 are redundant. This is future work.

Bibliography

[1] A. Brüggemann-Klein and D. Wood. One-unambiguous regular languages. *Information and computation*, 140(2):229–253, 1998.

[2] D. Debarbieux, O. Gauwin, J. Niehren, T. Sebastian, and M. Zergaoui. Early nested word automata for XPath query answering on XML streams. In S. Konstantinidis, editor, *Implementation and Application of Automata*, volume 7982 of *Lecture Notes in Computer Science*, pages 292–305. Springer Berlin Heidelberg, 2013.

[3] D. Dubé and M. Feeley. Efficiently Building a Parse Tree From a Regular Expression. *Acta Informatica*, 37(2):121–144, 2000.

[4] G. Fowler. An interpretation of the POSIX regex standard. `http://www2.research.att.com/~astopen/testregex/re-interpretation.html`, January 2003. Inaccessible as of September 2016. Copies are provided upon request to the author of this dissertation.

[5] 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 (LNCS)*, pages 618–629, Turku, Finland, July 2004. Springer.

[6] 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.

[7] A. K. Gupta and D. Suciu. Stream processing of XPath queries with predicates. In *Proc. 2003 ACM SIGMOD International Conference on Management of Data*, SIGMOD '03, pages 419–430, New York, NY, USA, 2003. ACM.

[8] IEEE Computer Society. *Standard for Information Technology - Portable Operating System Interface (POSIX), Base Specifications, Issue 7*. IEEE, 2008. IEEE Std 1003.1.

[9] S. Kearns. Extending regular expressions with context operators and parse extraction. *Software - Practice and Experience*, 21(8):787–804, 1991.

[10] L. Nielsen and F. Henglein. Bit-coded Regular Expression Parsing. In *Proc. 5th Int'l Conf. on Language and Automata Theory and Applications (LATA)*, volume 6638 of *Lecture Notes in Computer Science (LNCS)*, pages 402–413. Springer, May 2011.

[11] S. Okui and T. Suzuki. Disambiguation in regular expression matching via position automata with augmented transitions. In M. Domaratzki and K. Salomaa, editors, *Implementation and Application of Automata*, volume 6482 of *Lecture Notes in Computer Science*, pages 231–240. Springer Berlin Heidelberg, 2011.

[12] L. J. Stockmeyer and A. R. Meyer. Word problems requiring exponential time (preliminary report). In *Proc. Fifth Annual ACM Symposium on Theory of Computing*, pages 1–9. ACM, 1973.

[13] M. Sulzmann and K. Z. M. Lu. Regular expression sub-matching using partial derivatives. In *Proc. 14th symposium on Principles and practice of declarative programming*, PPDP '12, pages 79–90, New York, NY, USA, 2012. ACM.

[14] 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.

[15] K. Thompson. Programming techniques: Regular expression search algorithm. *Commun. ACM*, 11(6):419–422, 1968.

[16] L. Wall, T. Christiansen, and J. Orwant. *Programming Perl*. O'Reilly Media, Incorporated, 2000.

[17] X. Wu and D. Theodoratos. A survey on XML streaming evaluation techniques. *The VLDB Journal*, 22(2):177–202, Apr. 2013.

Paper C

Kleenex: High-Performance Grammar Based Stream Processing

Bjørn Bugge Grathwohl, Fritz Henglein, Ulrik Terp Rasmussen, Kristoffer Aalund Søholm and Sebastian Paaske Tørholm. "Kleenex: Compiling Nondeterministic Transducers to Deterministic Streaming Transducers". In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. Pages 284-297. ACM, 2016. DOI: 10.1145/2837614.2837647.

The enclosed paper has been renamed. It has also been reformatted to fit the layout of this dissertation. A confusing typo have been corrected in Section C.4 ($|u|$ was consistently used instead of the correct $\overline{u}$).

Kleenex: High-Performance Grammar Based Stream Processing[1]

Niels Bjørn Bugge Grathwohl*, Fritz Henglein*, Ulrik Terp Rasmussen*,
Kristoffer Aalund Søholm† and Sebastian Paaske Tørholm†

*Department of Computer Science, University of Copenhagen (DIKU)

†Jobindex, Denmark

Abstract

We present and illustrate Kleenex, a language for expressing general nondeterministic finite transducers, and its novel compilation to streaming string transducers with worst-case linear-time performance and sustained high throughput. Its underlying theory is based on transducer decomposition into oracle and action machines: the oracle machine performs streaming greedy disambiguation of the input; the action machine performs the output actions. In use cases Kleenex achieves consistently high throughput rates around the 1 Gbps range on stock hardware. It performs well, especially in complex use cases, in comparison to both specialized and related tools such as `awk`, `sed`, RE2, Ragel and regular-expression libraries.

C.1 Introduction

A Kleenex program consists of a context-free grammar, restricted to guarantee regularity, with embedded side-effecting semantic actions.

We illustrate Kleenex by an example. Consider a large text file containing unbounded numerals, which we want to make more readable by inserting separators; e.g. “12742” is to be replaced by “12,742”). In Kleenex, this transformation can be specified as follows:

```
main  := (num /[^0-9]/ | other)*
num   := digit{1,3} ("," digit{3})*
digit := /[0-9]/
other := /./
```

This is the complete program. The program defines a set of nonterminals, with `main` being the start symbol. The constructs `/[0-9]/`, `/[^0-9]/` and `/./` specify matching a single digit, any non-digit and any symbol, respectively, and echoing the matched symbol to the output. The construct `","` reads nothing and outputs a single comma. The star * performs the inner transformation

[1] The order of authors is insignificant.

zero or more times; the repetition `{1,3}` performs it between 1 and 3 times. Finally, the `|` operator denotes prioritized choice, with priority given to the left alternative. An example of its execution is as follows:

Input read so far	…and output produced so far
Surf	Surf
Surface:␣	Surface:␣
Surface:␣14479	Surface:␣
Surface:␣1447985	Surface:␣
Surface:␣144798500␣	Surface:␣144,798,500␣
Surface:␣144798500␣km^2	Surface:␣144,798,500␣km^2

The example highlights the following:

Ambiguity by design. Any string is *accepted* by this program, since any string matching `num /[^0-9]/` also matches `(other)*`. Greedy disambiguation forces the `num /[^0-9]/` transformation to be tried first, however, and only if that fails do we fall back to echoing the input verbatim to the output using `other`.

Streaming output. The program almost always detects the earliest possible time an output action can be performed. Any non-digit symbol is written to the output immediately, and as soon as the first non-digit symbol after a sequence of digits is read, the resulting numeral with separators is written to the output stream. The first of a sequence of digits is not output right away, however. Employing a strategy that *always* outputs as early as possible would require solving a PSPACE-hard problem.

A Kleenex program is first compiled to a possibly ambiguous *(finite-state) transducer*. Any transducer can be decomposed into two transducers: an *oracle machine*, which maps an input string to a bit-coded representation of the transducer paths accepting the input, and a deterministic *action machine*, which translates such a bit-code to the corresponding sequence of output actions in the original transducer. The greedy leftmost path in the oracle machine corresponds to the lexicographically least bit-code of paths accepting a given input; consequently, disambiguation reduces to computing this bit-code for a given input. To compute it, the oracle machine is simulated in a streaming fashion. This generalizes NFA simulation to not just yield a single-bit output—accept or reject—but also the lexicographically least path witnessing acceptance. The simulation algorithm maintains a *path tree* from the initial state to all the oracle machine states reachable by the input prefix read so far. A branching node represents both sides of an alternative where both are still viable. The output actions on the (possibly empty) path segment from the initial state to the first branching node can be performed 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 algorithm generalizes *greedy regular expression parsing* [31, 32] to *arbitrary* right-regular grammars. Regular expressions correspond to certain well-structured oracle machines via their McNaughton-Yamada-Thompson construction. The simulation algorithm *automatically* results in constant memory space consumption for grammars that are deterministic modulo finite lookahead, e.g. one-unambiguous regular expressions [19]. For arbitrary transducers the simulation requires linear space in the size of the input in the worst case. No algorithm can guarantee constant space consumption: the number of unique path trees computed by the streaming algorithm is potentially unbounded due to the possibility of arbitrarily much lookahead required to determine which of two possible alternatives will eventually succeed. Unbounded lookahead is the reason that not all unambiguous transducers can be determinized to a finite state machine [53, 13].

By identifying path trees with the same ordered leaves and underlying branching structure, we obtain an equivalence relation with finite index. That is, a path tree can be seen as a rooted full binary tree together with an association of output strings with tree edges, and the set of reachable rooted full binary trees of an oracle machine can can be precomputed analogous to the NFA state sets reachable in an NFA. We can thus compile an oracle machine to a *streaming string transducer* [5, 4, 7], a deterministic machine model with (unbounded sized) string registers and affine (copy-free) updates associated with each transition: a path tree is represented as an abstract state and the contents of a finite set of registers, each containing a bit sequence coding a path segment of the represented path tree. Upon reading an input, the state is changed and the registers are updated in-place to represent the subsequent path tree. This yields a both asymptotically and practically very efficient implementation: the example shown earlier compiles to an efficient C program that operates with sustained high throughput in the 1 Gbps range on stock desktop hardware.

The semantic model of context-free grammars with unbridled "regular" ambiguity and embedded semantic actions is flexible and the above implementation technology is quite general. For example, the action transducer is not constrained to producing output in the string monoid, but can be extended to any monoid. By considering the monoid of affine register updates, Kleenex can code all nondeterministic streaming string transducers [8].

Contributions

This paper makes the following novel contributions:

- A *streaming* algorithm for *nondeterministic finite state transducers (FST)*, which emits the lexicographically least output sequence generated by all accepting paths of an input string based on decomposition into an input-processing *oracle machine* and an output-effecting *action machine*. It runs in $O(mn)$ time for transducers of size m and inputs of size n.

- An effective determinization of FSTs into a subclass of *streaming string transducers (SST)* [4], finite state machines with copy-free updating of string registers when entering a new state upon reading an input symbol.
- An expressive declarative language, *Kleenex*, for specifying FSTs 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 nondeterminism and its effective determinization and compilation to SSTs, which both highlights and complements the significance of SSTs as a deterministic machine model.
- An implementation, including empirically evaluated optimizations, of Kleenex that generates SSTs and deterministic finite-state machines, each rendered as standard single-threaded C-code that is eventually compiled to x86 machine code. The optimizations illustrate the design and implementation flexibility obtained by the underlying theories of FSTs and SSTs.
- Use cases that illustrate the expressive power of Kleenex, and a performance comparison with related tools, including Ragel [65], RE2 [62] 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 less expressive tools with special-cased algorithms and to tools with no or limited support for nondeterminism.

Overview of paper

In Section C.2 we introduce normalized transducers with explicit deterministic and nondeterministic ϵ-transitions. Kleenex and its translation to such transducers is defined in Section C.3. We then devise an efficient streaming transducer simulation (Section C.4) and its determinization (Section C.5) to streaming string transducers. In Section C.6 we briefly describe the compilation to C-code and some optimizations, and we then empirically evaluate the implementation on a number of simple benchmarks and more realistic use cases (Section C.7). We conclude with a discussion of related and possible future work (Section C.8).

[2] This avoids the $\Omega(M(n))$ lower bound for context-free grammar parsing, where $M(n)$ is the complexity of multiplying $n \times n$ matrices [40].

We assume basic knowledge of automata [39], compilation [2], and algorithms [21]. Basic results in these areas are not explicitly cited.

C.2 Transducers

An *alphabet* A is a finite set; e.g. the binary alphabet $\mathbf{2} = \{0,1\}$ and the empty alphabet $\emptyset = \{\}$. A^* denotes the free monoid generated by A, that is the strings over A with concatenation, expressed by juxtaposition, and the empty string ε as neutral element. We write $A[x,\ldots]$ for extending A with additional elements $x,\ldots$ not in A.

Definition 19 (Finite state transducer). A *finite state transducer (FST)* $\mathcal{T}$ over Σ and Γ is a tuple $(\Sigma, \Gamma, Q, q^-, q^f, E)$ where

- Σ and Γ are alphabets;
- Q is a finite set of *states*;
- $q^-, q^f \in Q$ are the *initial* and *final* states, respectively;
- $E : Q \times \Sigma[\epsilon] \times \Gamma[\epsilon] \times Q$ is the *transition relation*.

Its *size* is the cardinality of its transition relation: $|T| = |E|$.

$\mathcal{T}$ is *deterministic* if for all $q \in Q, a \in \Sigma[\epsilon]$ we have

$$\begin{aligned}(q,a,b',q') \in E \wedge (q,a,b'',q'') \in E \quad &\Rightarrow \quad b' = b'' \wedge q' = q''\\ (q,\epsilon,b',q') \in E \wedge (q,a,b'',q'') \in E \quad &\Rightarrow \quad \epsilon = a\end{aligned}$$

The *support* of a state is the set of symbols it has transitions on:

$$\mathrm{supp}(q) = \{a \in \Sigma[\epsilon] \mid \exists q', b.\, (q,a,b,q') \in E\}.$$

Deterministic FSTs with no ϵ-transitions and $\mathrm{supp}(q) = \Sigma$ for all q are *Mealy machines*. Conversely, every deterministic FST is easily turned into a Mealy machine by adding a failure state and transitions to it.

We write $q \xrightarrow{a/b} q'$ whenever $(q,a,b,q') \in E$, and E is understood from the context. A *path* in $\mathcal{T}$ is a possibly empty sequence of transitions

$$q_0 \xrightarrow{a_1/b_1} q_1 \xrightarrow{a_2/b_2} \ldots \xrightarrow{a_n/b_n} q_n$$

It has *input* $u = a_1 a_2 \ldots a_n$ and *output* $v = b_1 b_2 \ldots b_n$. We write $q_0 \xrightarrow{u/v} q_n$ if there exists such a path.

Definition 20 (Relational semantics, input language). FST $\mathcal{T}$ denotes the binary relation

$$\mathcal{R}[\![\mathcal{T}]\!] = \{(\overline{u}, \overline{v}) \mid q^- \xrightarrow{u/v} q^f\}$$

where the *ϵ-erasure* $\overline{\cdot} : \Sigma[\epsilon]^* \to \Sigma^*$ is $\overline{\epsilon} = \varepsilon$ and $\overline{a} = a$ for all $a \in \Sigma$, extended homomorphically to strings. Its *input language* is

$$\mathcal{L}[\![\mathcal{T}]\!] = \{s \mid \exists t \,.\, (s,t) \in \mathcal{R}[\![\mathcal{T}]\!]\}.$$

Two FSTs are *equivalent* if they have the same relational semantics.

The class of relations denotable by FSTs are the *rational relations*; their input languages are the *regular languages* [13].

Definition 21 (Normalized FST)**.** A *normalized finite state transducer* over Σ and Γ is a deterministic FST over $\Sigma[\epsilon_0, \epsilon_1]$ and Γ such that for all $q \in Q$, q is:

- a *choice state*: $\mathrm{supp}(q) = \{\epsilon_0, \epsilon_1\}$ and $q \neq q^f$, or
- a *skip state*: $\mathrm{supp}(q) = \{\epsilon\}$ and $q \neq q^f$, or
- a *symbol state*: $\mathrm{supp}(q) = \{a\}$ for some $a \in \Sigma$ and $q \neq q^f$, or
- the *final state*: $\mathrm{supp}(q) = \{\}$ and $q = q^f$

We say that q is a *resting state* if q is either a symbol state or the final state.

The relational semantics $\mathcal{R}[\![\mathcal{T}]\!]$ of a normalized FST is the same as in Definition 20, where ϵ-erasure is extended by $\overline{\epsilon}_0 = \overline{\epsilon}_1 = \varepsilon$.

Proposition C.2.1. *For every FST of size m there exists an equivalent normalized FST of size at most $3m$. Conversely, for every normalized FST of size m there exists an equivalent FST of the same size.*

Proof. (Sketch) For each state q with $k > 1$ outgoing transitions, add k new states $q^{(1)}, \ldots, q^{(k)}$, replace the i-th outgoing transition (q, a, b, q') by $(q^{(i)}, a, b, q')$ and add a full binary tree of ϵ_0- and ϵ_1-transitions for reaching each $q^{(i)}$ from q. In the converse direction, replace ϵ_0 and ϵ_1 by ϵ. □

Normalized FSTs are useful by limiting transition outdegree to 2, having explicit ϵ-transitions and classifying them into deterministic (ϵ) and ordered nondeterministic ones (ϵ_0, ϵ_1).

Proviso. Henceforth we will call normalized FSTs simply *transducers*.

Let $|\cdot| : \Sigma[\epsilon_0, \epsilon_1, \epsilon] \to \mathbf{2}[\epsilon]$ be defined by $|\epsilon_0| = 0$, $|\epsilon_1| = 1$ and $|a| = \epsilon$ for all $a \in \Sigma[\epsilon]$.

Definition 22 (Oracle and action machines)**.** Let $\mathcal{T}$ be a transducer. The *oracle machine* $\mathcal{T}^{\mathsf{C}}$ is defined as $\mathcal{T}$, but with each transition (q, a, b, q') replaced by $(q, a, |a|, q')$. Its *action machine* $\mathcal{T}^{\mathsf{A}}$ is $\mathcal{T}$, but with each transition (q, a, b, q') replaced by $(q, |a|, b, q')$.

The oracle machine is a transducer over Σ and $\mathbf{2}$; the action machine a deterministic FST over $\mathbf{2}$ and Γ. Each transducer can be canonically decomposed into its oracle and action machines:

Proposition C.2.2. $\mathcal{R}[\![\mathcal{T}]\!] = \mathcal{R}[\![\mathcal{T}^{\mathsf{A}}]\!] \circ \mathcal{R}[\![\mathcal{T}^{\mathsf{C}}]\!]$

```
main := (num /\n/)*
num := digit{1,3} ("," digit{3})*
digit := /a/
```

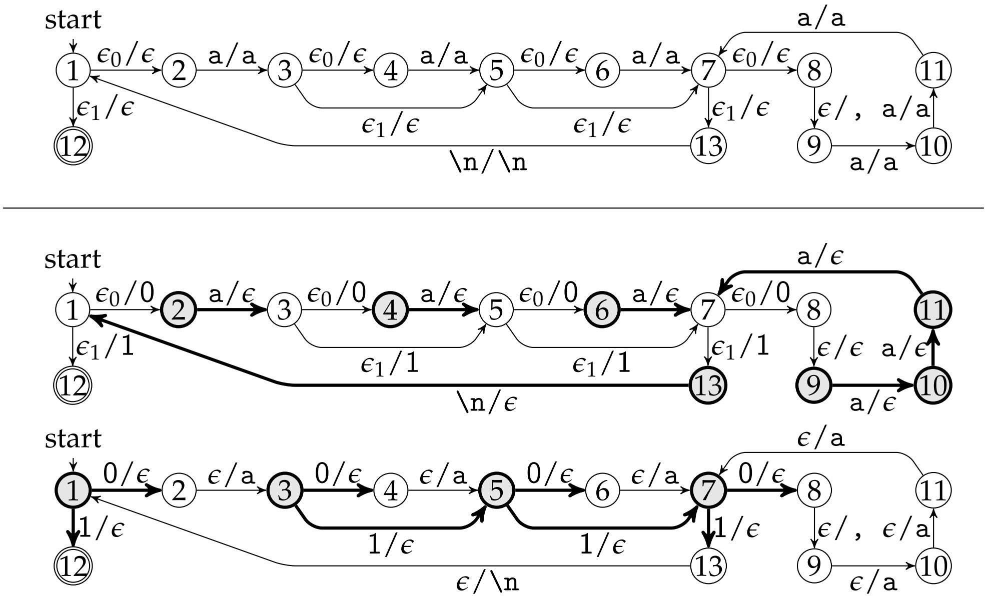


Figure C.1: Top: a Kleenex program and its associated transducer. The program accepts a list of newline-separated numbers (simplified to unary numbers with digit a) and inserts thousands separators. Bottom: The corresponding oracle and action machines.

where ∘ denotes relational composition. Note that the oracle machine is independent of the outputs in the original transducer; in particular, a transducer where only the outputs are changed has the same oracle machine. Intuitively, the action machine starts at the initial state the original transducer, automatically follows transitions from resting and skip states, and uses the bit string from the oracle machine as an oracle—hence the name—to choose which transition to take from a choice state; in this process it emits the outputs it traverses.

Example 1. Figure C.1 shows a Kleenex program (see Section C.3), the associated transducer and its decomposition into oracle and action machines.

Observe that if there is a path $q \xrightarrow{u/v} q'$ then u uniquely identifies the path from q to q' in a transducer and, furthermore, in an oracle machine so does v.

We write $q \xrightarrow{u/v}_{\mathsf{np}} q''$ if the path $q \xrightarrow{u/v} q''$ does not contain an *ϵ-loop*, that is a subpath $q' \xrightarrow{u'/v'} q'$ where $\overline{u'} = \varepsilon$. Paths without ϵ-loops are called *nonproblematic* paths [29].

Definition 23 (Greedy semantics)**.** The *greedy semantics* of a transducer T is $\mathcal{G}[\![\mathcal{T}]\!] = \mathcal{R}[\![\mathcal{T}^{\mathsf{A}}]\!] \circ \mathcal{G}[\![\mathcal{T}^{\mathsf{C}}]\!]$ where

$$\begin{aligned}\mathcal{G}[\![\mathcal{T}^{\mathsf{C}}]\!] \quad = \quad & \{(\overline{u},\overline{v}) \mid q^{-} \xrightarrow{u/v}_{\mathsf{np}} q^{f} \wedge \\ & \forall u',v'.\, q^{-} \xrightarrow{u'/v'}_{\mathsf{np}} q^{f} \wedge \overline{u} = \overline{u'} \Longrightarrow \overline{v} \leq \overline{v'}\}\end{aligned}$$

and $\leq$ denotes the lexicographic ordering on bit strings.

Given input string s, the greedy semantics chooses the lexicographically least path in the transducer accepting s and outputs the corresponding output symbols encountered along the path. The restriction to nonproblematic paths ensures that there are only finitely many paths accepting s and thus the lexicographically least amongst them exists, if s is accepted at all. We write $q \xrightarrow{u/v}_{\mathsf{min}} q'$ if $q \xrightarrow{u/v} q'$ is the lexicographically least nonproblematic path from q to q'.

A transducer $\mathcal{T}$ over Σ and Γ is *single-valued* if $\mathcal{R}[\![\mathcal{T}]\!]$ is a partial function from Σ^* to Γ^*.

Proposition C.2.3. *Let $\mathcal{T}$ be a transducer over Σ and Γ.*

- *$\mathcal{G}[\![\mathcal{T}]\!]$ is a partial function from Σ^* to Γ^*.*
- *$\mathcal{G}[\![\mathcal{T}]\!] = \mathcal{R}[\![\mathcal{T}]\!]$ if $\mathcal{T}$ is single-valued.*

The greedy semantics can be thought of as a *disambiguation policy* for transducers that conservatively extends the standard semantics for single-valued transducers to a deterministic semantics for arbitrary transducers.

C.3 Kleenex

Kleenex[3] is a language for compactly and conveniently expressing transducers.

Core Kleenex

Core Kleenex is a grammar for directly coding transducers.

Definition 24 (Core Kleenex syntax)**.** A *Core Kleenex* program is a nonempty list $p = d_0 d_1 \ldots d_n$ of *definitions* d_i, each of the form $N\!:=t$, where N is an identifier and t is generated by the grammar

$$t ::= \varepsilon \mid N \mid a\ N' \mid \text{"}b\text{"}\ N' \mid N_0 \,|\, N_1$$

[3] Kleenex is a contraction of *Kleene* and *expression* in recognition of the fundamental contributions by Stephen Kleene to language theory.

where $a \in \Sigma$ and $b \in \Gamma$ for given alphabets Σ, Γ, e.g. some character set. N ranges over some set of identifiers. The identifiers occurring in p are called *nonterminals*. There must be at most one definition of each nonterminal, and every occurrence of a nonterminal must have a definition.

Definition 25 (Core Kleenex transducer semantics)**.** The *transducer associated with Core Kleenex program p* for nonterminal $N \in \mathcal{N}$ is

$$\mathcal{T}_p(N) = (\Sigma, \Gamma, \mathcal{N}[q^f], N, q^f, E)$$

where $\mathcal{N}$ is the set of nonterminals in p, and E consists of transitions constructed from each production in p as follows:

$N{:=}\, \varepsilon$	$N \xrightarrow{\epsilon/\epsilon} q^f$
$N{:=}\, N'$	$N \xrightarrow{\epsilon/\epsilon} N'$
$N{:=}\, a\ N'$	$N \xrightarrow{a/\epsilon} N'$
$N{:=}$ "b" N'	$N \xrightarrow{\epsilon/b} N'$
$N{:=}\, N' \vert N''$	$N \xrightarrow{\epsilon_0/\epsilon} N'$ and $N \xrightarrow{\epsilon_1/\epsilon} N''$

The *semantics* of p is the greedy semantics of its associated transducer: $\mathcal{G}[\![p]\!] = \mathcal{G}[\![\mathcal{T}_p]\!](N_0)$ where N_0 is a designated start nonterminal. (By convention, this is `main`.)

Standard Kleenex

We extend the syntax of right-hand sides in Kleenex productions with arbitrary concatenations of the form and $N'N''$ and slightly simplify the remaining rules as follows:

$$t ::= \varepsilon \mid N \mid a \mid \texttt{"}b\texttt{"} \mid N_0 \,|\, N_1 \mid N'N''$$

Let p be such a *Standard Kleenex* program. Its *dependency graph* $G_p = (\mathcal{N}, D)$ consists of its nonterminals $\mathcal{N}$ and the *dependencies*

$$D = \{N \to N' \mid N' \text{ occurs in the definition of } N \text{ in } p\}.$$

Define the *strict dependencies* $D_s = \{N \to N' \mid (N{:=}N'N'') \in p\}$.

Definition 26 (Well-formedness)**.** A Standard Kleenex program p is *well-formed* if no strong component of G_p contains a strict dependency.

Well-formedness ensures that the underlying grammar is non-self-embedding [10], and thus its input language is regular.

Definition 27 (Kleenex syntax and semantics)**.** Let p be a well-formed Kleenex program with nonterminals $\mathcal{N}$. Define the transitions $E \subseteq \mathcal{N}^* \times \Sigma[\epsilon_0, \epsilon_1, \epsilon] \times \Gamma[\epsilon] \times \mathcal{N}^*$ as follows:

For rule d	add these transitions for all $X \in \mathcal{N}^*$ to E
$N{:=}\,\varepsilon$	$NX \xrightarrow{\epsilon/\epsilon} X$
$N{:=}\,N'$	$NX \xrightarrow{\epsilon/\epsilon} N'X$
$N{:=}\,a$	$NX \xrightarrow{a/\epsilon} X$
$N{:=}\,"b"$	$NX \xrightarrow{\epsilon/b} X$
$N{:=}\,N'\,N''$	$NX \xrightarrow{\epsilon/\epsilon} N'N''X$
$N{:=}\,N' \mid N''$	$NX \xrightarrow{\epsilon_0/\epsilon} N'X$ and $NX \xrightarrow{\epsilon_1/\epsilon} N''X$

Let $\text{Reach}(N) = \{\vec{N}_k \mid N \xrightarrow{\cdot/\cdot} \ldots \xrightarrow{\cdot/\cdot} \vec{N}_k\}$ be the nonterminal sequences reachable from N along transitions in E. The *transducer* $\mathcal{T}_p$ *associated with* p is $(\Sigma, \Gamma, R, N, \varepsilon, E|_R)$ where $R = \text{Reach}(N)$ for designated start symbol N and $E|_R$ is E restricted to R. The *(greedy) semantics* of p is the greedy semantics of $\mathcal{T}_p$: $\mathcal{G}[\![p]\!] = \mathcal{G}[\![\mathcal{T}_p]\!]$.

The following proposition justifies calling $\mathcal{T}_p$ a transducer.

Proposition C.3.1. *Let p be a well-formed Standard Kleenex program, with $\mathcal{T}_p$ as defined above. Then R is finite, and $\mathcal{T}_p$ is a transducer, that is normalized FST.*

Proof. (Sketch) $\text{Reach}(N)$ consists of all the nonterminal suffixes of sentential forms of left-most derivations of p considered as a context-free grammar. In well-formed Kleenex programs, their maximum length is bounded by $|\mathcal{N}|$. It is easy to check that every state in R is either a resting, skip, choice or final state. □

Observe that the transducer associated with a Kleenex program can be exponentially bigger than the program itself.

Since a transducer has a straightforward representation in Core Kleenex, the construction of $\mathcal{T}_p$ provides a translation of a well-formed Standard Kleenex program into Core Kleenex. For example, the Kleenex program on the left translates into the Core Kleenex program on the right:

$$\begin{array}{rcl} M & := & M' \mid N \\ M' & := & N N_a \\ N_a & := & a \\ N & := & N' \mid N_\varepsilon \\ N' & := & N_b N \\ N_b & := & b \\ N_\varepsilon & := & \varepsilon \end{array} \quad \Longrightarrow \quad \begin{array}{rcl} M & := & N' \mid N \\ M' & := & N' \mid N_a \\ N_a & := & a N_\varepsilon \\ N' & := & b M' \\ N & := & N' \mid N_\varepsilon \\ N_\varepsilon & := & \varepsilon \end{array}$$

The Full Surface Language

The full surface syntax of *Kleenex* is obtained by extending Standard Kleenex with the following term-level constructors, none of which increase the expressive power:

$$\begin{aligned} t ::= \ldots \mid \texttt{"}v\texttt{"} \mid \texttt{/}e\texttt{/} \mid \texttt{\textasciitilde}t \mid t_0 t_1 \mid t_0 \texttt{|} t_1 \mid t\texttt{*} \mid t\texttt{+} \mid t\texttt{?} \\ \mid t\texttt{\{}n\texttt{\}} \mid t\texttt{\{}n\texttt{,\}} \mid t\texttt{\{,}m\texttt{\}} \mid t\texttt{\{}n\texttt{,}m\texttt{\}} \end{aligned}$$

where $v \in \Gamma^*$, $n, m \in \mathbb{N}$, and e is a *regular expression*. The terms $t_0 t_1$ and $t_0 \texttt{|} t_1$ desugar into $N_0 N_1$ and $N_0 \texttt{|} N_1$, respectively, with additional productions $N_0 \texttt{ := } t_0$ and $N_1 \texttt{ := } t_1$ for new nonterminals N_0, N_1. The term $\texttt{"}v\texttt{"}$ is shorthand for a sequence of outputs.

Regular expressions are special versions of Kleenex terms without nonterminals. They desugar to terms that output the matched input string, i.e. $\texttt{/}e\texttt{/}$ desugars by adding an output symbol $\texttt{"}a\texttt{"}$ after every input symbol a in e. For example, the regular expression `/a*|b{n,m}|c?/` becomes

$$(a\texttt{"}a\texttt{"})\texttt{*}\texttt{|}(b\texttt{"}b\texttt{"})\texttt{\{}n\texttt{,}m\texttt{\}}\texttt{|}(c\texttt{"}c\texttt{"})\texttt{?},$$

which can then be further desugared.

A *suppressed* subterm $\texttt{\textasciitilde}t$ desugars into t with all output symbols removed, including any that might have been added in t by the above construction. For example, $\texttt{\textasciitilde}(\texttt{"}b\texttt{"}\texttt{/}a\texttt{/})$ desugars into $\texttt{\textasciitilde}(\texttt{"}b\texttt{"}\ a\ \texttt{"}a\texttt{"})$, which further desugars into a.

The operators $\cdot\texttt{*}$, $\cdot\texttt{+}$ and $\cdot\texttt{?}$ desugar to their usual meaning as regular operators, as do the repetition operators $\cdot\texttt{\{}n\texttt{\}}$, $\cdot\texttt{\{}n\texttt{,\}}$, $\cdot\texttt{\{,}m\texttt{\}}$, and $\cdot\texttt{\{}n\texttt{,}m\texttt{\}}$. Note that they all desugar into their *greedy* variants where matching a subexpression is preferred over skipping it. For example:

$$M \texttt{ := } (a\,\texttt{"}b\texttt{"})\texttt{+} \implies \begin{array}{lll} M & \texttt{:=} & (a\,\texttt{"}b\texttt{"})N' \\ N' & \texttt{:=} & a\,\texttt{"}b\texttt{"}N'\texttt{|}\varepsilon \end{array}$$

Lazy variants can be encoded by making ε the left rather than the right choice of an alternative.

Register Update Actions

By viewing Γ as an alphabet of *effects*, we can extend the expressivity of Kleenex beyond rational functions [13]. Let X be a computable set, and assume that there is an effective partial action $\Gamma \times X \rightarrow X$. It is simple to define a deterministic machine implementing the function $\Gamma^* \times X \rightarrow X$ by successively applying a list of actions to some starting state X. Any Kleenex program then denotes a function $\Sigma^* \times X \rightarrow X$ by composing its greedy semantics with such a machine. If we can implement the pure transducer part in a streaming fashion, then a state X can be maintained on-the-fly by interpreting output actions as soon as they become available.

Let $X = (\Gamma^*)^+ \times (\Gamma^*)^n$ for some n, representing a non-empty stack of output strings and n string registers. The transducer output alphabet is extended to $\Gamma[\mathsf{push}, \mathsf{pop}_0, ..., \mathsf{pop}_n, \mathsf{write}_0, ..., \mathsf{write}_n]$, with actions defined by

$$\begin{aligned}
(t\vec{w}, v_0, ..., v_n) \cdot a &= ((ta)\vec{w}, v_0, ..., v_n) && (a \in \Gamma)\\
(\vec{w}, v_0, ..., v_n) \cdot \mathsf{push} &= ((\varepsilon)\vec{w}, v_0, ..., v_n)\\
(t\vec{w}, v_0, ..., v_i, ..., v_n) \cdot \mathsf{pop}_i &= (\vec{w}, v_0, ..., t, ..., v_n) && (|\vec{w}| > 0)\\
(t\vec{w}, v_0, ..., v_n) \cdot \mathsf{write}_i &= ((tv_i)\vec{w}, v_0, ..., v_n)
\end{aligned}$$

The bottom stack element can only be appended to and models a designated output register—popping it is undefined. The stack and the variables can be used to perform complex string interpolation. To access the extended actions, we extend the surface language:

$$\begin{aligned}
t ::= \ldots \;&|\; R \,\texttt{@}\, t \;|\; \texttt{!}R\\
&|\; \texttt{[}\, R \,\texttt{<-}\, (R \mid \texttt{"}v\texttt{"})^\star \,\texttt{]} \;|\; \texttt{[}\, R \,\texttt{+=}\, (R \mid \texttt{"}v\texttt{"})^\star \,\texttt{]}
\end{aligned}$$

where R ranges over *register names* standing for indices.

The term $R \,\texttt{@}\, t$ desugars to $\texttt{"push"}\ t\ \texttt{"pop}_R\texttt{"}$, and the term $\texttt{!}R$ desugars to $\texttt{"write}_R\texttt{"}$. The term $\texttt{[}\, R \,\texttt{<-}\, x_1 ... x_m \,\texttt{]}$ desugars to $\texttt{"push"}t'_1 ... t'_m\texttt{"pop}_R\texttt{"}$, where $t'_i = \mathsf{write}_{R_i}$ if $x_i = R_i$, and $t'_i = x_i$ otherwise. Finally, $\texttt{[}\, R \,\texttt{+=}\, \vec{x} \,\texttt{]}$ desugars to $\texttt{[}\, R \,\texttt{<-}\, R\ \vec{x} \,\texttt{]}$.

Thus all streaming string transducers (see Section C.5) can be coded. 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/
```

where the first line above desugars to

$$\begin{aligned}
&\texttt{main := "push" line "pop}_a\texttt{" "push" line "pop}_b\texttt{"}\\
&\qquad\qquad\texttt{"write}_b\texttt{" "write}_a\texttt{"}
\end{aligned}$$

C.4 Streaming Simulation

As we have seen, every Kleenex program has an associated transducer, which can be split into oracle and action machines. The action machine is a straightforwardly implemented deterministic FST. The oracle machine is nondeterministic, however: The key challenge is how to (deterministically) find and output the lexicographically least path that accepts a given input string. In this section we develop an efficient oracle machine simulation algorithm that inputs a stream of symbols and streams the output bits almost as early as possible during input processing.

Path Trees

Given an oracle machine $\mathcal{T}^{\mathsf{C}}$ as in Definition 22, consider input s such that $q^- \xrightarrow{u/v}_{\mathsf{min}} q^f$ where $\overline{u} = s$. Recall that $q \xrightarrow{u/v}_{\mathsf{min}} q'$ uniquely identifies a path from q to q' in $\mathcal{T}^{\mathsf{C}}$, which is furthermore asserted to be the lexicographically minimal amongst all nonproblematic paths from q to q'.

Proposition C.4.1 (Path decomposition)**.** *Assume $q^- \xrightarrow{u/v}_{\mathsf{min}} q^f$. For every prefix s' of $\overline{u}$ there exist unique u', v', u'', v'', q' such that $q^- \xrightarrow{u'/v'}_{\mathsf{min}} q' \xrightarrow{u''/v''}_{\mathsf{min}} q^f$, q' is a resting state, $\overline{u'} = s'$, $u'u'' = u$ and $v'v'' = v$.*

Proof. Let u' be the longest prefix of u such that $\overline{u'} = s'$ and let $q^- \xrightarrow{u'/v'}_{\mathsf{np}} q'$ be the path from q determined by u'. (Such a prefix must exist.) Claim: This is the q' in the proposition.

1. q' is a resting state. If it were not, we could transition on ϵ, ϵ_0 or ϵ_1 resulting in a longer prefix w with $\overline{w} = s'$.

2. $q^- \xrightarrow{u'/v'}_{\mathsf{min}} q'$ and $q' \xrightarrow{u''/v''}_{\mathsf{min}} q^f$. If any of these subpaths were not lexicographically minimal, we could replace it with one that is lexicographically less, resulting in a path from q^- to q^f that is lexicographically less than $q^- \xrightarrow{u/v}_{\mathsf{np}} q^f$, contradicting our assumption $q^- \xrightarrow{u/v}_{\mathsf{min}} q^f$. □

After reading input prefix s' we need to find the above $q^- \xrightarrow{u'/v'}_{\mathsf{min}} q'$ where $\overline{u'} = s'$. Since we do not know the remaining input yet, however, we maintain *all* paths $q^- \xrightarrow{u'/v'}_{\mathsf{min}} q'$ for any resting state q' such that $\overline{u'} = s'$.

Definition 28 (Path tree)**.** Let $\mathcal{T}^{\mathsf{C}}$ be given. Its *path tree* $\mathrm{P}(s)$ for s is the set of paths $\{q^- \xrightarrow{u/v}_{\mathsf{min}} q' \mid \overline{u} = s\}$.

Consider a transducer as a directed labeled graph where the nodes are transducer states indexed by the strings reaching them,

$$\{q_s \mid \exists u, v. q^- \xrightarrow{u/v} q \wedge \overline{u} = s\},$$

and the edges are the corresponding transitions,

$$\{q_s \xrightarrow{a/b} q'_{s\overline{a}} \mid q \xrightarrow{a/b} q'\}.$$

It can be seen that $\mathrm{P}(s)$ is a subgraph that forms a non-full rooted edge-labeled binary tree. The *stem* of $\mathrm{P}(s)$ is the longest path in this tree from q^-_ε to some $q_{s'}$ for a prefix s' of s only involving nodes with at most one child. The *leaves* of $\mathrm{P}(s)$ are the states q such that q_s is reachable, in lexicographic order of the paths reaching them from q^-_ε.

Example 2. Recall the oracle machine for the decimal converter in the lower left of Figure C.1. Its path tree for input a is shown in the upper left of Figure C.2. The nodes are subscripted with the length of the input prefix rather the input prefix itself. Note that the leaf states are listed from top to bottom in lexicographic order of their paths reaching them. This means that the top state is the prime candidate for being q' in Proposition C.4.1. If the remainder of the input is not accepted from it, though, the other leaf states take over in the given order.

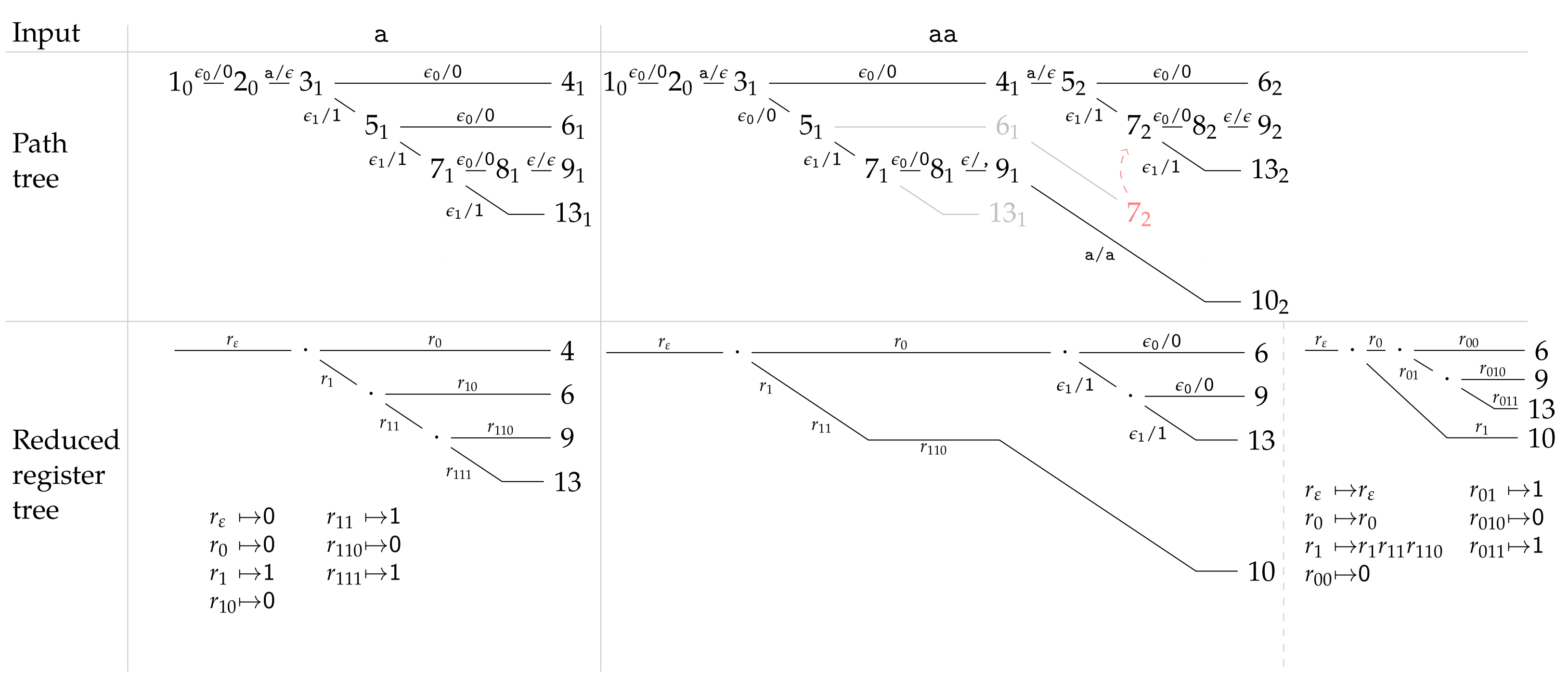


Figure C.2: Path trees for the decimal conversion oracle in Figure C.1. Above left: path tree reading a. Subscripts denote the number of input symbols read when the given state was visited. Above right: path tree reading aa. Failing paths are shown in gray. Below left: reduced register tree reading a, with register valuation. Below middle: extension of register tree after reading an additional a. Note mix of registers and bits, and that bottom branch is now labeled by a sequence of registers. Below right: the path tree and register *update* after reading aa. The registers r_1, r_{11}, and r_{110} are all on the same unary path and are concatenated.

Basic Simulation Algorithm

The basic streaming simulation algorithm works as follows:

Algorithm 5 Basic streaming algorithm

Let $s = a_1 \ldots a_n \in \Sigma^*$ be the input string.

```
for $i = 1$ to $n$ do
    if $P(a_1 ... a_i) = \emptyset$ then
        terminate with failure (input rejected)
    if stem$(P(a_1 ... a_i))$ longer than stem$(P(a_1 ... a_{i-1}))$ then
        emit the output bits on the stem extension
if $P(a_1 ... a_n)$ contains path to $q^f$ then
    if path tree contains at least one branch, emit output bits on path from
    highest binary ancestor to $q^f$
    terminate with success (input accepted)
else
    terminate with failure (input rejected)
```

The critical step in the algorithm is incrementally computing the path tree for $s'a$ from the path tree for s'.

Algorithm 6 Incremental path tree computation

Let P be $P(s')$ for some prefix s' of the input string, and let $[q_0, ..., q_n]$ be its leaves in lexicographic order of the paths reaching them. Upon reading a, incrementally compute $P(s'a)$ as follows.

```
for $q = q_0$ to $q_n$ do
    compute $P_q(a)$, the path tree of lexicographically least $(u/v)$ paths
    with $\overline{u} = a$ from $q$ to resting states, but excluding resting states that
    have been reached in a previous iteration
    if $P_q(a)$ is non-empty then
        replace leaf node $q$ in P by $P_q(a)$
    else
        prune branch from lowest binary ancestor to leaf node $q$; if binary
        ancestor does not exist, then terminate with failure (input rejected)
```

Example 3. The upper right in Figure C.2 shows P(aa) for the decimal converter. Observe how it arises from P(a) by extending leaf states 4 and 9, which have an a-transition, and building the ϵ-closure as a binary tree. It prunes branches either because they reach a state already reached by a lexicographical lower path (state 6) or because the leaf does not have transition on a (state 13). The algorithm outputs 0 after reading the first a since 0 is the sequence of output bits on the stem of the path tree. It does not output anything after reading the second a since $\mathrm{P}(aa)$ has the same stem as $\mathrm{P}(a)$.

Definition 29 (Optimal streaming)**.** Let f be a partial function from Σ^* to Γ^*, $s \in \Sigma^*$. Let $T(s) = \{f(ss') \mid s' \in \Sigma^* \wedge ss' \in \mathrm{dom} f\}$. The *output $f^\#(s)$ determined by f for s* is the longest common prefix of $T(s)$ if $T(s)$ is nonempty; otherwise it is undefined. The partial function $f^\#$ is called the *optimally streaming version* of f. An *optimally streaming algorithm* for f is an algorithm that implements $f^\#$: It emits output symbols as soon as they are semantically determined by the input prefix read so far.

Let transducer $\mathcal{T}$ be given. Write $\mathcal{L}[\![q]\!]$ for $\mathcal{L}[\![\mathcal{T}']\!]$ where $\mathcal{T}'$ is $\mathcal{T}$, but with q as initial state instead of q^-. A state q is *covered* by $\{q_1, \ldots, q_k\}$ if $\mathcal{L}[\![q]\!] \subseteq \mathcal{L}[\![q_1]\!] \cup \ldots \cup \mathcal{L}[\![q_k]\!]$. A path tree $\mathrm{P}(s)$ with lexicographically ordered leaves $[q_1, \ldots, q_n]$ is *cover-free* if no q_i is covered by $\{q_1, \ldots, q_{i-1}\}$. $\mathcal{T}$ is *cover-free* if $\mathrm{P}(s)$ is cover-free for all $s \in \Sigma^*$.

Theorem C.4.2. *Let $\mathcal{T}$ be cover-free. Then Algorithm 5 with Algorithm 6 for incremental path tree recomputation is an optimally streaming algorithm for $\mathcal{G}[\![\mathcal{T}^{\mathsf{C}}]\!]$ that runs in time $O(mn)$, where $m = |\mathcal{T}^{\mathsf{C}}|$ and n is the length of the input string.*

Proof. (Sketch) Algorithm 6 can be implemented to run in time $O(m)$ since it visits each transition in $\mathcal{T}^{\mathsf{C}}$ at most once and pruning can be amortized: every deallocation of an edge can be charged to its allocation. Algorithm 5 invokes Algorithm 6 n times. Optimal streaming follows from a generalization of the proof of optimal streaming for regular expression parsing [32]. □

The algorithm can be made optimally streaming for all oracle transducers by also pruning leaf states that are covered by other leaf states in Step 6 of Algorithm 6. Coverage is PSPACE-complete, however. Eliding the coverage check does not seem to make much of a difference to the streaming behavior in practice.

C.5 Determinization

NFA simulation maintains a set of NFA states. This is the basis of compiling an NFA into a DFA: precompute and number the set of *all* NFA state sets reachable by *any* input from the initial NFA state, observing that there are only finitely many such sets. In the transducer simulation in Section C.4 path trees play the role of NFA state sets. The corresponding determinization idea does not work for transducers, however: $\{\mathrm{P}(s) \mid s \in \Sigma^*\}$ is in general infinite. For example, for the oracle machine in Figure C.1, the trees $\mathrm{P}(\mathtt{a}^n)$ all have the same stem, but contain paths with bit strings of length proportional to n. This is inherently so. A single-valued transducer can be transformed effectively [12, 66] into a form of deterministic finite-state transducer if its relational semantics is *subsequential* [53, 13], but nondeterministic finite state transducers in general are properly more expressive than their deterministic counterparts. We can factor a path tree into its underlying full binary tree and the labels associated with the edges, though. Since there are only finitely many different such trees, we can achieve determinization to transducers with registers storing the potentially unbounded label data.

Definition 30 (Streaming String Transducer [4])**.** A deterministic *streaming string transducer* (SST) over alphabets Σ, Γ is a tuple $\mathcal{S} = (X, Q, q^-, F, \delta^1, \delta^2)$ where

- X is a finite set of *register variables;*
- Q is is a finite set of *states;*
- F is a partial function $Q \to (\Gamma \cup X)^*$ mapping each *final state* $q \in \mathrm{dom}(F)$ 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 $Q \times \Sigma \to Q$;
- δ^2 is a *register update* function $Q \times \Sigma \to (X \to (\Gamma \cup X)^*)$ such that for each $q \in Q$, $a \in \Sigma$ and $x \in X$, there is at most one occurrence of x in the multiset of strings $\{\delta^2(q,a)(y) \mid y \in X\}$.

A *configuration* of an SST $\mathcal{S} = (X, Q, q^-, F, \delta^1, \delta^2)$ is a pair (q, ρ) where $q \in Q$ is a state, and $\rho : X \to \Gamma^*$ is a *valuation*. A valuation extends to a monoid homomorphism $\widehat{\rho} : (X \cup \Gamma)^* \to \Gamma^*$ by setting $\rho(x) = x$ for $x \in \Gamma$. The initial configuration is (q^-, ρ^-) where $\rho^-(x) = \epsilon$ for all $x \in X$.

A configuration steps to a new one given an input symbol: $\delta((q,\rho),a) = (\delta^1(q,a), \widehat{\rho} \circ \delta^2(q,a))$. The transition function extends to a transition function on words δ^* by $\delta^*((q,\rho),\epsilon) = (q,\rho)$ and $\delta^*((q,\rho),au) = \delta^*(\delta((q,\rho),a),u)$.

Every SST $\mathcal{S}$ denotes a partial function $\mathcal{F}[\![\mathcal{S}]\!] : \Sigma^* \to \Gamma^*$ where for any $u \in \Sigma^*$ such that $\delta^*((q^-,\rho^-),u) = (q',\rho')$, we define

$$\mathcal{F}[\![\mathcal{S}]\!](u) = \begin{cases} \widehat{\rho'}(F(q')) & \text{if } q' \in \mathrm{dom}(F) \\ \text{undefined} & \text{otherwise} \end{cases}$$

In the following, let $X = \{r_p \mid p \in \mathbf{2}^*\}$ be a set of registers.

Definition 31 (Reduced register tree)**.** Let P be a path tree. Its *reduced register tree* $\mathcal{R}(\mathrm{P})$ is a pair $(R_{\mathrm{P}}, \rho_{\mathrm{P}})$ where ρ_{P} is a valuation $X \to \mathbf{2}^*$ and R_{P} is a full binary tree with state-labeled leaves, obtained from P by first contracting all unary branches and concatenating edge labels; then replacing each edge label (u/v) by a single register symbol r_p, where p denotes the unique path from the root to the edge destination node, and setting $\rho_{\mathrm{P}}(r_p) = v$.

The set $\{R_{\mathrm{P}(s)} \mid s \in \Sigma^*\}$ is finite: it is bounded by the number of full binary trees with up to $|Q|$ leaves times the number of possible permutations of the leaves.

Let R be R_{P} and $a \in \Sigma$ a symbol, and apply Algorithm 6 to R. The result is a non-full binary tree with edges labeled either by a register or by a (u/v) pair. By reducing the tree again and treating registers as output labels, we get a pair $(R_a, \kappa_{R,a})$ where $\kappa_{R,a} : X \to (\mathbf{2} \cup X)^*$ is a register update.

Example 4. Consider the bottom left tree in Figure C.2. This is the reduced register tree obtained from the path tree above it. The evaluation map ρ can be seen below it, where register subscripts denote their position in the register tree. In the middle is the result of extending the register tree using Algorithm 6. Reducing this again yields the tree on the right. The update map κ is shown below it—note that the range of this map is mixed register/bit sequences.

Proposition C.5.1. *Let $\mathcal{T}^{\mathsf{C}}$ be given, and let $\mathrm{P} = \mathrm{P}(s)$, $\mathrm{P}' = \mathrm{P}(sa)$, $(R, \rho) = \mathcal{R}(\mathrm{P})$ and $(R', \rho') = \mathcal{R}(\mathrm{P}')$ for some s and a. Then $R' = R_a$ and $\rho' = \widehat{\rho} \circ \kappa_{R,a}$.*

Theorem C.5.2. *Let $\mathcal{T}^{\mathsf{C}}$ be an oracle machine of size m. There is an SST $\mathcal{S}$ with $O(2^{m \log m})$ states such that $\mathcal{F}[\![\mathcal{S}]\!] = \mathcal{G}[\![\mathcal{T}]\!]$.*

Proof. Let $Q_{\mathcal{S}} = \{R_{\mathrm{P}(s)} \mid s \in \Sigma^*\} \cup \{R_0\}$ and $q^-_{\mathcal{S}} = R_0$, where R_0 is the single-leaf binary tree with leaf $q^-_{\mathcal{T}}$. The set of registers $X_{\mathcal{S}}$ is the finite subset of register variables occurring in $Q_{\mathcal{S}}$. The transition maps are given by $\delta^1_{\mathcal{S}}(R, a) = R_a$ and $\delta^2_{\mathcal{S}}(R, a) = \kappa_{R,a}$. For any $R \in Q_{\mathcal{S}} - \{R_0\}$, define the final output $F_{\mathcal{S}}(R)$ to be the sequence of registers on the path from the root to the final state $q^f_{\mathcal{T}}$ in R if R contains it as a leaf; otherwise let $F_{\mathcal{S}}(R)$ be undefined. Let $F_{\mathcal{S}}(R_0) = \overline{v}$ if $q^-_{\mathcal{T}} \xrightarrow{\epsilon/v}_{\mathsf{min}} q^f$ for some v; otherwise let $F_{\mathcal{S}}(R_0)$ be undefined.

Correctness follows by showing $\delta^*((R_0, \rho^-), u) = \mathcal{R}(\mathrm{P}(u))$ for all $u \in \Sigma^+$. We prove this by induction, applying Proposition C.5.1 in each step. For the case $u = \varepsilon$ correctness follows by the definition of $F_{\mathcal{S}}(R_0)$.

The upper bound follows from the fact that there are at most $C_{k-1}(k-1)! = O(2^{m \log m})$ full binary trees with k pairwise distinct leaves where k is the number of resting states in $\mathcal{T}^{\mathsf{C}}$ and C_{k-1} is the $(k-1)$-st *Catalan* number. □

Example 5. The oracle machine in Figure C.1 yields the SST in Figure C.3. The states 1 and 2 are identified by the left and right reduced trees, respectively, in the bottom of Figure C.2.

Corollary C.5.3. *The SST $\mathcal{S}$ for $\mathcal{T}^{\mathsf{C}}$ can be implemented to execute in time $O(mn)$ where $m = |\mathcal{T}^{\mathsf{C}}|$.*

Proof. (Sketch) Use a data structure for imperatively extending a string register, $r := rs$, in amortized time $O(n)$ where n is the size of s, independent of the size of the string stored in r. The result then follows from the fact that the steps in Algorithm 6 can be implemented in the same amortized time. □

In practice, the compiled version of the SST is much more efficient—roughly one to two orders of magnitude faster—than streaming simulation since it compiles away the interpretive overhead of explicitly managing the binary trees underlying path trees and employs machine word-level parallelism by operating on bit strings in fewer registers rather than many edges each labeled by at most one bit.

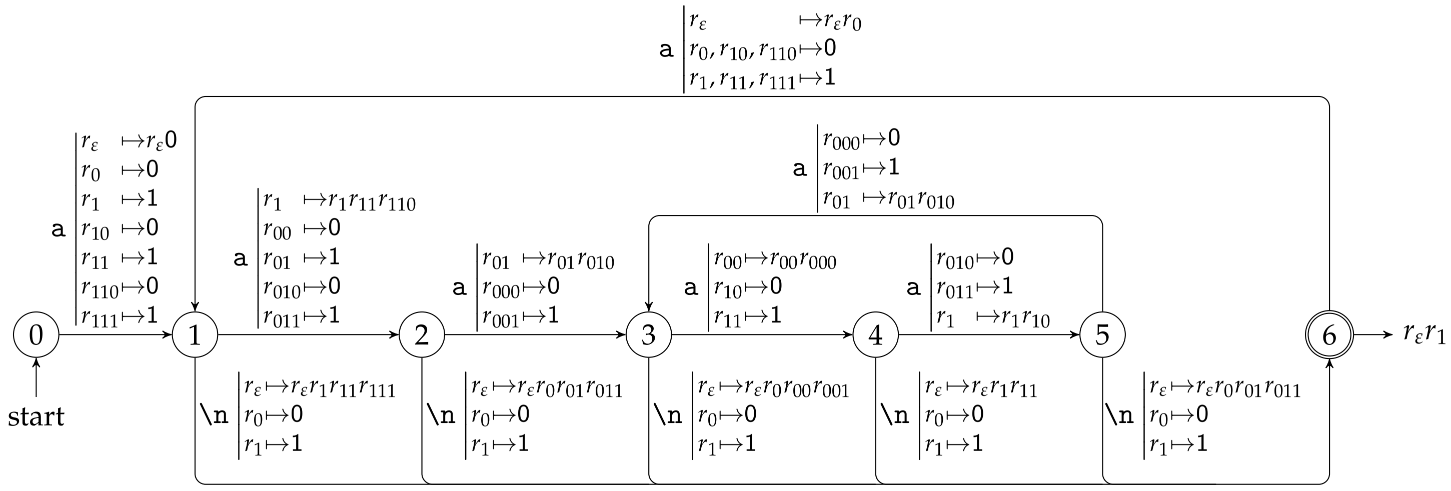


Figure C.3: SST constructed from the oracle machine in Figure C.1.

C.6 Implementation and Benchmarks

Our implementation[4] compiles the action machine and the oracle SST to machine code via C. We have implemented several optimizations which are orthogonal to the underlying principles behind our compilation from Kleenex via transducers to SSTs:

Inlining of output actions The action machine and the oracle SST need to be composed. We can do this at runtime by piping the SST output to the action machine, or we can apply a form of deforestation [70] to inline the output actions directly into the SST. This is straightforward since the machines are deterministic.

Constant propagation The SSTs generated by the construction underlying Theorem C.5.2 typically contain many constant-valued registers (e.g. most registers in Figure C.3 are constant). We eliminate these using constant propagation: compute reaching definitions by solving a set of data-flow constraints.

Symbolic representation A more succinct SST representation is obtained by using a symbolic representation of transitions where input symbols are replaced by *predicates* and output symbols by *terms* indexed by input symbols. This is a straightforward extension of similar representations for automata [72] and transducers [66, 68, 67, 69]. Our implementation uses simple predicates in the form of byte ranges, and simple output terms represented by byte-indexed lookup tables. We refer the reader to the cited literature for the technical details of symbolic transducers.

Finite lookahead Symbolic FSTs with bounded lookahead have been shown to reduce the state space when representing string encoders [22, 67, 69]. We have implemented a form of finite lookahead in our SST representation. Opportunities for lookahead is detected by the compiler, and arise in the case where the program contains a string constant with length above one. In this case a lookahead transition is used to check once and for all if the string constant is matched by the input instead of creating an SST state for each symbol. This may in some cases reduce the size of the generated code since we avoid tabulating all states of the whole program for every prefix of the string constant.

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

RE2, Google's regular expression C++ library [62].
RE2J, a recent re-implementation of RE2 in Java [63].

[4] Source code and benchmarks available at `http://kleenexlang.org/`

GNU `AWK` and GNU `sed`, programming languages and tools for text processing and extraction [60].

Origlib, a regular expression library written in C++ with support for different character encodings [38].

Ragel, a finite state machine compiler with multiple language backends [65].

In addition, we implemented test programs using the standard regular expression libraries in the scripting languages Perl [71], Python [41], and Tcl [73].

The benchmark suite, Kleenex programs, and version numbers of libraries used can be found at `http://kleenexlang.org`.

Meaning of plot labels Kleenex plot labels indicate the compilation path, and follow the format `[<0|3>[-la] | woACT] [clang|gcc]`. `0/3` indicates whether constant propagation was disabled/enabled. `la` 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 Section C.6. 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. 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.

Kleenex compilation timeout On some plots, some versions of the Kleenex programs are not included. This is because the C compiler times out (after 30 seconds). As we fully determinize the transducers, the resulting C code can explode in some cases. The two worst-case exponential blow-ups in generating transducers from Kleenex and then generating SSTs implemented in C code from transducers are *inherent*, though, and as such can be considered a *feature* of Kleenex: tools based on finite machines with no or limited nondeterminism support such as Ragel would require *hand-coding* a potentially huge machine that Kleenex generates *automatically*.[5]

Baseline

The following two 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 Figure C.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 Figure C.5 shows. Due to the high degree of interleaving and the lack of keywords, in this program the lookahead optimization has reduced 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 for us predictable way.

Rewriting

Thousand separators The following Kleenex program inserts thousand separators in a sequence of digits:

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

[5] We have found it excessively difficult to employ Ragel in some use cases with a natural nondeterministic specification.

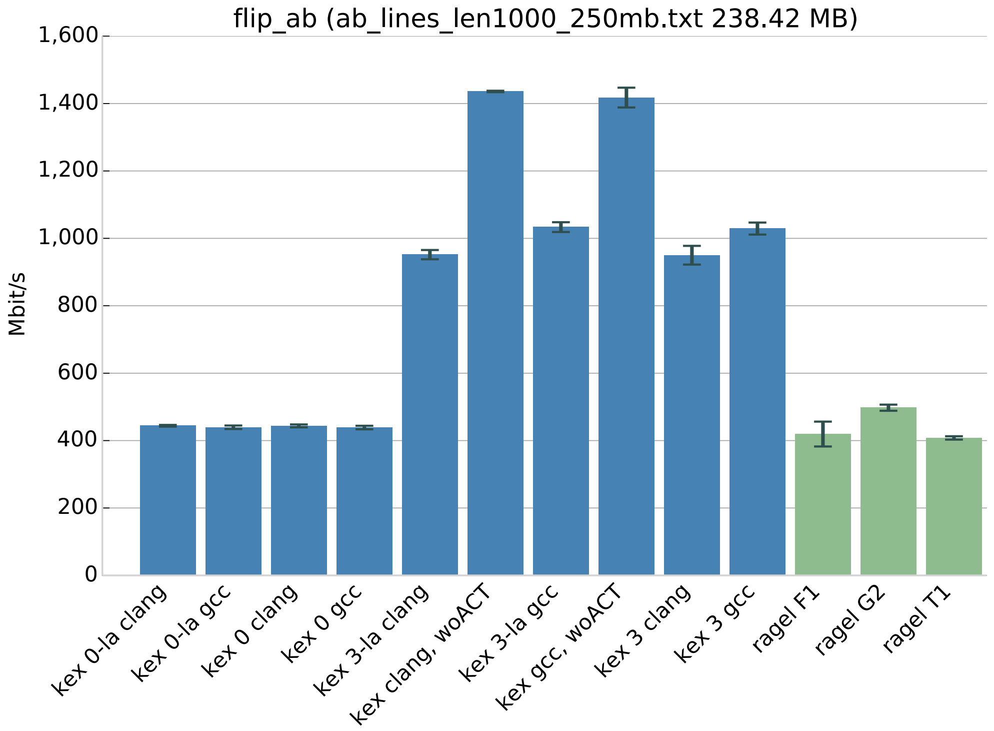


Figure C.4: `flip_ab` run on lines with average length 1000.

We evaluated the Kleenex implementation along with two other implementations using Perl and Python. The performance can be seen in Figure C.6. Both Perl and Python are significantly slower than all of the Kleenex implementations; the problem is tricky to solve with regular expressions unless one reads the input right-to-left.

IRC protocol handling The following Kleenex program parses the IRC protocol as specified in RFC 2812.[6] It follows roughly the output style described in part 2.3.1 of the RFC. Note that the Kleenex source code and the BNF grammar in the RFC are almost identical. Figure C.7 shows the throughput on 250 MiB 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
```

[6] https://tools.ietf.org/html/rfc2812

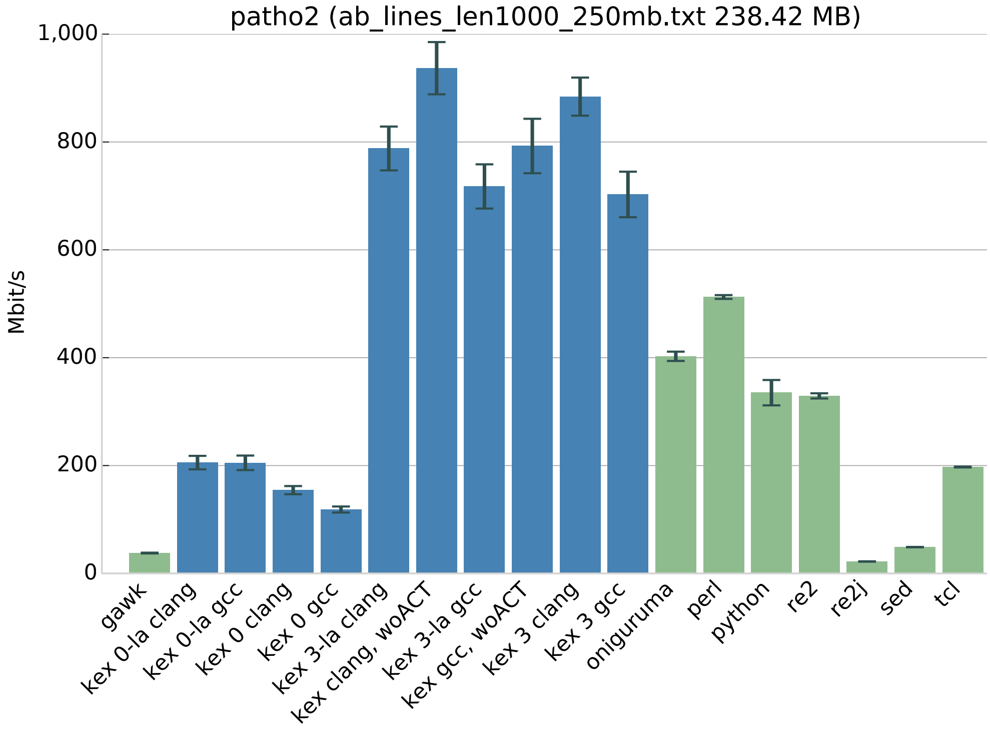


Figure C.5: `patho2` run on lines with average length 1000.

```
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}
```

CSV rewriting The program `csv_project3` deletes all columns but the 2nd and 5th from a CSV file:

```
main := (row /\n/)*
col  := /[^,\n]*/
row  := ~(col /,/) col "\t" ~/,/ ~(col /,/)
        ~(col /,/) col ~/,/      ~col
```

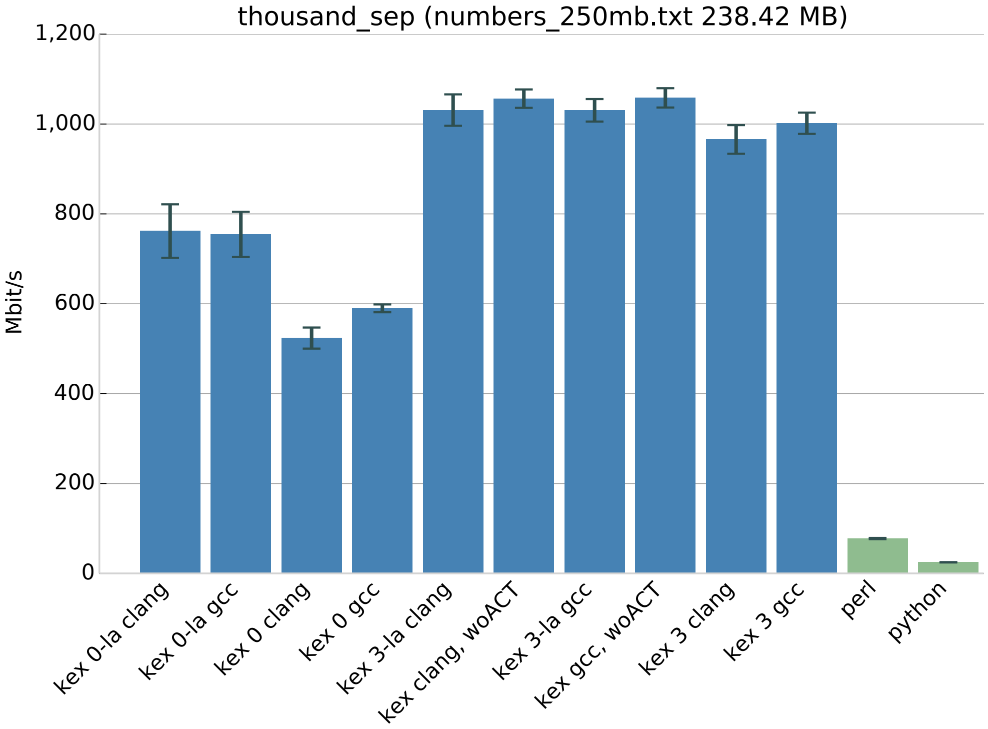


Figure C.6: Inserting separators in random numbers of average length 1000.

Various specialized tools that can handle this transformation are included in Figure C.8; 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 very fast for its own use case, a 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 automatically put in an output buffer which is flushed routinely, whereas a programmer has to manually handle buffering when writing a Ragel program.

With or Without Action Separation

One can choose to use the machine resulting from fusing the oracle and action machines when compiling Kleenex. Doing so results in only one process performing both disambiguation and outputting, which in some cases is faster and in other cases slower. Figures C.8, C.9, and C.11 illustrate both situations. It depends on the structure of the problem whether it pays off to split up the work into two processes; if all the work happens in the oracle machine and the action machine does nearly nothing, then the added overhead incurred by the process context switches becomes noticeable. On the other hand, in cases where both machines perform much work, the fact that two CPU cores can be

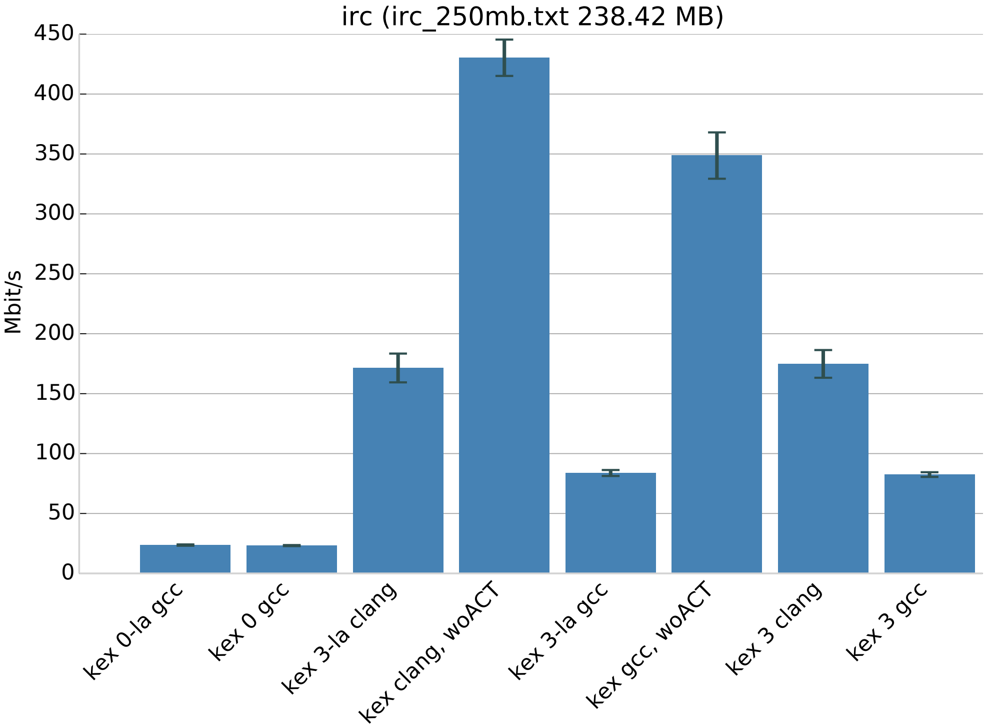


Figure C.7: Throughput when parsing 250 MiB random IRC data.

utilized in parallel speeds up execution. This is more likely once Kleenex has support for actions that can perform arbitrary computations, e.g. in the form of embedded C code.

C.7 Use Cases

We briefly touch upon various use cases—natural application scenarios—for Kleenex.

JSON logs to SQL We have implemented a Kleenex program that transforms a JSON log file into an SQL insert statement. The program works on the logs provided by Issuu.[7]

The Ragel version we implemented outperforms Kleenex by about 50% (Figure C.9), indicating that further optimizations of our SST construction should be possible.

[7] The line-based data set consists of 30 compressed parts; part one is available from `http://labs.issuu.com/anodataset/2014-03-1.json.xz`

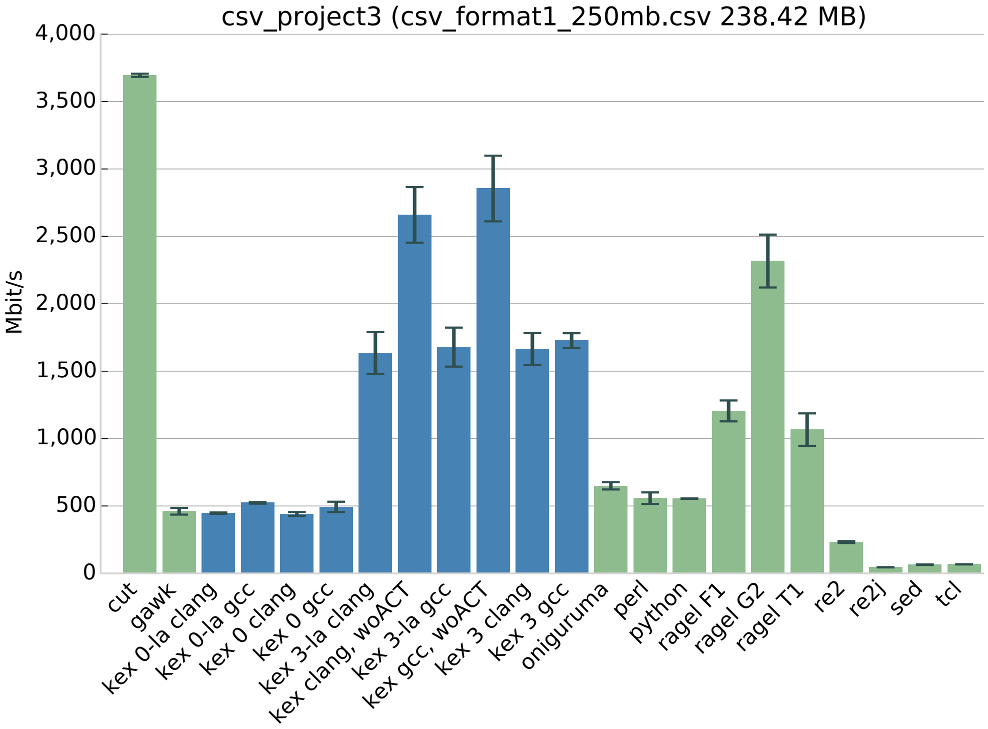


Figure C.8: `csv_project3` reads in a CSV file with six columns and outputs columns two and five. “gawk” is GNU AWK that uses the native AWK way of splitting up lines. “cut” is a tool from GNU coreutils that splits up lines.

Apache CLF to JSON The Kleenex program below rewrites Apache CLF[8] 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\":\"" /-/ "\""
authuser := "\"authuser\":\"" /[^ \n]+/ "\""
timestamp := "\"date\":\"" ~/\[/ /[^\n\]]+/ ~/]/ "\""
request := "\"request\":" quotedString
code := "\"status\":\"" integer "\""
bytes := "\"size\":\"" (integer | /-/) "\""
referer := "\"url\":" quotedString
useragent := "\"agent\":" quotedString
sep := "," ~/[\t ]+/
quotedString := /"([^"\n]|\\")*"/
```

[8] https://httpd.apache.org/docs/trunk/logs.html#common

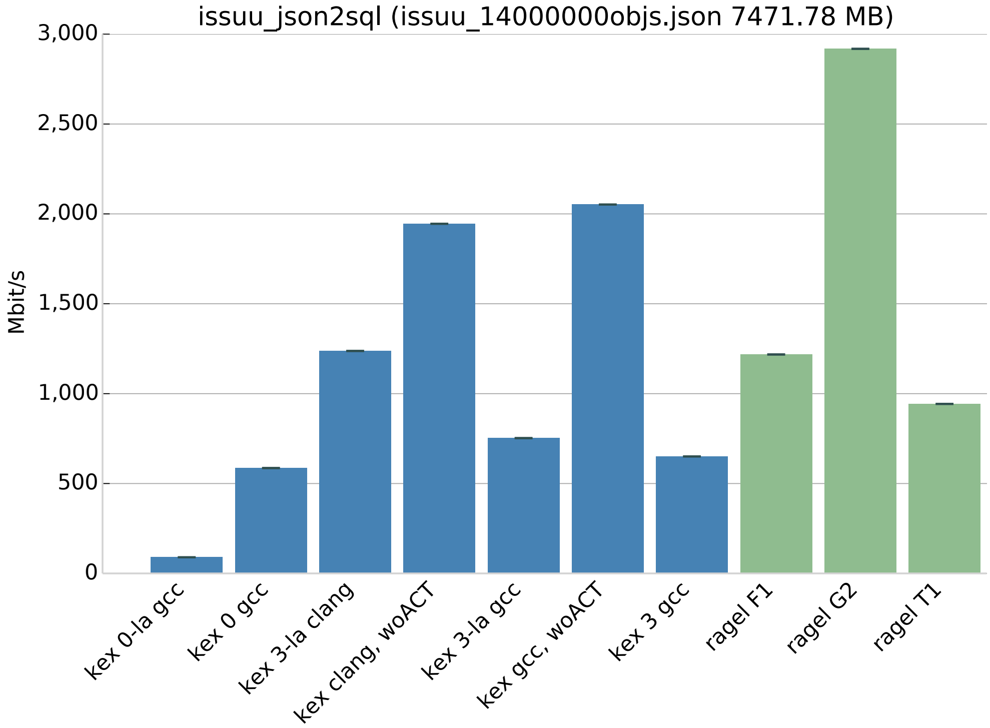


Figure C.9: The speeds of transforming JSON objects to SQL INSERT statements using Ragel and Kleenex.

```
integer := /[0-9]+/
ip := integer (/\./ integer){3}
```

This is a re-implementation of a Ragel program.[9] Figure C.10 shows the benchmark results. The versions compiled with clang are 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 [30], the program `iso_datetime_to_json` converts date and time stamps in an ISO standard format to a JSON object. Figure C.11 shows the performance.

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 /-->/
```

[9] https://engineering.emcien.com/2013/04/5-building-tokenizers-with-ragel

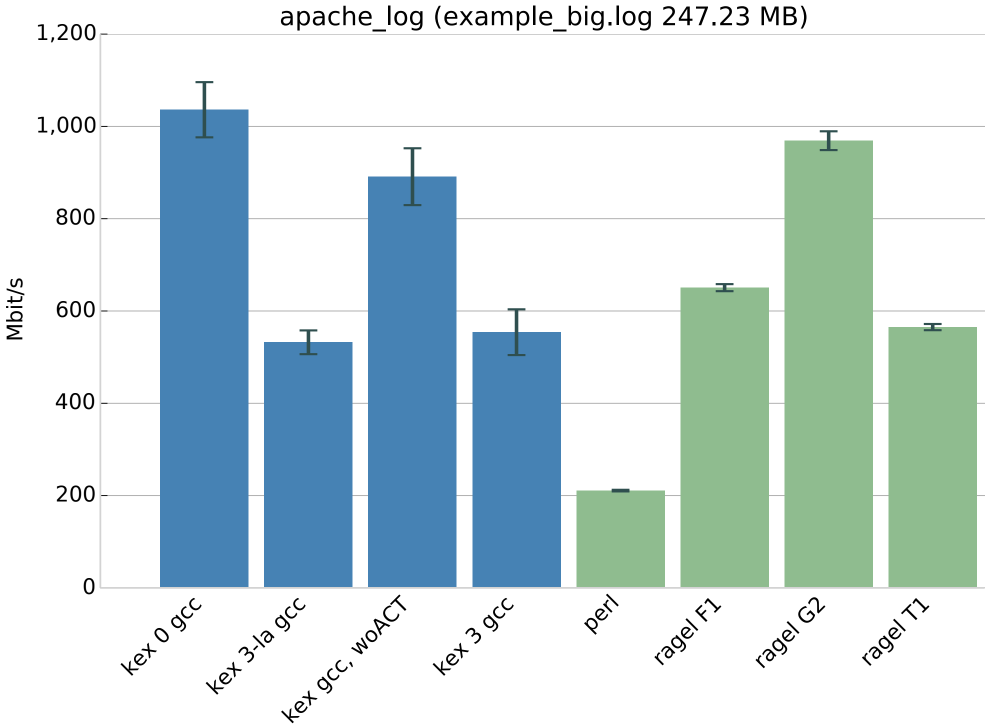


Figure C.10: Speed of the conversion from the Apache Common Log Format to JSON.

```
           "<div>" !render "</div>"
doc := ~/\*/ t@/[^*]*/ ~/\*/
       [ orig += "*" t "*" ] [ render += "<b>" t "</b>" ]
     | t@/./ [ orig += t ] [ render += t ]
clear := [ orig  <- "" ] [ render <- "" ]
```

Syntax highlighting Kleenex can be used to write syntax highlighters; in fact, the Kleenex syntax in this paper was highlighted using a Kleenex program.

C.8 Discussion

We discuss related and future work by building Kleenex conceptually up from regular expression matching via regular expressions as types for bit-coded parsing to transducers and eventually grammars with embedded actions.

Regular Expression Matching. Regular expression *matching* has different meanings in the literature.

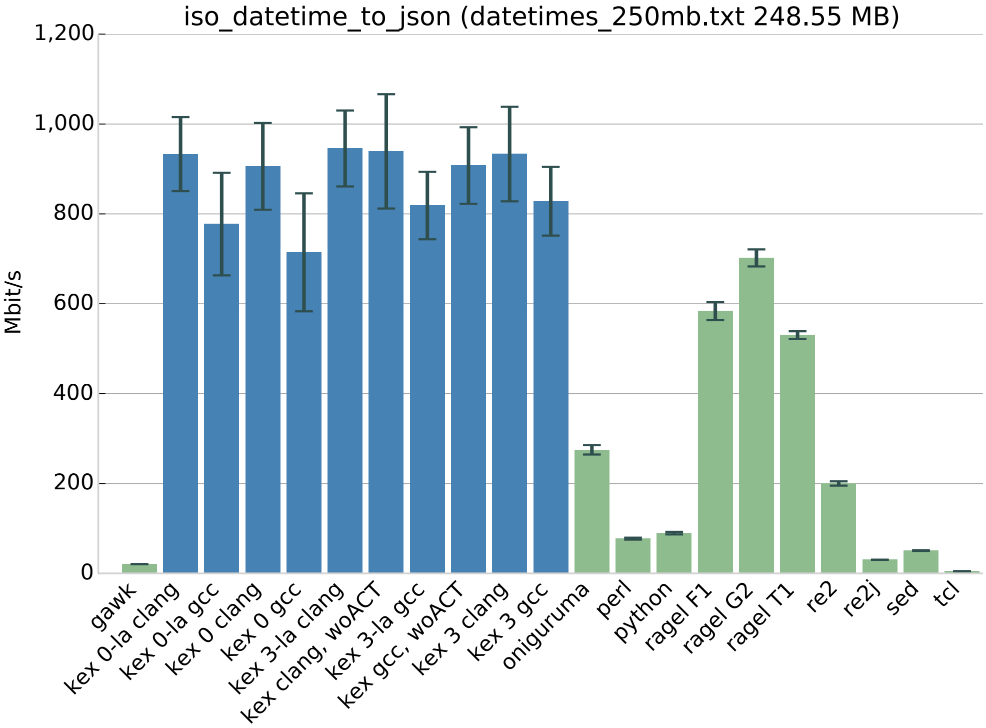


Figure C.11: The performance of the conversion of ISO time stamps into JSON format.

For *acceptance testing*, the subject of *automata theory* where only a single bit is output, NFA-simulation and DFA-construction are classical techniques. Bille and Thorup [14] improve on Myers' [46] log-factor improved classical NFA-simulation for regular expressions, based on tabling. They design an $O(kn)$ algorithm [15] with word-level parallelism, where $k \leq m$ is the 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 [34], an input is not only classified as accepting or not, but a substring is returned for each sub-RE of interest. Subgroup matching exposes ambiguity in the RE. Subgroup matching is often implemented by backtracking over alternatives, which implements *greedy* disambiguation.[10] Backtracking may result in exponential-time worst case behavior, however, even in the absence of inherently hard matching with backreferences [1]. Considerable human effort is usually expended to engineer REs used in practice to perform well anyway. More recently, REs designed to force exponential run-time behavior are used in algorithmic attacks, though [56,

[10] Committing to the left alternative before checking that the remainder of the input is accepted is the essence of *parsing expression grammars* [28].

52]. Some subgroup matching libraries have guaranteed worst-case linear-time performance based on automata-theoretic techniques, notably Google's RE2 [62]. Intel's Hyperscan [61] is also described as employing automata-theoretic techniques. A key point of Kleenex is implementing the natural backtracking semantics without actually performing backtracking and without requiring storage of the input.

Myers, Oliva and Guimaraes [44] and Okui, Suzuki [50] describe a $O(mn)$, respectively $O(m^2n)$ *POSIX*-disambiguated matching algorithms. Sulzmann and Lu [57] use Brzozowski [20] and Antimirov derivatives [11] for Perl-style subgroup matching for greedy and POSIX disambiguation. Borsotti, Breveglieri, Reghizzi, and Morzenti [16, 17] have devised a Berry-Sethi based parser generator that can be configured for greedy or POSIX disambiguation.

Regular expression parsing. Full RE *parsing*, also called RE matching [29], generalizes subgroup matching to return a full parse tree. The set of parses are exactly the elements of a regular expression read as a *type* [29, 35]: 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 *(McNaughton-Yamada-)Thompson NFA* [42, 64] 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 [31, 33]. A Thompson NFA equipped with $0, 1$ outputs [31] is a certain kind of oracle machine. The bit-code it generates can also be computed directly from the RE underlying the Thompson automaton [35, 49]. The *greedy RE parsing problem* produces the lexicographically least bit-code for a string matching a given RE. Kearns [37], Frisch and Cardelli [29] 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 [31] and an optimally streaming [32] greedy regular expression parsing algorithm. The algorithm works for all NFAs, indeed transducers, not just Thompson NFAs.

Sulzman and Lu [58] remark that POSIX is notoriously difficult to implement correctly and show how to use Brzozowski derivatives [20] for POSIX RE parsing.

Regular expression implementation optimizations. There are specialized RE matching tools and techniques too numerous to review comprehensively. We mention a few employing automaton optimization techniques potentially applicable to Kleenex, but presently unexplored. Yang, Manadhata, Horne, Rao, Ganapathy [75] 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 [54] implement NFAs directly on an FPGA, essentially performing

NFA-simulation in parallel; it outperforms GNU `grep`. Brodie, Taylor, Cytron [18] 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 [55]. Navarro and Raffinot [48] show how to code DFAs compactly for efficient simulation.

Finite state transducers. From RE parsing it is a surprisingly short distance to the implementation of arbitrary nondeterministic finite state transducers (FSTs) [13, 43]. 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 noted, 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 [36], but may be inherently quadratically bigger than automata, even for DFAs [24, Theorem 23]. This is why Kleenex employs well-formed context-free grammars, which are much more compact than regular expressions.

Streaming string transducers. We have shown in this paper that the greedy semantics of arbitrary FSTs can be compiled to a *subclass* of streaming string transducers (SSTs). SSTs extensionally correspond to regular transductions, functions implementable by 2-way deterministic finite-state transducers [4], MSO-definable string transductions [25] and a combinator language analogous to regular expressions [9]. The implementation techniques used in Kleenex appear to be directly applicable to all SSTs, not just the ones corresponding to FSTs.

DReX [7] is a combinatory functional language for expressing all SST-definable transductions. Kleenex without register operations is expressively more restrictive; with copy-less register operations it appears to compactly code exactly the nondeterministic SSTs and thus SSTs. Programs in DReX must be unambiguous by construction while programs in Kleenex may be nondeterministic and ambiguous, which is greedily disambiguated.

Symbolic transducers. Veanes, Molnar, Mytkowics [69] employ symbolic transducers [68, 23] in the implementation of the Microsoft Research languages BEK[11] and BEX[12] for multicore execution. These techniques can be thought of as synthesizing code that implements the transition function of a finite state machine not only efficiently, but also compactly. Tabling in code form (switch

[11] `http://research.microsoft.com/en-us/projects/bek`

[12] `http://research.microsoft.com/en-us/projects/bex`

statement) or data form (lookup in array) is the standard implementation technique for the transition function. It is efficient when applicable, but not compact enough for large alphabets and multistrided processing. Kleenex employs basic symbolic transition. Compact coding of multistrided transitions is likely to be crucial for exploiting word-level parallelism—processing 64 bits at a time—in practice.

Parallel transducer processing. Allender and Mertz [3] show that the functions computable by cost register automata [6], which generalize the string monoid used in SSTs to admit arbitrary monoids and more general algebraic structures, are in NC and thus inherently parallelizable. This appears to be achievable by performing relational FST-composition by matrix multiplication on the matrix representation of FSTs [13], which can be performed by parallel reduction. This requires in principle running an FST from all states, not just the input state, on input string fragments. Mytkowicz, Musuvathi, Schulte [47] observe that there is often a small set of cut states sufficient to run each FST. This promises to be an interesting parallel harness for a suitably adapted Kleenex implementation running on fragments of very large inputs.

Syntax-directed translation schemes. A Kleenex program is an example of a *syntax-directed translation scheme (SDTS)* or a domain-specific stream processing language such as PADS [26, 27] and Ziria [55]. In these the underlying grammar is typically deterministic modulo short lookahead so that semantic actions can be executed immediately when encountered during parsing.

Kleenex is restricted to non-self-embedding grammars to avoid the matrix-multiplication lower bound on general context-free parsing [40]; it supports full nondeterminism without lookahead restriction, though. A key contribution of Kleenex is that semantic actions are scheduled no earlier than semantically permissible and no later than necessary.

C.9 Conclusions

We have presented Kleenex, a convenient language for specifying nondeterministic finite state transducers, and its compilation to machine code implementing streaming string transducers.

Kleenex is comparatively expressive and performs consistently well. For complex regular expressions with nontrivial amounts of output it is almost always better than industrial-strength text processing tools such as RE2, Ragel, `AWK`, `sed` and RE-libraries of Perl, Python and Tcl in the evaluated use cases.

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, specifically visibly push-down transducers [51, 59], restricted versions of backreferences and approximate regular expression matching[45, 74];
- known, but so far unexplored optimizations, such as multistriding, automata minimization and symbolic representation, hybrid FST simulation and SST construction;
- massively parallel (log-depth, linear work) processing.

Bibliography

[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] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Pearson Education, 2006.

[3] E. Allender and I. Mertz. Complexity of regular functions. In *Proc. LATA*, 2015.

[4] R. Alur and P. Černý. Expressiveness of streaming string transducers. In *Proc. Foundations of Software Technology and Teoretical Computer Science (FSTTCS)*, 2010.

[5] R. Alur and P. Černý. Streaming transducers for algorithmic verification of single-pass list-processing programs. *ACM SIGPLAN Notices*, 46(1):599–610, 2011.

[6] R. Alur, L. D'Antoni, J. Deshmukh, M. Raghothaman, and Y. 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.

[7] 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.

[8] R. Alur and J. Deshmukh. Nondeterministic streaming string transducers. *Automata, Languages and Programming*, 2011.

[9] 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.

[10] M. Anselmo, D. Giammarresi, and S. Varricchio. Finite automata and non-self-embedding grammars. In *Implementation and Application of Automata*, pages 47–56. Springer, 2003.

[11] V. Antimirov. Partial derivatives of regular expressions and finite automaton constructions. *Theor. Comput. Sci.*, 155(2):291–319, 1996.

[12] M.-P. Béal and O. Carton. Determinization of transducers over finite and infinite words. *Theoretical Computer Science*, 289(1):225–251, Oct. 2002.

[13] J. Berstel. *Transductions and Context-Free Languages*. Teubner, 1979.

[14] 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.

[15] P. Bille and M. Thorup. Regular expression matching with multi-strings and intervals. In *Proc. 21st ACM-SIAM Symposium on Discrete Algorithms (SODA)*, 2010.

[16] A. Borsotti, L. Breveglieri, S. C. Reghizzi, and A. Morzenti. BSP: A parsing tool for ambiguous regular expressions. In *Implementation and Application of Automata*, pages 313–316. Springer, 2015.

[17] A. Borsotti, L. Breveglieri, S. C. Reghizzi, and A. Morzenti. From ambiguous regular expressions to deterministic parsing automata. In *Implementation and Application of Automata*, pages 35–48. Springer, 2015.

[18] 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.

[19] A. Brüggemann-Klein and D. Wood. One-unambiguous regular languages. *Information and computation*, 140(2):229–253, 1998.

[20] J. A. Brzozowski. Derivatives of regular expressions. *J. ACM*, 11(4):481–494, 1964.

[21] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. The MIT Electrical Engineering and Computer Science Series. MIT Press and McGraw-Hill, 3d edition, 2009.

[22] L. D'Antoni and M. Veanes. Static Analysis of String Encoders and Decoders. In *VMCAI 2013*, volume 7737 of *LNCS*, pages 209–228. Springer Verlag, 2013.

[23] 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.

[24] 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.

[25] 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.

[26] K. Fisher and R. Gruber. PADS: a domain-specific language for processing ad hoc data. *ACM Sigplan Notices*, 40(6):295–304, 2005.

[27] K. Fisher and D. Walker. The PADS project: an overview. In *Proceedings of the 14th International Conference on Database Theory*, pages 11–17. ACM, 2011.

[28] B. Ford. Parsing expression grammars: a recognition-based syntactic foundation. In *ACM SIGPLAN Notices*, number 1 in 39, pages 111–122. ACM, 2004.

[29] 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 (LNCS)*, pages 618–629. Springer, July 2004.

[30] J. Goyvaerts and S. Levithan. *Regular Expressions Cookbook*. O'Reilly, 2009.

[31] 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.

[32] 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.

[33] C. Graulund. On automata-theoretic characterizations of regular expressions as types. Bachelor Thesis, Department of Mathematics, University of Copenhagen, May 2015.

[34] P. Hazel. PCRE – Perl-compatible regular expressions. Concatenation of PCRE man pages, January 3 2010.

[35] F. Henglein and L. Nielsen. Regular expression containment: Coinductive axiomatization and computational interpretation. In *Proc. 38th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL)*, volume 46 of *SIGPLAN Notices*, pages 385–398. ACM Press, January 2011.

[36] T. Johnson, N. Robertson, P. D. Seymour, and R. Thomas. Directed treewidth. *Journal of Combinatorial Theory, Series B*, 82(1):138–154, 2001.

[37] S. Kearns. Extending regular expressions with context operators and parse extraction. *Software - Practice and Experience*, 21(8):787–804, 1991.

[38] K. Kosako. The Oniguruma regular expression library. `http://www.geocities.jp/kosako3/oniguruma/`, 2014.

[39] D. Kozen. *Automata and computability*. Springer Verlag, 1997.

[40] L. Lee. Fast context-free grammar parsing requires fast boolean matrix multiplication. *Journal of the ACM (JACM)*, 49(1):1–15, 2002.

[41] M. Lutz. *Programming Python*, volume 8. O'Reilly, 4th edition edition, December 2010.

[42] R. McNaughton and H. Yamada. Regular expressions and state graphs for automata. *IRE Trans. on Electronic Comput.*, EC-9(1):38–47, 1960.

[43] M. Mohri. Finite-state transducers in language and speech processing. *Computational linguistics*, 23(2):269–311, 1997.

[44] E. Myers, P. Oliva, and K. Guimarães. Reporting exact and approximate regular expression matches. In *Combinatorial Pattern Matching*, pages 91–103. Springer, 1998.

[45] E. W. Myers and W. Miller. Approximate matching of regular expressions. *Bulletin of mathematical biology*, 51(1):5–37, 1989.

[46] G. Myers. A four Russians algorithm for regular expression pattern matching. *J. ACM*, 39(2):432–448, 1992.

[47] 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.

[48] G. Navarro and M. Raffinot. Compact DFA representation for fast regular expression search. *Algorithm Engineering*, pages 1–13, 2001.

[49] L. Nielsen and F. Henglein. Bit-coded Regular Expression Parsing. In *Proc. 5th Int'l Conf. on Language and Automata Theory and Applications (LATA)*, volume 6638 of *Lecture Notes in Computer Science (LNCS)*, pages 402–413. Springer, May 2011.

[50] S. Okui and T. Suzuki. Disambiguation in regular expression matching via position automata with augmented transitions. In M. Domaratzki and K. Salomaa, editors, *Implementation and Application of Automata*, volume 6482 of *Lecture Notes in Computer Science*, pages 231–240. Springer Berlin Heidelberg, 2011.

[51] J.-F. Raskin and F. Servais. Visibly Pushdown Transducers. In L. Aceto, I. Damgård, L. A. Goldberg, M. Halldórsson, A. Ingólfsdóttir, and I. Walukiewicz, editors, *Automata, Languages and Programming*, volume 5126 of *Lecture Notes in Computer Science*, pages 386–397. Springer Berlin Heidelberg, 2008.

[52] A. Rathnayake and H. Thielecke. Static analysis for regular expression exponential runtime via substructural logics. *CoRR*, abs/1405.7058, 2014.

[53] M. Schützenberger. Sur une variante des fonctions sequentielles. *Theoretical Computer Science*, 4(1):47–57, Feb. 1977.

[54] 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.

[55] G. Stewart, M. Gowda, G. Mainland, B. Radunovic, D. Vytiniotis, and C. L. Agulló. Ziria: A DSL for wireless systems programming. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 415–428. ACM, 2015.

[56] S. Sugiyama and Y. Minamide. Checking time linearity of regular expression matching based on backtracking. In *IPSJ Transactions on Programming*, number 3 in 7, pages 1–11, 2014.

[57] M. Sulzmann and K. Z. M. Lu. Regular expression sub-matching using partial derivatives. In *Proc. 14th symposium on Principles and practice of declarative programming*, PPDP '12, pages 79–90, New York, NY, USA, 2012. ACM.

[58] 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.

[59] J.-M. Talbot and P.-A. Reynier. Visibly Pushdown Transducers with Well-nested Outputs. Technical report, Aix Marseille Universite, CNRS, 2014.

[60] The GNU Project. http://www.gnu.org/software/coreutils/coreutils.html, 2015.

[61] The Hyperscan authors. Hyperscan. https://01.org/hyperscan, October 2015.

[62] The RE2 authors. RE2. https://github.com/google/re2, 2015.

[63] The RE2J authors. RE2J. https://github.com/google/re2j, 2015.

[64] K. Thompson. Programming techniques: Regular expression search algorithm. *Commun. ACM*, 11(6):419–422, 1968.

[65] A. Thurston. Ragel state machine compiler. http://www.colm.net/open-source/ragel/, 2015.

[66] G. van Noord and D. Gerdemann. Finite State Transducers with Predicates and Identities. *Grammars*, 4(3):263–286, 2001.

[67] M. Veanes. Symbolic String Transformations with Regular Lookahead and Rollback. In *Ershov Informatics Conference (PSI'14)*. Springer Verlag, 2014.

[68] 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.

[69] 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.

[70] P. Wadler. Deforestation: transforming programs to eliminate trees. *Theoretical Computer Science*, 73(2):231–248, June 1990.

[71] L. Wall, T. Christiansen, and J. Orwant. *Programming Perl*. O'Reilly, 3rd edition, July 2000.

[72] B. W. Watson. Implementing and using finite automata toolkits. *Natural Language Engineering*, 2(04):295–302, 1996.

[73] B. B. Welch, K. Jones, and J. Hobbs. *Practical programming in Tcl and Tk*. Prentice Hall, 4th edition edition, 2003.

[74] S. Wu and U. Manber. Agrep–a fast approximate pattern-matching tool. *Usenix Winter 1992*, pages 153–162, 1992.

[75] 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.

Paper D

PEG Parsing in Less Space Using Progressive Tabling and Dynamic Analysis

The following paper is unpublished at the time of writing, but is planned for submission. The manuscript and the majority of the development was done by the author of this dissertation, with parts of the theory developed in collaboration with Fritz Henglein.

PEG Parsing in Less Space Using Progressive Tabling and Dynamic Analysis[1]

Fritz Henglein and Ulrik Terp Rasmussen

Department of Computer Science, University of Copenhagen (DIKU)

Abstract

Tabular top-down parsing and its lazy variant, Packrat, are linear-time execution models for the TDPL family of recursive descent parsers with limited backtracking. By tabulating the result of each (nonterminal, offset)-pair, we avoid exponential work due to backtracking at the expense of always using space proportional to the product of the input length and grammar size. Current methods for limiting the space usage relies either on manual annotations or on static analyses which are sensitive to the syntactic structure of the grammar.

We present *progressive tabular parsing* (PTP), a new execution model which progressively computes parse tables for longer prefixes of the input and simultaneously generates a leftmost expansion of the parts of the parse tree that can be resolved. Table columns can be discarded on-the-fly as the expansion progresses through the input string, providing best-case constant and worst-case linear memory use. Furthermore, semantic actions are scheduled before the parser has seen the end of the input. The scheduling is conservative in the sense that no action has to be "undone" in the case of backtracking.

The time complexity is $O(dmn)$ where m is the size of the parser specification, n is the size of the input string, and d is either a configured constant or the maximum parser stack depth.

For common data exchange formats such as JSON, we demonstrate practically constant space usage, and without static annotation of the grammar.

D.1 Introduction

Parsing of computer languages has been a topic of research for several decades, leading to a large family of different parsing methods and formalisms. Still, with each solution offering varying degrees of expressivity, flexibility, speed and memory usage, and often at a trade-off, none of them can be regarded as an ideal general approach to solving to all parsing problems. For example, compiler writers often specify their languages in a declarative formalism such as context-free grammars (CFG), relying LL(k) or LR(k) parser generators to turn their specifications into executable parsers. The resulting parsers are often fast, but with the downsides that a separate lexical preprocessing

[1] The order of authors is insignificant.

is needed, and that the programmer is required to mold the grammar into a form that is deterministic for the chosen parser technology. Such solutions require a large investment in time, as identifying the sources of non-determinism in a grammar can be quite difficult. A user who needs to write an ad-hoc parser will thus not find that the amount of time invested makes up for the apparent benefits.

Aho and Ullman's TDPL/GTDPL languages [1], which were later popularized as Parsing Expression Grammars (PEG) [6], provide a formal foundation for the specification of recursive-descent parsers with limited backtracking. They do away with the problem of non-determinism by always having, by definition, a single unique parse for every accepted input. The syntax of PEGs resembles that of CFGs, but where a CFG is a set of generative rules specifying its language, a PEG is a set of rules for a backtracking *recognizer*, and its language is the set of strings recognized. This ensures unique parses, but with the downside that it can sometimes be quite hard to determine what language a given PEG represents. Recognition can be performed in linear time and space by an algorithm which computes a table of results for every (nonterminal, input offset)-pair [1], although it seems to never have been used in practice, probably due to its large complexity constants. Ford's Packrat parsing [5] reduces these constants by only computing the table entries that are needed to resolve the actual parse. However, the memory usage of Packrat is $\Theta(mn)$ for PEGs of size m and inputs of size n, which can be prohibitively expensive for large m and n, and completely precludes applying it in a streaming context where input is potentially infinite. Heuristics for reducing memory usage [10, 17] still store the complete input string, and even risks triggering exponential time behavior. One method [14] can remove both table regions and input prefixes from memory during runtime, but relies on manual annotations and/or a static analysis which does not seem to perform well beyond LL languages [18].

In this paper, we present *progressive tabular parsing* (PTP), a new execution model for the TDPL family of languages. The method is based on the tabular parsing of Aho and Ullman, but avoids computing the full parse table at once. We instead start by computing a table with a single column based on the first symbol in the input. For each consecutive symbol, we append a corresponding column to the table and update all other entries based on the newly added information. We continue this until the end of the input has been reached and the full parse table has been computed. During this process, we have access to partial parse tables which we use to guide a leftmost expansion of the parse tree for the overall parse. Whenever a prefix of the input has been uniquely parsed by this process, the prefix and its corresponding table columns can be removed from memory. The result is a linear-time parsing algorithm which still uses $O(mn)$ memory in the worst case, but $O(m)$ in the best case. Since we have access to the partial results of every nonterminal during parsing, a simple dynamic analysis can use the table to rule out alternative branches and speculatively expand the parse tree before the corresponding production has been fully resolved. The speculation is conservative and never has to undo an expansion unless the whole parse turns out to fail. The analysis changes the time complexity to $O(dmn)$ for a configurable constant d bounded by the maximum stack depth of the parser, but preliminary experiments suggests that it pays for itself in practice by avoiding the computation of unused table entries.

The method can be formulated elegantly using least fixed points of monotone table operators in the partial order of tables with entrywise comparison, and where unresolved entries are considered a bottom element in the partial order. The computation of parse tables is then an instance of *chaotic iteration* [4] for computing least fixed points using a work set instead of evolving all entries in parallel. The work set is maintained such that we obtain meaningful partial parse tables as intermediate results which can be used by the dynamic analysis. Linear time is obtained by using an auxiliary data structure to ensure that each table entry is added to the work set at most once.

Our evaluation demonstrates that PTP dynamically adapts its memory usage based on the amount of lookahead required to resolve productions. The complexity constant due to indiscriminately computing all entries of the parse table can be quite large, but we are confident that this problem can be alleviated in the same way as Packrat reduced the constants for conventional tabular parsing. We believe that our general formulation of PTP offers a solid foundation for further development of both static and dynamic analyses for improving performance.

To summarize, we make the following contributions:

- *Progressive tabular parsing* (PTP), a new execution model for the TDPL family of parsing formalisms. The execution of a program proceeds by progressively computing parse tables, one for each prefix of the input, using the method of *chaotic iteration* for computing least fixed points. Meanwhile, a leftmost expansion of the parse tree is generated in a streaming fashion using the parse table as an oracle. Table columns are discarded on-the-fly as soon as the method detects that a backtracking parser would never have to return to the corresponding part of the input.

- An algorithm for computing progressive parse tables in an incremental fashion. It operates in amortized time $O(mn)$ for grammars of size m and inputs of size n, and produces n progressive approximations of the parse table. The algorithm implements the execution model in $O(mn)$ time and space. We show that for certain grammars and inputs, as little as $O(m)$ space is consumed.

- A configurable dynamic analysis which can dramatically improve the streaming behavior of parsers by allowing a longer trace to be generated earlier in the parse. The dynamic analysis changes the time complexity to $O(dmn)$ where d is either a configured constant or the maximum parser stack depth.

- An evaluation of a prototype of the algorithm which demonstrates that a) for an unannotated JSON parser written in the PEG formalism, memory usage is practically constant, b) for parsers of non-LL languages, the algorithm adjusts memory usage according to the amount of lookahead required, c) however, ambiguous tail-recursive programs trigger worst--case behavior.

The rest of the paper is organized as follows. The GTDPL and PEG parsing formalisms are introduced in Section D.2, together with a notion of parse trees and a definition *streaming parsing*. In Section D.3 we recall the linear-time tabular parsing method, but defined using least fixed points. We extend this in Section D.4 to obtain an approximation of the full parse table based on a prefix of the full input string. In the same section, we define the streaming generation of execution traces based on dynamic analysis of approxmation tables, which we then use to present the *progressive tabular parsing* method. In Section D.5 we define—and prove correct—an amortized linear-time algorithm for computing all progressive table approximations for all consecutive prefixes of an input string. A prototype implementation is evaluated on three different parsing programs in Section D.6, where we also point out the main challenges towards a high-performance implementation. We conclude with a discussion of related and future work in Section D.7.

D.2 Parsing Formalism

The *generalized top-down parsing language* (GTDPL) is a language for specifying top-down parsing algorithms with limited backtracking [1, 3]. It has the same recognition power as the *top-down parsing language* (TDPL), from which it was generalized, and *parsing expression grammars* (PEG) [6], albeit using a smaller set of operators.

The top-down parsing formalism can be seen as a recognition-based alternative to declarative formalisms used to describe machine languages, such as context-free grammars (CFGs). A CFG constitutes a set of generative rules that characterize a language, and the presence of ambiguity and non-determinism poses severe challenges when such a specification must be turned into a deterministic parsing algorithm. In contrast, every GTDPL/PEG by definition denotes a deterministic *program* which operates on an input string and returns with an outcome indicating failure or success. The recognition power of CFGs and GTDPL/PEG is incomparable. There are GTDPLs which recognize languages that are not context-free [1], e.g. the language $\{a^n b^n c^n \mid n \geq 0\}$. On the other hand, GTDPL recognition is linear-time [1] and CFG recognition is super-linear [11], which suggests that there exists a context-free languages that cannot be recognized by any GTDPL.[2]

Let Σ be a finite alphabet, and $\mathcal{N}$ a finite set of *nonterminal symbols*.

Definition 32 (Program)**.** A GTDPL program (henceforth just *program*) is a tuple $P = (\Sigma, V, S, R)$ where

1. Σ is a finite input alphabets; and
2. V is a finite set of *nonterminal* symbols; and
3. $S \in V$ is the starting nonterminal; and
4. $R = \{A_0 {\leftarrow} g_0, ..., A_{m-1} {\leftarrow} g_{m-1}\}$ is a non-empty finite set of numbered *rules*, where each A_i is in V and each $g_i \in \mathsf{GExpr}$ is an *expression* generated by the grammar

   $$\mathsf{GExpr} \ni g ::= \epsilon \mid \mathsf{f} \mid a \mid A[B, C]$$

   where $A, B, C \in V$, $a \in \Sigma$. Rules are unique: $i \neq j$ implies $A_i \neq A_j$.

Define the *size* $|P|$ of a program to be the cardinality of its rule set $|R| = m$. When P is understood, we will write $A {\leftarrow} g$ for the assertion $A {\leftarrow} g \in R$. By uniqueness of rule definitions, we can write i_A for the unique index of a rule $A_i {\leftarrow} g_i$ in R. If g_i is of the form $B[C, D]$ we call it a *complex expression*, otherwise we call it a *simple expression*.

The intuitive semantics of a production $A {\leftarrow} B[C, D]$ is to first try parsing the input with B. If this succeeds, parse the remainder with C, otherwise backtrack and parse from the beginning of the input with D. For this reason we call B the *condition* and C and D the *continuation branch* and *failure branch*, respectively.

Given sets X, Y, write $X + Y$ for their disjoint union $\{0\} \times X \cup \{1\} \times Y$.

Definition 33 (Operational semantics)**.** Let $P = (\Sigma, V, S, R)$ be a program and define a matching relation $\Rightarrow_P$ from $V \times \Sigma^*$ to results $r \in \Sigma^* + \{\mathsf{f}\}$. That is, it relates pairs of the form $(A, u) \in V \times \Sigma^*$ to either the failure value f, or a result value $v \in \Sigma^*$ indicating success, where v is the suffix of u that remains unconsumed. We leave out the subscript P when it is clear from the context.

Let $\Rightarrow_P$ be generated by the following rules:

$$(1)\ \frac{}{(A, u) \Rightarrow_P u}\ (A {\leftarrow} \epsilon) \qquad (2)\ \frac{}{(A, u) \Rightarrow_P \mathsf{f}}\ (A {\leftarrow} \mathsf{f})$$

$$(3\text{i})\ \frac{}{(A, au) \Rightarrow_P u}\ (A {\leftarrow} a)$$

$$(3\text{ii})\ \frac{}{(A, u) \Rightarrow_P \mathsf{f}}\ (A {\leftarrow} a \text{ and } a \text{ not prefix of } u)$$

[2] To the best of our knowledge, no such language is known.

$$\text{(4i)}\ \frac{(B,u) \Rightarrow_P v \qquad (C,v) \Rightarrow_P r}{(A,u) \Rightarrow_p r}\ (A{\leftarrow}B[C,D])$$

$$\text{(4ii)}\ \frac{(B,u) \Rightarrow_P \mathsf{f} \qquad (D,u) \Rightarrow_P r}{(A,u) \Rightarrow_p r}\ (A{\leftarrow}B[C,D])$$

The proof derivations generated by the rules will be denoted by subscripted variations of the letter $\mathcal{D}$.

Write $(A,u) \not\Rightarrow_P$ when there does not exist an r such that $(A,u) \Rightarrow_P r$. Say that A *matches* u when $(A,u) \Rightarrow_P v$ for $v \in \Sigma^*$ (note that A does not have to consume all of the input). The *language* recognized by a nonterminal A is the set $L_P(A) = \{u \in \Sigma^* \mid \exists v \in \Sigma^*. (A,u) \Rightarrow_P v\}$. The language *rejected* by A is the set $\overline{L}_P(A) = \{u \in \Sigma^* \mid (A,u) \Rightarrow_P \mathsf{f}\}$. We say that A *handles* u when $u \in L_P(A) \cup \overline{L}_P(A)$. The program P is *complete* if the start symbol S handles all strings $u \in \Sigma^*$.

The following two properties are easily shown by induction.

Proposition D.2.1 (Suffix output). *If $(A,u) \Rightarrow_P (s,w)$, then w is a suffix of u ($\exists v.\, u = vw$).*

Proposition D.2.2 (Determinacy). *If $(A,u) \Rightarrow_P r_1$ by $\mathcal{D}_1$ and $(A,u) \Rightarrow_P r_2$ by $\mathcal{D}_2$, then $\mathcal{D}_1 = \mathcal{D}_2$ and $r_1 = r_2$.*

We recall the following negative decidability results proved by Ford for the PEG formalism [6]. Since any GTDPL can be converted to an equivalent PEG and vice-versa, they hold for GTDPL as well.

Proposition D.2.3. *It is undecidable whether $L_P(A) = \emptyset$ and whether $L_P(A) = \Sigma^*$.*

Proposition D.2.4. *It is undecidable whether a program is complete.*

Parsing Expression Grammars

Having only a single complex operator, GTDPL offers a minimal foundation which simplifies the developments in later sections. The drawback is that it is very hard to determine the language denoted by a given GTDPL program. In order to make examples more readable, we will admit programs to be presented with expressions from the extended set PExpr defined as follows:

$$\mathsf{PExpr} \ni e ::= g \in \mathsf{GExpr} \mid e_1 e_2 \mid e_1 / e_2 \mid e^* \mid !e_1$$

This corresponds to the subset of *predicate-free parsing expressions* extended with the ternary GTDPL operator. A program P with productions in PExpr is called a *PEG program*, and desugars to a pure GTDPL program by adding productions $E{\leftarrow}\epsilon$ and $F{\leftarrow}\mathsf{f}$ and replacing every non-conforming production as follows:

$$\begin{array}{lcl}
A{\leftarrow}e_1e_2 & \longmapsto & A{\leftarrow}B[C,F] \\
 & & B{\leftarrow}e_1 \\
 & & C{\leftarrow}e_2 \\
\hline
A{\leftarrow}e_1/e_2 & \longmapsto & A{\leftarrow}B[E,C] \\
 & & B{\leftarrow}e_1 \\
 & & C{\leftarrow}e_2 \\
\hline
A{\leftarrow}e_1^* & \longmapsto & A{\leftarrow}B[A,E] \\
 & & B{\leftarrow}e_1 \\
\hline
A{\leftarrow}!e_1 & \longmapsto & A{\leftarrow}B[F,E] \\
 & & B{\leftarrow}e_2
\end{array}$$

The desugaring embeds the semantics of PEG in GTDPL [6], so there is no need to introduce semantic rules for parsing expressions. Note that although parsing expressions resemble regular expressions, the recognizers that they denote may not recognize the same languages as their usual set-theoretic interpretation. For example, the expression a^*a recognizes the empty language!

Parse Trees

We are usually interested in providing a parse tree instead of just doing recognition, e.g. for the purpose of executing semantic actions associated with parsing decisions. Unlike generative frameworks, any program uniquely matches an input via a unique derivation $\mathcal{D}$, which we therefore could take as our notion of parse tree. However, for space complexity reasons we will employ a more compact notion for which we also define a bit coding for the purpose of providing a definition of streaming parsing.

A *parse tree* $\mathcal{T}$ is an ordered tree where each leaf node is labeled by the empty string or a symbol in Σ, and each internal node is labeled by a nonterminal subscripted by a symbol from $\mathbf{2} \cup \{\varepsilon\}$ where $\mathbf{2} = \{0, 1\}$.

Definition 34 (Parse trees and codes)**.** For any $A \in V$, $u, v \in \Sigma^*$, and derivation $\mathcal{D} :: (A, u) \Rightarrow_P v$, define simultaneously a *parse tree* $\mathcal{T}_\mathcal{D}$ and a *parse code* $\mathcal{C}_\mathcal{D} \in \mathbf{2}^*$ by recursion on $\mathcal{D}$:

1. If $A{\leftarrow}\epsilon$, respectively $A{\leftarrow}a$, then $\mathcal{T}_\mathcal{D}$ is a node labeled by A_ε with a single child node labeled by ε, respectively a. Let $\mathcal{C}_\mathcal{D} = \varepsilon$.

2. If $A{\leftarrow}B[C, D]$ and $\mathcal{D}_1 :: (B, u) \Rightarrow_P u'$ we must have $\mathcal{D}_2 :: (C, u') \Rightarrow_P v$. Let $\mathcal{T}_\mathcal{D}$ be a node A_0 with subtrees $\mathcal{T}_{\mathcal{D}_1}$ and $\mathcal{T}_{\mathcal{D}_2}$. Let $\mathcal{C}_\mathcal{D} = 0\,\mathcal{C}_{\mathcal{D}_1}\mathcal{C}_{\mathcal{D}_2}$.

3. If $A{\leftarrow}B[C, D]$ and $\mathcal{D}_1 :: (B, u) \Rightarrow_P \mathsf{f}$, then we must have $\mathcal{D}_2 :: (D, u') \Rightarrow_P v$. Create a node labeled by A_1 with a single subtree $\mathcal{T}_{\mathcal{D}_2}$. Let $\mathcal{C}_\mathcal{D} = 1\,\mathcal{C}_{\mathcal{D}_2}$.

The size of a parse tree $|\mathcal{T}|$ is the number of nodes in it. Note that only the parts of a derivation counting towards the successful match contribute to its parse tree, while failing subderivations are omitted. This ensures that parse trees have size proportional to the input, in contrast to derivations which can grow exponentially in the worst case.

Proposition D.2.5 (Linear tree complexity)**.** *Fix a program P. For all $A \in V$ and $u, v \in \Sigma^*$ and derivations $\mathcal{D} :: (A, u) \Rightarrow_P v$ we have $|\mathcal{T}(\mathcal{D})| = O(|u|)$.*

Parse trees and parse codes both provide injective codings of the subset of derivations with non-failing results.

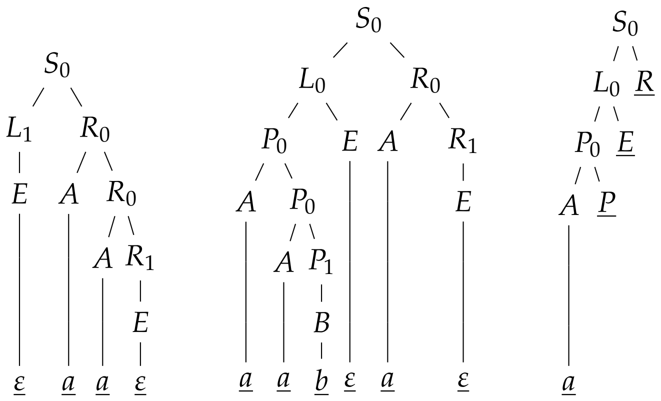
Proposition D.2.6 (Injectivity)**.** *Fix a program P and symbol $A \in V$. For all $u_1, u_2, v_1, v_2 \in \Sigma^*$ and derivations $\mathcal{D}_1 :: (A, u_1) \Rightarrow_P v_1$ and $\mathcal{D}_2 :: (A, u_2) \Rightarrow_P v_2$, if $\mathcal{D}_1 \neq \mathcal{D}_2$, then $\mathcal{T}_{\mathcal{D}_1} \neq \mathcal{T}_{\mathcal{D}_2}$ and $\mathcal{C}_{\mathcal{D}_1} \neq \mathcal{C}_{\mathcal{D}_2}$.*

It is easy to check that a code can be used to construct the corresponding parse tree in linear time, regardless of the size of the underlying derivation. In general, a code can be viewed as an oracle which guides a leftmost expansion of the corresponding parse tree. Any prefix of a code can thus be seen as a partially expanded parse tree. During expansion, we maintain a stack of nodes that are not yet expanded. If the top node is simple it can be expanded deterministically, and if it is complex the next code symbol determines its expansion; its child nodes are pushed on the stack.

Example 6. Consider the PEG program $S \leftarrow (a^* b / \epsilon) a^*$, which desugars into:

$$\begin{array}{llll} S \leftarrow L[R, F] & L \leftarrow P[E, E] & P \leftarrow A[P, B] & R \leftarrow A[R, E] \\ A \leftarrow a & B \leftarrow b & E \leftarrow \epsilon & F \leftarrow \mathsf{f} \end{array}$$

We have derivations $\mathcal{D} :: (S, aa) \Rightarrow \varepsilon$ and $\mathcal{D}' :: (S, aaba) \Rightarrow \varepsilon$. Visualized below is, from left to right: the trees $\mathcal{T}_{\mathcal{D}}$, $\mathcal{T}_{\mathcal{D}'}$, and the partial tree expanded from the prefix 000 of the code $\mathcal{C}_{\mathcal{D}'}$. The leftmost nonterminal leaf is the next to be expanded.


S_0
L_1 R_0
E A R_0
A R_1
E
$\underline{\varepsilon}$ $\underline{a}$ $\underline{a}$ $\underline{\varepsilon}$
S_0
L_0 R_0
P_0 E A R_1
A P_0 E
A P_1
B
$\underline{a}$ $\underline{a}$ $\underline{b}$ $\underline{\varepsilon}$ $\underline{a}$ $\underline{\varepsilon}$
S_0
L_0 $\underline{R}$
P_0 $\underline{E}$
A $\underline{P}$
$\underline{a}$

The parse codes are $\mathcal{C}_{\mathcal{D}} = 01001$ and $\mathcal{C}_{\mathcal{D}'} = 0000101$, respectively. Observe that codes correspond to the subscripts of the internal nodes in the order they would be visited by an in-order traversal, reflecting the leftmost expansion order.

Streaming Parsing

Using parse codes, we can define *streaming parsing*.

Definition 35 (Streaming parsing function)**.** Let $\# \notin \Sigma$ be a special end-of-input marker. A *streaming parsing function* for a program P is a function $f : \Sigma^*(\# \cup \varepsilon) \to \mathbf{2}^*$ which for every input prefix $u \in \Sigma^*$ satisfies the following:

1. it is monotone: For all $v \in \Sigma^*$, $f(uv) = f(u)c'$ for some $c' \in \mathbf{2}^*$.

2. it computes code prefixes: For all $v \in \Sigma^*$ and matching derivations $\mathcal{D} :: (A, uv) \Rightarrow_P w$ ($w \in \Sigma^*$), we have $\mathcal{C}_{\mathcal{D}} = f(u)c'$ for some $c' \in \mathbf{2}^*$.

3. it completes the code: if there exists a matching derivation $\mathcal{D} :: (A, u) \Rightarrow_P w$, then $\mathcal{C}_{\mathcal{D}} = f(u\#)$.

In the rest of this chaper, we develop an algorithm which implements a streaming parsing function as defined above. The code prefix produced allows consumers to perform parsing actions (e.g. construction of syntax trees, evaluation of expressions, printing, etc.) before all of the input string has been consumed. Monotonicity ensures that no actions will have to be "un-done", with the caveat that further input might cause the whole parse to be rejected.

D.3 Tabulation of Operational Semantics

In the following we fix a program $P = (\Sigma, V, S, R)$.

We will be working with various constructions defined as least fixed points of monotone operators on partially ordered sets. A partial order is a pair $(X, \sqsubseteq)$ where X is a set and $\sqsubseteq$ is a reflexive, transitive and antisymmetric relation on X. Given two elements $x, y \in X$, we will write $x \sqsubset y$ when $x \sqsubseteq y$ and $x \neq y$.

For any set X, let $(X, \sqsubseteq)$ be the discrete partial order, the smallest partial order on X (i.e. $x \sqsubseteq x'$ implies $x = x'$). Write $X_\bot$ for the set $X + \{\bot\}$ and let $(X_\bot, \sqsubseteq)$ be the *lifted* partial order with $\bot$ as an adjoined bottom element, i.e. $\forall x \in X_\bot . \bot \sqsubseteq x$.

A *table* on X is a $|P| \times \omega$ matrix T where each entry T_{ij} is in $X_\bot$, and indices (i, j) are in the set $\mathsf{Index} = \{(i, j) \mid 0 \leq i < |P| \wedge 0 \leq j\}$. The set of all tables on X is denoted $\mathsf{Table}(X)$, and forms a partial order $(\mathsf{Table}(X), \sqsubseteq)$ by comparing entries pointwise: for $T, T' \in \mathsf{Table}(X)$, we write $T \sqsubseteq T'$ iff for all $(i, j) \in \mathsf{Index}$, we have $T_{ij} \sqsubseteq T'_{ij}$. Write $\bot \in \mathsf{Table}(X)$ for the table with all entries equal to $\bot \in X_\bot$. It is easy to verify that the partial order on $\mathsf{Table}(X)$ has the following structure:

complete partial order: For all chains $T_0 \sqsubseteq T_1 \sqsubseteq ...$ where $T_i \in \mathsf{Table}(X)$, $i \in \{0, 1, ...\}$, the least upper bound $\bigsqcup_i T_i$ exists.

meet-semilattice: For all non-empty subsets $S \subseteq \mathsf{Table}(X)$, the greatest lower bound $\bigsqcap S$ exists.

A function $F : \mathsf{Table}(X) \to \mathsf{Table}(X)$ is said to be *continuous* if it preserves least upper bounds: For all $S \subseteq \mathsf{Table}(X)$, we have $F(\bigsqcup S) = \bigsqcup_{T \in S} F(T)$. A continous function is automatically *monotone*, meaning that $T \sqsubseteq T'$ implies $F(T) \sqsubseteq F(T')$. A *least fixed point* of F is an element T such that $F(T) = T$ (T is a fixed point) and also $T \sqsubseteq T'$ for all fixed points T'. A general property of complete partial orders is that if F is a continuous function then its least fixed point lfp F exists and is given by

$$\text{lfp}\, F = \bigsqcup_n F^n(\bot)$$

where F^n is the n-fold composition of F with itself. We will also rely on the following generalization:

Lemma D.3.1 (Lower bound iteration)**.** *If* $T \sqsubseteq \text{lfp}\, F$*, then* $\text{lfp}\, F = \bigsqcup_n F^n(T)$.

Parse Tables

We now recall the parse table used in the dynamic programming algorithm for linear time recognition [1], but presented here as a least fixed point. The table will have entries in the set $\mathsf{Res} = \omega + \{\mathsf{f}\}$, i.e. either a natural number or f indicating failure. Given a finite (respectively, infinite) string $w = a_0 a_1 ... a_{n-1}$ ($w = a_0 a_1 ...$), and an offset $0 \le j < n$ ($0 \le j$), write u_j for the suffix $a_j a_{j+1} ... a_{n-1}$ ($a_j a_{j+1} ...$) obtained by skipping the first j symbols.

Definition 36 (Parse table)**.** Let $u \in \Sigma^*$. Define a table operator F^u on $\mathsf{Table}(\mathsf{Res})$ as follows. Let $w = u\#^\omega$, the infinite string starting with u followed by an infinite number of repetitions of the end marker $\# \notin \Sigma$. For any table $T \in \mathsf{Table}(\mathsf{Res})$ define $F^u(T) = T'$ such that for all $(i, j) \in \mathsf{Index}$:

$$T'_{ij} = \begin{cases} \mathsf{f} & A_i {\leftarrow} \mathsf{f} \text{ or } A_i {\leftarrow} a \text{ and } a \text{ not a prefix of } w_j \\ 1 & A_i {\leftarrow} a \text{ and } a \text{ is a prefix of } w_j \\ 0 & A_i {\leftarrow} \epsilon \\ m + m' & A_i {\leftarrow} A_x[A_y, A_z];\ T_{xj} = m;\ T_{y(j+m)} = m' \\ \mathsf{f} & A_i {\leftarrow} A_x[A_y, A_z];\ T_{xj} = m;\ T_{y(j+m)} = \mathsf{f} \\ T_{zj} & A_i {\leftarrow} A_x[A_y, A_z];\ T_{xj} = \mathsf{f} \\ \bot & \text{otherwise} \end{cases}$$

The operator F^u is easily seen to be continuous, and we define the *parse table for u* by $T(u) = \operatorname{lfp} F^u$.

For any $u \in \Sigma^*$, the table $T(u)$ is a tabulation of all parsing results on all suffixes of u:

Theorem D.3.2 (Fundamental theorem)**.** *Let $u \in \Sigma^*$ and consider $T(u)$ as defined above. For all $(i, j) \in \mathsf{Index}$:*

1. *$j \leq |u|$ and $T(u)_{ij} = \mathsf{f}$ iff $(A_i, u_j) \Rightarrow_P \mathsf{f}$; and*
2. *$j \leq |u|$ and $T(u)_{ij} = m \in \omega$ iff $(A_i, u_j) \Rightarrow_P u_{j+m}$; and*
3. *$j \leq |u|$ and $T(u)_{ij} = \bot$ iff $(A_i, u_j) \not\Rightarrow_P$;*
4. *if $j > |u|$ then $T_{ij} = T_{i|u|}$*

The converse also holds: for any T satisfying the above, we have $T = T(u)$.

Property 4 is sufficient to ensure that all parse tables have a finitary representation of size $|P| \times |u|$. It is straightforward to extract a parse code from $T(u)$ by applying Definition 34 and the theorem.

Example 7. Consider the program P from Example 6. The tables $T = T(aa)$ and $T' = T(aaba)$ are shown below:

	0	1	2 ···	
	a	a	# ···	
A	1	1	f	
B	f	f	f	
E	0	0	0	
F	f	f	f	
L	0	0	0	···
P	f	f	f	
R	2	1	0	
S	2	1	0	

	0	1	2	3	4 ···	
	a	a	b	a	# ···	
A	1	1	f	1	f	
B	f	f	1	f	f	
E	0	0	0	0	0	
F	f	f	f	f	f	
L	3	2	1	0	0	···
P	3	2	1	f	f	
R	2	1	0	1	0	
S	4	3	2	1	0	

Note that columns 1,2 in the left table equals columns 3,4 in the right table. In general, columns depend on the corresponding input suffix but are independent of the previous columns. This is a simple consequence of Theorem D.3.2.

For a table T and $m \in \omega$, let $T[m]$ be the table obtained by removing the first m columns from T, i.e. $T[m]_{ij} = T_{i(j+m)}$.

Corollary D.3.3 (Independence)**.** *Let $u \in \Sigma^*$. For all $0 \leq m$, we have $T(u)[m] = T(u_m)$.*

Proof. By Theorem D.3.2. For example, if $T(u)_{i(j+m)} = m'$ for some m' then $(A_i, u_{j+m}) \Rightarrow u_{j+m+m'}$. Have $(u_m)_j = u_{m+j}$, so $(A_i, (u_m)_j) \Rightarrow (u_m)_{j+m'}$, and therefore $T(u_m)_{ij} = m'$. □

Independence leads to the linear-time parsing algorithm of Aho and Ullman. For input u with $|u| = n$, compute $T(u)$ column by column, starting from the right. For each $m \leq n$, we compute column m by fixed point iteration of F^{u_m} on the current table state. Since $T(u)[m+1] = T(u_{m+1})$ has already been computed, only $|P|$ entries need to be processed in each step, which takes time $O(|P|^2)$.

D.4 Streaming Parsing with Tables

The linear-time parsing algorithm has asymptotically optimal time complexity. However, it always uses space linear in the length of the input string, since all columns of the parse table has to be computed before the final result can be obtained. For large grammars and inputs, this can be prohibitively expensive. In the following we describe a method for computing only an initial part of the table. The initial columns will in some cases provide enough information to construct a prefix of the parse code and allow us to continue parsing with a smaller table, saving space.

Let us illustrate the idea by an example. Let $w = uv$ be an input string, and let $A_i \leftarrow A_x[A_y, A_z]$ be a rule in the program. Suppose that by analyzing only the prefix u, we can conclude that there is a constant m such that $T(uv')_{x0} = m$ for all v'. In particular, this holds for $v' = v$, so $T(w)_{i0} \in \omega$ if and only if $T(w)_{i0} = m + m'$ where $m' = T(w)_{ym} = T(w)[m]_{y0} = T(w_m)_{y0}$ (the last equation follows by independence). By examining only the prefix u, we have thus determined that the result only depends on $T(w_m)$, freeing up m columns of table space. The process can be repeated for the remaining input w_m.

We will need an analysis that can predict results as described. The theoretically optimal analysis is defined as follows:

Definition 37 (Optimal prefix table)**.** Let $u \in \Sigma^*$, and define the *optimal prefix table* $T^{\sqcap}(u) \in \mathsf{Table}(\mathsf{Res})$ as the largest approximation of all the complete tables for all extensions of u:

$$T^{\sqcap}(u) = \bigsqcap_{v \in \Sigma^*} T(uv)$$

Theorem D.4.1. *For all u, i, j:*

1. *if $T^{\sqcap}(u)_{ij} \neq \bot$ then $\forall v.\, T(uv)_{ij} = T^{\sqcap}(u)_{ij}$;*
2. *if $(\forall v.\, T(uv)_{ij} = r \neq \bot)$, then $T^{\sqcap}(u)_{ij} = r$.*

Unfortunately, we cannot use this for parsing, as the optimal prefix table is too precise to be computable:

Theorem D.4.2. *There is no procedure which computes $T^{\sqcap}(u)$ for all GTDPLs P and input prefixes u.*

Proof. Assume otherwise that $T^{\sqcap}(u)$ is computable for any u and GTDPL P. Then $L(P) = \emptyset$ iff $T^{\sqcap}(\varepsilon)_{i_S,0} = \mathsf{f}$. Hence emptiness is decidable, a contradiction by Proposition D.2.3. □

A conservative and computable approximation of $T^{\sqcap}$ can easily be defined as a least fixed point. Given a table operator F and a subset $J \subseteq \mathsf{Index}$ define a restricted operator F_J by

$$F_J(T)_{ij} = \begin{cases} F(T)_{ij} & \text{if } (i,j) \in J \\ T_{ij} & \text{otherwise} \end{cases}$$

If $J = \{(p,q)\}$ is a singleton, write F_{pq} for F_J. Clearly, if F is continuous then so is F_J.

For any $u \in \Sigma^*$, define an operator $F^{(u)}$ by $F^{(u)} = F^{u}_{J_u}$ where $J_u = \{(i,j) \in \mathsf{Index} \mid j < |u|\}$. The *prefix table* for u is the least fixed point of this operator:

$$T^{<}(u) = \operatorname{lfp} F^{(u)}$$

Intuitively, a prefix table contains as much information as can be determined without depending on column $|u|$. Prefix tables are clearly computable by virtue of being least fixed points, and properly approximate the optimal analysis:

Theorem D.4.3 (Approximation). *For all $u \in \Sigma^*$, we have $T^{<}(u) \sqsubseteq T^{\sqcap}(u)$. In particular, if $T^{<}(u)_{ij} = m$ or $T^{<}(u)_{ij} = \mathsf{f}$, then $\forall v.\, T(uv)_{ij} = m$ or $\forall v.\, T(uv)_{ij} = \mathsf{f}$, respectively.*

Perhaps not surprisingly, prefix tables become better approximations as the input prefix is extended. We will make use of this property and Lemma D.3.1 to efficiently compute prefix tables in an incremental fashion:

Proposition D.4.4 (Prefix monotonicity). *For all $u, v \in \Sigma^*$, we have $T^{<}(u) \sqsubseteq T^{<}(uv)$.*

The full parse table can be recovered as a prefix table if we just append an explicit end marker to the input string:

Proposition D.4.5 (End marker). *For all $u \in \Sigma^*$ and $(i,j) \in \mathsf{Index}$, if $j \leq |u|$ then $T^{<}(u\#)_{ij} = T(u)_{ij}$.*

Independence carries over to prefix tables. For all $u \in \Sigma^*$ and $m \geq 0$, we thus have $T^{<}(u)[m] = T^{<}(u_m)$.

Streaming Code Construction

The resolved entries of a prefix table can be used to guide a partial leftmost expansion of a parse tree. We model this expansion process by a labeled transition system which generates the corresponding parse code. By constructing the expansion such that it is a prefix of all viable expansions, the parse code can be computed in a streaming fashion. In order to determine as much of the parse code as possible, we speculatively guess that choices succeed when a dynamic analysis can determine that the alternative must fail.

Definition 38 (Leftmost parse tree expansion)**.** Let $T \in \mathsf{Table}(\mathsf{Res})$ be a table and $d \in \omega$ a *speculation constant*. Define a labeled transition system $\mathcal{E}_T = (Q, E)$ with states $Q = V^* \times \omega$ and transitions $E \subseteq \{q \xrightarrow{c} q' \mid c \in \mathbf{2}^*; q, q' \in Q\}$. Let E be the smallest set such that for all $A_i \in V$, $\vec{K} \in V^*$ and $j \in \omega$:

1. If $A_i \leftarrow A_x[A_y, A_z]$; and either $T_{xj} \in \omega$ or $(A_z\vec{K}, j)$ $\mathbf{fails}_d$, then:
$$(A_i\vec{K}, j) \xrightarrow{0} (A_xA_y\vec{K}, j) \in E$$

2. If $A_i \leftarrow A_x[A_y, A_z]$; and $T_{xj} = \mathsf{f}$, then:
$$(A_i\vec{K}, j) \xrightarrow{1} (A_z\vec{K}, j) \in E$$

3. If $A_i \leftarrow \epsilon$ or $A_i \leftarrow a$; and $T_{ij} = m$, then:
$$(A_i\vec{K}, j) \xrightarrow{\varepsilon} (\vec{K}, j) \in E$$

4. If $q \xrightarrow{c} q' \in E$ and $q' \xrightarrow{c'} q''$, then: $q \xrightarrow{cc'} q'' \in E$.

where for all $\vec{K}, j, n$, write $(\vec{K}, j)$ $\mathbf{fails}_n$ if $\vec{K} = A_i\vec{K}'$ and either

1. $T_{ij} = \mathsf{f}$; or
2. $T_{ij} = m$, $n = n' + 1$ and $(\vec{K}', j + m)$ $\mathbf{fails}_{n'}$.

A state encodes the input offset and the stack of leaves that remain unexpanded. The node on the top of the stack is expanded upon a transition to the next state, with the expansion choice indicated in the label of the transition. The system is deterministic in the sense that every state can step to at most one other state in a single step (the label is determined by the source state).

The highlighted disjunct allows us to speculatively resolve a choice as succeeding when the failure branch is guaranteed to fail. This is determined by examining the table entries for at most d nonterminals on the current stack $\vec{K}$.

Example 8. The partial parse tree of Example 6 corresponds to the following steps in $\mathcal{E}_{T'}$ where T' is the table from Example 7:

$$(S, 0) \xrightarrow{0} (LR, 0) \xrightarrow{0} (PER, 0) \xrightarrow{0} (APER, 0) \xrightarrow{\varepsilon} (PER, 1)$$

A state q is *quiescent* if there is no transition from it. Say that q is *convergent* and write $q \xrightarrow{c} q' \downarrow$ if either there is a path $q \xrightarrow{c} q'$ such that q' quiescent; or, q is already quiescent and $q' = q$ and $c = \varepsilon$. Clearly, if such c and q' exists, then they are unique and can be effectively determined. Otherwise, we say that q is *divergent*.

Expansions compute coded (matching) derivations in full parse tables:

Proposition D.4.6. *Let $u \in \Sigma^*$ and consider the system $\mathcal{E}_{T(u)}$.*

1. *There is a derivation $\mathcal{D} :: (A, u) \Rightarrow_P u_m$ with $c = \mathcal{C}_\mathcal{D}$ if and only if $(A, 0) \xrightarrow{c} (\varepsilon, m) \downarrow$.*

2. *We have $(A, u) \Rightarrow_p$ f if and only if $(A, 0)$ **fails**$_n$ for some n.*

It follows that a state $(A, 0)$ is only divergent if the input is unhandled:

Proposition D.4.7. *Let $u \in \Sigma^*$ and consider the system $\mathcal{E}_{T(u)}$. Then $(A, u) \not\Rightarrow_P$ if and only if $(A, 0)$ is divergent.*

Hence, if P is complete, then every state is convergent in $\mathcal{E}_{T(u)}$, and the relation $q \xrightarrow{c} q' \downarrow$ becomes a total function $q \mapsto (c, q')$.

The function associating every input prefix u with the code c given by $(S, 0) \xrightarrow{c} q' \downarrow$ in the system $\mathcal{E}_{T^<(u)}$ is a streaming parse function as per Definition 35. This is ensured by the following sufficient condition, which states that expansions never "change direction" as the underlying table is refined:

Proposition D.4.8. *If $T \sqsubseteq T'$ and $(\vec{K}, j) \xrightarrow{c} (\vec{K}', j')$ in $\mathcal{E}_T$, then either $(\vec{K}, j) \xrightarrow{c} (\vec{K}', j')$ in $\mathcal{E}_{T'}$ or $(\vec{K}, j)$ fails in $\mathcal{E}_{T'}$.*

Expansions also never backtrack in the input, that is, if $(\vec{K}, j) \xrightarrow{c} (\vec{K}', j')$ then $j \leq j'$. This allows us to discard the initial columns of a table as we derive a leftmost expansion:

Proposition D.4.9. *Let T be a table. Then $(\vec{K}, m) \xrightarrow{c} (\vec{K}', n)$ in $\mathcal{E}_T$ if and only if $(\vec{K}, 0) \xrightarrow{c} (\vec{K}', n - m)$ in $\mathcal{E}_{T[m]}$.*

Progressive Tabular Parsing

Assume that P is a complete program. We use the constructions of this section to define our *progressive tabular parsing* procedure. The algorithmic issues of space and time complexity will not be of our concern yet, but will we be adressed in the following section.

Given an input string with end marker $w\# = a_0 a_1 ... a_{n-1}$ ($a_{n-1} = \#$), the procedure decides whether there exists a matching derivation $\mathcal{D} :: (S, w) \Rightarrow_P w_k$, and in that case produces $\mathcal{C}_\mathcal{D}$ in a streaming fashion. In each step $0 \leq k \leq$

(1)	a
A	1
B	f
E	**0**
F	f
L	⊥
P	⊥
R	⊥
S	⊥

(2)	a	a
A	1	1
B	f	f
E	**0**	0
F	f	f
L	⊥	⊥
P	⊥	⊥
R	⊥	⊥
S	⊥	⊥

(3)	a	a	b	
A	**1**	**1**	f	⊥
B	f	f	**1**	⊥
E	**0**	0	0	⊥
F	f	f	f	⊥
L	⊥	⊥	⊥	⊥
P	**3**	2	1	⊥
R	2	1	0	⊥
S	⊥	⊥	⊥	⊥

(4)	a	a	b	a	
A	**1**	**1**	f	**1**	⊥
B	f	f	**1**	f	⊥
E	**0**	0	0	**0**	⊥
F	f	f	f	f	⊥
L	⊥	⊥	⊥	⊥	⊥
P	**3**	2	1	⊥	⊥
R	2	1	0	⊥	⊥
S	⊥	⊥	⊥	⊥	⊥

(5)	a	a	b	a	#
A	**1**	**1**	f	**1**	f
B	f	f	**1**	f	f
E	**0**	0	0	**0**	**0**
F	f	f	f	f	f
L	⊥	⊥	⊥	⊥	0
P	**3**	2	1	⊥	f
R	2	1	0	⊥	0
S	⊥	⊥	⊥	⊥	0

$(1) : (S, aaba\#) \xrightarrow{0} (LR, aaba\#)$
$(2) : (LR, aaba\#)$
$(3) : (LR, aaba\#) \xrightarrow{0} (PER, aaba\#) \xrightarrow{0} (APER, aaba\#) \rightarrow (PER, aba\#)$
$\xrightarrow{0} (APER, aba\#) \rightarrow (PER, ba\#) \xrightarrow{1} (BER, ba\#) \rightarrow (ER, a\#)$
$(4) : (ER, a\#) \rightarrow (R, a\#) \xrightarrow{0} (AR, a\#) \rightarrow (R, \#)$
$(5) : (R, \#) \xrightarrow{1} (E, \#) \rightarrow (\varepsilon, \#)$

Figure D.1: In the top is five consecutive tables during the parse of input $w = aaba$, using the program from Example 6. Only the columns to the right of the dashed line has to be stored for the next iteration. Newly computed entries are colored; entries considered by the expansion process are written in bold face. The progression of the leftmost expansion is shown below.

n, we compute a table $T^k \in \mathsf{Table}(\mathsf{Res})$, a stack $\vec{K}^k$, an offset $m^k \le k$ and a code chunk $c^k \in \mathbf{2}^*$. Upon termination, we will have $\mathcal{C}_\mathcal{D} = c^0 c^1 ... c^n$.

Initially $T^0 = T^<(\varepsilon)$, $q^0 = (S, 0)$ and $c^0 = \varepsilon$. For each $1 \le k \le n$, the values $T^k, \vec{K}^k, m^k$ and c^k are obtained by

$$\begin{aligned} T^k &= T^<(a_{m^{k-1}} ... a_{k-1}) \\ m^k &= m^{k-1} + m' \\ \text{where } (\vec{K}^{k-1}, 0) &\xrightarrow{c^k} (\vec{K}^k, m') \downarrow \end{aligned}$$

Since P is complete, we have by Proposition D.4.7 that the last line above can be resolved.

If $\vec{K}^n = \varepsilon$, accept the input; otherwise reject.

Theorem D.4.10. *The procedure computes $\mathcal{C}_{\mathcal{D}}$ iff there is a derivation $\mathcal{D} :: (S, w) \Rightarrow_P w_k$.*

Proof. We claim that after each step k, we have $(S, 0) \overset{c^0 \ldots c^k}{\rightarrow} (\vec{K}^k, m^k) \downarrow$ in $\mathcal{E}_{T(w)}$. This holds for $k = 0$, as $(S, 0)$ is quiescent. For $k > 0$, we assume that it holds for $k - 1$ and must show $(\vec{K}^{k-1}, m^{k-1}) \overset{c^k}{\rightarrow} (\vec{K}^k, m^k) \downarrow$ in $\mathcal{E}_{T(w)}$. By construction, we have a path $(\vec{K}^{k-1}, 0) \overset{c^k}{\rightarrow} (\vec{K}^k, m^k - m^{k-1})$ in $\mathcal{E}_{T^k}$. By Proposition D.4.4 and Theorem D.4.3, we have $T^k = T^{<}(a_{m^{k-1}} \ldots a_{k-1}) \sqsubseteq T^{<}(w_{m^{k-1}}) \sqsubseteq T(w_{m^{k-1}}) = T(w)[m^k - 1]$, so by Proposition D.4.8 the path is in $\mathcal{E}_{T(w)[m^{k-1}]}$, and by Proposition D.4.9, we obtain our subgoal.

If the procedure accepts the input, then we are done by Proposition D.4.6. If it rejects, it suffices to show that $(\vec{K}, m^k)$ is quiescent in $\mathcal{E}_{T(w)}$ which by Proposition D.4.6 implies that there is no matching derivation. Since $T^m = T^{<}(w\#)$, we can apply Proposition D.4.5 to easily show $\mathcal{E}_{T^m} = \mathcal{E}_{T(w)}$, and we are done. □

Figure D.1 shows an example of a few iterations of the procedure applied to the program in Example 6.

In the next section we show that the above procedure can be performed using at most linear time and space. Linear space is easily seen to be obtained by observing that the table T^{k-1} is no longer needed once T^k has been computed. On the other hand, obtaining a linear time guarantee requires careful design: Computing each table T^k using the classical right-to-left algorithm would take linear time in each step, and hence quadratic time in total. In the following section, we show how to obtain the desired time complexity by computing each table incrementally from the previous one.

D.5 Algorithm

The streaming parsing procedure of Section D.4 can be performed in amortized time $O(|w|)$ (treating the program size as a constant). We assume that the program P is complete.

Our algorithm computes each prefix table T^k using a work set algorithm for computing fixed points. We save work by starting the computation from $T^{k-1}[m^{k-1}]$ instead of the empty table $\perp$. In order to avoid unnecessary processing, an auxiliary data structure is used to determine exactly those entries which have enough information available to be resolved. This structure itself can be maintained in constant time per step. Since at most $O(|w|)$ unique entries need to be resolved over the course of parsing w, this is also the time complexity of the algorithm.

The algorithm is presented in two parts in Figure D.2. Algorithm 1 (Parse) takes as input a #-terminated input stream and maintains two structures: A table structure T which incrementally gets updated to represent $T^{<}(u)$ for a varying substring $u = a_{m^{k-1}} ... a_{k-1}$; and a structure R which keeps track of reverse data dependencies between the entries in T. In each iteration, any resolved code prefix is returned and the corresponding table columns freed. The main work is done in Algorithm 2 (Fix) which updates T and R to represent the next prefix table and its reverse dependencies, respectively.

We will sketch the correctness proof and highlight important lemmas during the presentation. Detailed proofs can be found in the appendix.

Work Sets

Let T be a table such that $T \sqsubseteq T^{<}(u)$ for some prefix u. The *work set* $\Delta_u(T) \subseteq \mathsf{Index}$ consists of all indices of entries that can be updated to bring T closer to $T^{<}(u)$ by applying $F^{(u)}$:

$$\Delta_u(T) = \{(i,j) \mid T_{ij} \sqsubset F^{(u)}(T)_{ij}\}.$$

It should be clear that $T = T^{<}(u)$ iff $\Delta_u(T) = \emptyset$, and that for all $(p,q) \in \Delta_u(T)$, we still have $F^{(u)}_{pq}(T) \sqsubseteq T^{<}(u)$ for the updated table. In the following we show how $\Delta_u(F^{(u)}_{pq}(T))$ can be obtained from $\Delta_u(T)$ instead of recomputing it from scratch.

Dependencies

In order to determine the effect of table updates on the work set, we need to make some observations about the dependencies between table entries.

Consider an index (i,j) such that $A_i {\leftarrow} A_x[A_y, A_z]$ and $T_{ij} = \bot$. The index (i,j) cannot be in the work set for T unless either $T_{xj} = m$ and $T_{y(j+m)} \neq \bot$; or $T_{xj} = \mathsf{f}$ and $T_{zj} \neq \bot$. We say that (i,j) *conditions* on (x,j). The reverse condition map C^{-1} in Figure D.2 associates every row index x with the set of row indices $i \in C^{-1}_x$ such that (i,j) conditions on (x,j) for all j.

If $T_{xj} = m$ or $T_{xj} = \mathsf{f}$ then (i,j) is in the work set iff $T_{y(j+m)} \neq \bot$ or $T_{zj} \neq \bot$, respectively. In either case we say that (i,j) has a *dynamic dependency* on $(y, j+m)$ or (z,j), respectively. The dependency is *dynamic* since it varies based on the value of T_{xj}. The partial map $D : \mathsf{Table}(\mathsf{Res}) \times \mathsf{Index} \to \mathsf{Index}_\bot$ defined in Figure D.2 associates every index (i,j) with its unique dynamic dependency D^T_{ij} in table T. The dynamic dependency is undefined ($\bot$) if the condition is unresolved or if the corresponding expression g_i is simple.

By the observations above, we can reformulate the work set using dependencies:

Lemma D.5.1 (Work set characterization). *For all T we have*

$$\begin{aligned}\Delta_u(T) = \{(i,j) \in J_u \mid \ & T_{ij} = \bot \\ & \wedge (g_i \textit{ complex} \Rightarrow D^T_{ij} \neq \bot \neq T_{D^T_{ij}})\}\end{aligned}$$

Algorithm 1 (Parse).
In: $w = a_0 a_1 ... a_{|w|-1} \in \Sigma^* \#$.
Out: Code $c^0 c^1 ... c^{|w|-1}$, accept/reject.

```
$u := \varepsilon$
$T := \bot$
$\vec{K} := \varepsilon$
$R := (i, j) \mapsto \emptyset$
for $k \in \{1, ..., |w|\}$ do
    $u := u\ a_{k-1}$
    run Fix
    compute $(\vec{K}, 0) \xrightarrow{c} (\vec{K}', m') \downarrow$
    $c^n := c$
    $\vec{K} := \vec{K}'$
    $T := T[m']$
    $R := R[m']$
accept if $\vec{K} = \varepsilon$ else reject
```

Algorithm 2 (Fix).
Precondition:
$u = a_m ... a_{k-1}$
$T = T^{<}(a_m ... a_{k-2}) \wedge R = (D^T)^{-1}$
Postcondition:
$u = a_m ... a_{k-1}$
$T = T^{<}(u) \wedge R = (D^T)^{-1}$

```
$W := \{(i, |u| - 1) \mid g_i \text{ simple}\}$
while $W \neq \emptyset$ do
    let $(p, q) \in W$
    $T := F^{(u)}_{pq}(T)$
    $W := W \setminus \{(p, q)\} \cup R_{pq}$
    for $i' \in C_p^{-1}$ do
        let $(k, \ell) = D^T_{i'q}$
        $R_{k\ell} := R_{k\ell} \cup \{(i', q)\}$
        if $T_{k\ell} \neq \bot$ then
            $W := W \cup \{(i', q)\}$
```

Reverse condition map

$$
\begin{aligned}
C^{-1} &: |P| \to \mathbf{2}^{|P|} \\
C_x^{-1} &= \{i \in |P| \mid A_i \leftarrow A_x[A_y, A_z]\}
\end{aligned}
$$

Dynamic (reverse) dependency map

$$
D : \mathsf{Table} \times \mathsf{Index} \to \mathsf{Index}_\bot
$$

$$
D^T_{ij} = \begin{cases} (y, j + m) & \text{if } A_i \leftarrow A_x[A_y, A_z] \wedge T_{xj} = m \\ (z, j) & \text{if } A_i \leftarrow A_x[A_y, A_z] \wedge T_{xj} = \mathsf{f} \\ \bot & \text{otherwise} \end{cases}
$$

$$
(D^T)^{-1}_{k\ell} = \{(i, j) \mid D^T_{ij} = (k, \ell)\}
$$

Restrictions

$$
\begin{aligned}
T[m]_{ij} &= T_{i(m+j)} \\
R[m]_{k\ell} &= \{(i, j - m) \mid (i, j) \in R_{k(m+\ell)} \wedge j \geq m\}
\end{aligned}
$$

Figure D.2: Parsing algorithm.

Incremental Work Set Computation

When a table S is updated by computing $T = F^{(u)}_{pq}(S)$ for $(p,q) \in \Delta_u(S)$, Lemma D.5.1 tells us that the changes to the work set can be characterized by considering the entries (i,j) for which one or more of the values D^T_{ij} and $T_{D^T_{ij}}$ differ from D^S_{ij} and $S_{D^S_{ij}}$, respectively.

An important observation is that the dependency map only gets more defined as we go from S to T:

Lemma D.5.2 (Dependency monotonicity). *If $T \sqsubseteq T'$, then for all $(i,j) \in \mathsf{Index}$, we have $D^T_{ij} \sqsubseteq D^{T'}_{ij}$.*

Using this and the fact that $S \sqsubseteq T$, it is easy to show that we must have $\Delta_u(T) \supseteq \Delta_u(S) \setminus \{(p,q)\}$. Furthermore, we observe that $(i,j) \in \Delta_u(T) \setminus (\Delta_u(S) \setminus \{(p,q)\})$ iff

1. $D^S_{ij} \sqsubset D^T_{ij}$ and $T_{D^T_{ij}} \neq \bot$; or
2. $D^S_{ij} = D^T_{ij} \neq \bot$ and $S_{D^S_{ij}} \sqsubset T_{D^T_{ij}}$.

Since the second case can only be satisfied when $D^T_{ij} = (p,q)$, it is completely characterized by the reverse dependency set $(D^T)^{-1}_{pq}$, defined in Figure D.2. The first case is when (i,j) conditions on (p,q) (equivalent to $D^S_{ij} \sqsubset D^T_{ij}$) and $T_{D^T_{ij}} \neq \bot$. The entries satisfying the former are completely characterized by the reverse condition map:

Lemma D.5.3 (Dependency difference). *Let $S \in \mathsf{Table}(\mathsf{Res})$ such that $S \sqsubseteq T^{<}(u)$ and $(p,q) \in \Delta_u(S)$, and define $T = F^{(u)}_{pq}(S)$. Then $\{(i,j) \mid D^S_{ij} \sqsubset D^T_{ij}\} = C^{-1}_p \times \{q\}$.*

By Lemmas D.5.1, D.5.2 and D.5.3, we obtain the following incremental characterization of the work set:

Lemma D.5.4 (Work set update). *Let $S \sqsubseteq F^{(u)}(S) \sqsubseteq T^{<}(u)$, $(p,q) \in \Delta_u(S)$ and $T = F^{(u)}_{pq}(S)$. Then*

$$\begin{aligned}\Delta_u(T) = \Delta_u(S) &\setminus \{(p,q)\} \\ &\cup (D^S)^{-1}_{pq} \\ &\cup \{(i',q) \mid i' \in C^{-1}_p \wedge \bot \neq T_{D^T_{i'q}}\}\end{aligned}$$

The extra premise $S \sqsubseteq F^{(u)}(S)$ says that every entry in S must be a consequence of the rules encoded by $F^{(u)}$, and can easily be shown to be an invariant of our algorithm.

Reverse dependency map lookups $(D^T)^{-1}_{pq}$ cannot easily be computed efficiently. To accomodate efficient evaluation of these lookups, the algorithm maintains a data structure R to represent $(D^T)^{-1}$. The following Lemma shows that the loop 6-10 will reestablish the invariant that $R = (D^T)^{-1}$:

Lemma D.5.5 (Dependency update). *Let $S \sqsubseteq T^<(u)$, $(p, q) \in \Delta_u(S)$ and $T = F^{(u)}_{pq}(S)$. Then for all $(k, \ell) \in \mathsf{Index}$, we have $(D^T)^{-1}_{k\ell} = (D^S)^{-1}_{k\ell} \cup \{(i', q) \mid i' \in C^{-1}_p \wedge (k, \ell) = D^T_{i'q}\}$.*

Correctness

Theorem D.5.6 (Correctness of Fix). *If the precondition of Fix holds, then the postcondition holds upon termination.*

Proof sketch. We first remark that the algorithm never attempts to perform an undefined action. It suffices to check that line 3 is always well-defined, and that Lemma D.5.3 implies that the right of the equation in line 7 is always resolved.

The outer loop maintains that $R = (D^T)^{-1}$ and $W = \Delta_u(T)$. Initially, only the entries in the last column which are associated with simple expressions can be updated. If S is the state of T at the beginning of an iteration of loop 2-10, then at the end of the iteration T will have the form of the right hand side of Lemma D.5.4. When the loop terminates we have $W = \Delta_u(T) = \emptyset$, so $T = T^<(u)$. □

Theorem D.5.7 (Correctness of Parse). *The algorithm Parse performs the streaming parsing procedure of Section D.4.*

Proof sketch. After executing lines 1-4, we verify that $R = (D^T)^{-1}$, and that for $k = 0$:

$$T = T^<(a_{m^k} ... a_{k-1}), \qquad \vec{K} = \vec{K}^k, \qquad u = a_{m^k} ... a_{k-1}$$

The loop maintains the invariant: When entering the loop, we increment k and thus have $R = (D^T)^{-1}$ and

$$T = T^<(a_{m^{k-1}} ... a_{k-2}), \qquad \vec{K} = \vec{K}^{k-1}, \qquad u = a_{m^{k-1} ... a_{k-2}}$$

After the assignment to u, we have $u = a_{m^{k-1}} ... a_{k-1}$. By running Fix, we then obtain $T = T^<(a_{m^{k-1}} ... a_{k-1}) = T^k$. By assumption that P is complete, line 8 is computable, and we obtain

$$\vec{K}' = \vec{K}^k \qquad c = c^k \qquad m' = m^k - m^{k-1}$$

The last updates in the loop thus reestablishes the invariant. □

Complexity

We give an informal argument for the linear time complexity. Let $d \in \omega$ be the constant from Definition 38 limiting the number of stack symbols considered when resolving choices.

It can be shown that the three sets on the right hand side of the equation in Lemma D.5.4 are pairwise disjoint; likewise for Lemma D.5.5. We thus never add the same element twice to W and R, meaning that they can be represented using list data structures, ensuring that all single-element operations are constant time.

The complexity argument is a simple aggregate analysis. To see that Parse runs in linear time, we observe that the work set invariant ensures that we execute at most $O(|u|)$ iterations of the loop 2-10 in Fix. Since we only add unprocessed elements to the work list, and no element is added twice, the total number of append operations performed in lines 5 and 10 is also $O(|u|)$. The same reasoning applies for the total number of append operations in line 8. The remaining operations in Fix are constant time.

Line 8 in Parse computes an expansion of aggregate length $O(mn)$. For each expansion transition, we use at most d steps to resolve choices, and we thus obtain a bound of $O(dmn)$.

The restriction operator $T[m]$ can be performed in constant time by moving a pointer. The restriction of the reverse dependency map $R[m]$ can be implemented in constant time by storing the offset and lazily performing the offset calculation $j - m$ and filtering by $j \leq m$ on lookup.

D.6 Evaluation

We have developed a simple prototype implementation for the purpose of measuring how the number of columns grow and shrink as the parser proceeds, which gives an indication of both its memory usage and its ability to resolve choices. The evaluation also reveals parts of the design which will require further engineering in order to obtain an efficient implementation. We have not yet developed an implementation optimized for speed, so a comparative performance comparison with other tools is reserved for future work.

We consider three programs: a) a simplified JSON parser, b) a simplified parser for the fragment of statements and arithmetic expressions of a toy programming language, c) a tail-recursive program demonstrating a pathological worst-case.

All programs are presented as PEGs for readability. Nonterminals are underlined, terminals are written in `typewriter` and a character class [a...z] is short for `a`/`b`/.../`z`.

JSON Parser We have written a simple JSON parser based on a simplification of the ECMA 404 specification[3] and taking advantage of the repetition operator of PEG. To keep the presentation uncluttered, we have left out handling of whitespace.

$$
\begin{array}{rcl}
\underline{\mathit{object}} & \leftarrow & \texttt{\{}\,\underline{\mathit{members}}\,\texttt{\}} \\
\underline{\mathit{members}} & \leftarrow & \underline{\mathit{pair}}(\texttt{,}\,\underline{\mathit{pair}})^{*}/\epsilon \\
\underline{\mathit{pair}} & \leftarrow & \underline{\mathit{string}}\,\texttt{:}\,\underline{\mathit{value}} \\
\underline{\mathit{array}} & \leftarrow & \texttt{[}\,\underline{\mathit{elements}}\,\texttt{]} \\
\underline{\mathit{elements}} & \leftarrow & \underline{\mathit{value}}(\texttt{,}\,\underline{\mathit{value}})^{*}/\epsilon \\
\underline{\mathit{value}} & \leftarrow & \underline{\mathit{string}}\,/\,\underline{\mathit{object}}\,/\,\underline{\mathit{number}}\,/\,\underline{\mathit{array}} \\
 & & /\texttt{true}/\texttt{false}/\texttt{null} \\
\underline{\mathit{string}} & \leftarrow & \texttt{"}[\mathsf{a...z}]^{*}\texttt{"} \\
\underline{\mathit{number}} & \leftarrow & \underline{\mathit{int}}(\underline{\mathit{frac}}\,/\epsilon)(\underline{\mathit{exp}}\,/\epsilon) \\
\underline{\mathit{int}} & \leftarrow & [\mathsf{1...9}]\,\underline{\mathit{digits}}\,/\texttt{-}[\mathsf{1...9}]\,\underline{\mathit{digits}}\,/\texttt{-}[\mathsf{0...9}]/[\mathsf{0...9}] \\
\underline{\mathit{frac}} & \leftarrow & \texttt{.}\,\underline{\mathit{digits}} \\
\underline{\mathit{exp}} & \leftarrow & \underline{\mathit{e}}\,\underline{\mathit{digits}} \\
\underline{\mathit{digits}} & \leftarrow & [\mathsf{0...9}][\mathsf{0...9}]^{*} \\
\underline{\mathit{e}} & \leftarrow & \texttt{e+/e-/e/E+/E-/E}
\end{array}
$$

The desugared program contains 158 rules. We ran the program on a 364 byte JSON input with several nesting levels and syntactic constructs exercising all rules of the grammar. The resulting parse code is computed in 3530 expansion steps based on the computed table information.

We would like to get an idea of how varying values of the speculation constant d affects the amount of memory consumed and also the amount of work performed. Recall that d specifies the number of stack symbols considered when determining whether a branch must succeed on all viable expansions. The results for the range 0 to 12 are summarized in the following table:

[3] http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-404.pdf

d	max cols (max 365)	non-imm. entries (max 23360)	spec. steps (rel. to 3530)	visited
0	362	23348 (99.95%)	0 (0.00%)	2866
1	229	23248 (99.52%)	6 (0.17%)	2876
2	229	23248 (99.52%)	9 (0.25%)	2876
3	10	19321 (82.71%)	271 (7.68%)	3116
4	10	19321 (82.71%)	283 (8.02%)	3117
5	10	19284 (82.55%)	295 (8.36%)	3121
6	10	19200 (82.19%)	312 (8.84%)	3134
7	10	19200 (82.19%)	321 (9.09%)	3134
8	2	18936 (81.06%)	419 (11.87%)	3162
9	2	18921 (81.00%)	431 (12.21%)	3173
10	2	18921 (81.00%)	442 (12.52%)	3173
11	2	18789 (80.43%)	453 (12.83%)	3173
12	2	18789 (80.43%)	453 (12.83%)	3173

The second column shows the maximum number of columns stored at any point. The worst case is $364 + 1 = 365$. We observe that $d = 8$ results in just two columns needing to be stored in memory.

The third column measures the potential work saved as d is increased. To explain it, we introduce the notion of an *immediate rule*, which is either simple, or of the form $A \leftarrow A[B, C]$ where A and B are immediate and either $C \leftarrow \epsilon$ or $C \leftarrow \mathtt{f}$. An entry T_{ij} where A_i is immediate is always resolved upon reading symbol j, and can thus be precomputed and looked up based on the symbol. The real run-time cost is therefore the number of computed non-immediate entries, which is shown in the third column together with the percentage compared to the worst case. The benchmark shows that for $d \geq 8$, an average of 52 complex entries must be resolved for each input symbol. This may turn out to be an issue for scalability, as the number of non-immediate entries can be expected to be proportional to the program size.

The fourth column is the number of steps spent evaluating the $(\vec{K}, j)$ $\textbf{fails}_n$ predicate, and the relative number compared to the number of expansion steps. For this particular program, the overhead is seen to be very small compared to the reduction in computed entries and the fact that parsing proceeds in practically constant memory.

The last column shows the total number of unique table entries visited by the expansion. This is much smaller than the number of entries actually computed, so there is ample room for optimization, e.g. by integration between the expansion process and the table computation in order to compute only the entries that are needed.

Statement/Expression Parser The following is inspired by an example from a paper on ALL(*) [15]. The program parses a sequence of statements, each terminated by semicolon, with the whole sequence terminated by a single dot representing an end-of-program token. Each statement is either a single arithmetic expression or an assignment.

$$\begin{array}{rcl}
\underline{prog} & \leftarrow & \underline{stat}\,\underline{stat}^{*}\ \texttt{.} \\
\underline{stat} & \leftarrow & \underline{sum}\,\texttt{=}\,\underline{sum}\ \texttt{;}\ /\ \underline{sum}\ \texttt{;} \\
\underline{sum} & \leftarrow & \underline{product}\,\texttt{+}\,\underline{sum}\ /\ \underline{product} \\
\underline{product} & \leftarrow & \underline{factor}\,\texttt{*}\,\underline{product}\ /\ \underline{factor} \\
\underline{factor} & \leftarrow & \underline{id}\,\texttt{(}\,\underline{sum}\,\texttt{)}\ /\ \texttt{(}\,\underline{sum}\,\texttt{)}\ /\ \underline{id} \\
\underline{id} & \leftarrow & [\mathsf{a}...\mathsf{z}]\,[\mathsf{a}...\mathsf{z}]^{*}
\end{array}$$

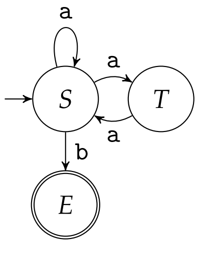
Top-down parsing of infix expressions may require unbounded buffering of the left operand, as the operator itself arrives later in the input stream. The following shows an input string, and below each symbol is the size of the parse table right after its consumption:

a_j	z	=	f	(	z	)	;	x	=	x	+	y	*	y	*	y	;	g	(	x	)	;	.	#
size	1	0	1	2	3	4	0	1	0	1	0	1	2	3	4	5	0	1	2	3	4	0	0	1

We are not concerned with the speculation constant; assume that it is unbounded. The example demonstrates how the method adapts the table size as input is consumed. Note that ; and = resolves the sum expression currently being parsed, truncating the table, and also that the left operand of the + symbol is correctly resolved, while the * expression must be buffered.

Ambiguous Tail-Recursive Programs Any non-deterministic finite automaton (NFA) can be interpreted as a PEG program by assigning a nonterminal to each state, and for each state q with transitions $q \xrightarrow{a_1} q_1, ..., q \xrightarrow{a_n} q_n$ creating a rule $\underline{q} \leftarrow a_1\,\underline{q_1}\,/\,...\,/\,a_n\,\underline{q_n}$. The ordering of transitions is significant and defines a disambiguation priority. The final state q^f is assumed to have no transitions, and is given the rule $\underline{q^f} \leftarrow \epsilon$.

If the NFA contains no ε-loops then its language will coincide with that of its PEG encoding, which is a complete program implementing a backtracking depth-first search for an accepting path. The following shows a simple example of an NFA and its prioritized interpretation as a PEG:



$$\begin{array}{l}
\underline{S} \leftarrow \texttt{a}\,\underline{S}\,/\,\texttt{a}\,\underline{T}\,/\,\texttt{b}\,\underline{U} \\
\underline{T} \leftarrow \texttt{a}\,\underline{S} \\
\underline{E} \leftarrow \epsilon
\end{array}$$

$$\begin{array}{ll}
\hline
S \leftarrow P[E,Q] & P \leftarrow A[S,F] \\
T \leftarrow A[S,F] & Q \leftarrow V[E,W] \\
E \leftarrow \epsilon & V \leftarrow A[T,F] \\
F \leftarrow \texttt{f} & W \leftarrow B[E,F] \\
B \leftarrow \texttt{b} & \\
A \leftarrow \texttt{a} &
\end{array}$$

The NFA is ambiguous, as any string of the form $a^{n+2}b$, $n \geq 0$, can be matched by more than one path from S to E. The priority enforced by the program dictates that T is never invoked, as the production aS *covers* the production aT, meaning that every string accepted by the latter is also accepted by the former, which has higher priority in the choice.

The example triggers worst-case behavior for our method, which fails to detect coverage regardless of the speculation bound, resulting in a table size proportional to the input length. This is obviously suboptimal, as any regular language can be recognized in constant space.

The problem is in the tail recursion; the desugared program has every recursive call occur as a condition which remains unresolved until the end-of-marker input has been seen. The analysis is oblivious to coverage, and thus fails to detect that T can never be on a viable expansion until the very end.

D.7 Discussion

We discuss our method in the context of the work of others, and point out directions for future work.

The workset algorithm is an instance of the scheme of *chaotic iteration* [4] for computing limits of finite iterations of monotone functions. Our parsing formalism goes back to the TS/TDPL formalism introduced by Birman and Ullman [3] and later generalized to GTDPL by Aho and Ullman [1]. They also present the linear-time tabular parsing technique and show that GTDPL can express recognizers for all deterministic context-free languages, including all deterministic LR-class languages. On the other hand, there are context-free languages that cannot be recognised by GTDPL, as general context-free parsing is super-linear [11]. Ford's Parsing Expression Grammars (PEG) [6] have the same recognition power as GTDPL, albeit using a larger set of operators which arguably are better suited for practical use.

Packrat parsing [5], is a direct implementation of the PEG operational semantics with memoization. It can be viewed as "sparse" tabular parsing where only the entries encountered on a depth-first search for an expansion are computed. Our evaluation shows that PTP computes a very large portion of the table. Some of this overhead is unavoidable, as the dynamic analysis relies on the exploration of both branches of the choice currently being resolved, but most of the computed entries are never considered by the expansion process. A closer integration of expansion and table computation inspired by Packrat may turn out to be a rewarding implementation strategy.

Heuristic approaches include Kuramitsu's Elastic Packrat algorithm [10] and Redziejowski's parser generator *Mouse* [17], both of which are Packrat implementations using memory bounded by a configurable constant. The former uses a sliding window to limit the number of stored table columns, and the latter limits the number of memoized calls per nonterminal. Both approaches risk triggering exponential behavior when backtracking exceeds the bounds of their configured constants, which however seems rare in practice. A disadvantage of heuristic memory reductions is that they have to store the full input string until the full parse is resolved, because they cannot guarantee that the parser will not backtrack.

Packrat With Static Cut Annotations

Mizushima, Maeda and Yamaguchi observes that when Packrat has no failure continuations on the stack, all table columns whose indices are less than the index of the current symbol can be removed from memory. To increase the likelihood of this, they extend PEG with cut operators à la Prolog to "cut away" failure continuations, and also devise a technique for sound automatic cut insertion, i.e. without changing the recognized language [14]. Manually inserted cuts yield significant reductions in heap usage and increases in throughput, but automatic cut insertion seems to miss several opportunities for optimization. Redziejowski further develops the theory of cut insertion and identifies sufficient conditions for soundness, but notes that automation is difficult: *"It appears that finding cut points in non-LL(1) grammars must to a large extent be done manually"* [18].

The method of Mizushima et al. is subsumed by PTP. An empty stack of failure continuations corresponds to the case where the condition A in a top-level choice $A[B, C]$ is resolved. Insertion of cuts is the same as refactoring the grammar using the GTDPL operator $A[B, C]$, which is the cut operator $A \uparrow B/C$ of Mizushima et al. in disguise. Increasing the speculation bound can achieve constant memory use without requiring any refactoring of the program.

Cost vs Benefit of Memoization

Several authors argue that the cost of saving parse results outweighs its benefits in practice [2, 9]. The PEG implementation for the Lua language [9] uses a backtracking parsing machine instead of Packrat in order to avoid paying the memory cost [12]. Becket and Somogyi compares the performances of Packrat parsers with and without memoization using a parser for the Java language as benchmark [2]. Their results show that full memoization is always much slower than plain recursive descent parsing, which never triggered the exponential worst case in any of their tests. On the other hand, memoizing only a few selected nonterminals *may* yield in a speedup, suggesting that memoization does not serve as a performance optimization, but as a safeguard against pathological worst-case scenarios which are rare in practice. However, another experiment by Redziejowki on PEG parsers for the C language show a significant overhead due to backtracking. This could not be completely eliminated by memoizing a limited number of nonterminals, but required manual rewriting of the grammar based on knowledge from the benchmark results [16].

Our technique uses full tabulation rather than memoization, but the results still apply to suggest that a direct implementation will likely be slower than plain recursive descent parsers on common inputs and carefully constructed grammars. However, ad-hoc parsers cannot be expected to be constructed in such an optimal way, and thus may need memoization to prevent triggering worst-case behavior. Furthermore, our best-case memory usage—which is bounded—outperforms recursive descent parsers which must store the complete input string in case of backtracking. This is crucial in the case of huge or infinite input strings which cannot fit in memory, e.g. logging data, streaming protocols or very large data files.

Parsing Using Regular Expressions

Medeiros, Mascarenhas and Ierusalimschy embed backtracking regular expression matching in PEG [13]. In fact, every regular expression corresponds to a right-regular context-free grammar[4], and one can easily check that interpreting this grammar as a PEG yields its backtracking matching semantics. Interestingly, the PEG encoding of ambiguous regular expressions make our method exhibit worst-case behavior with regards to streaming and memory usage, as the dynamic analysis is oblivous to detection of *coverage*. Coverage is undecidable for PEG in general, but is decidable for right-regular grammars [8].

Grathwohl, Henglein and Rasmussen give a streaming regular expression parsing technique which supports both approximate and optimal coverage analysis [8]. With Søholm and Tørholm they develop *Kleenex*, which compiles grammars for regular languages into high-performance streaming parsers with backtracking semantics [7]. Since PEGs combine lexical and syntactic analysis, they can be expected to contain many regular fragments. Perhaps the technique of Kleenex can be combined with PTP to obtain better streaming behavior for these.

D.8 Conclusion

We have presented PTP, a new streaming execution model for the TDPL family of recursive descent parsers with limited backtracking, together with a linear-time algorithm for computing progressive tables and a dynamic analysis for improving the streaming behavior of the resulting parsers. We have also demonstrated that parsers for both LL and non-LL languages automati-

[4] Contains only productions of the form $A \to \varepsilon$ and $A \to aB$, corresponding 1-1 to the transitions of an NFA.

cally adapt their memory usage based on the amount of lookahead necessary to resolve choices.

A practical performance-oriented implementation will be crucial in order to get a better idea of the applicability of our method. Our prototype evaluation shows that a substantial amount of the computed table entries are never used, so future work should focus on minimizing this overhead.

We believe that our method will be useful in scenarios where a streaming parse is desired, either because all of the input is not yet available, or because it is too large to be stored in memory at once. Possible applications include read-eval-print-loops, implementation of streaming protocols and processing of huge structured data files.

D.9 Proofs

Proposition D.2.5 (Linear tree complexity). *Fix a program P. For all $A \in V$ and $u, v \in \Sigma^*$ and derivations $\mathcal{D} :: (A, u) \Rightarrow_P v$ we have $|\mathcal{T}(\mathcal{D})| = O(|u|)$.*

Proof. Observe that $\mathcal{D}$ cannot contain a strict subderivation for the subject (A, u), as determinism would imply that $\mathcal{D}$ would be infinite.

We show by induction on $|u| - |v|$ that $|\mathcal{T}(\mathcal{D})| \leq 2^{|P|}(|u| - |v|)$. □

Tabulation of Operational Semantics

Lemma D.3.1 (Lower bound iteration). *If $T \sqsubseteq \operatorname{lfp} F$, then $\operatorname{lfp} F = \bigsqcup_n F^n(T)$.*

Proof. We prove both directions of the equality.

Claim: $\operatorname{lfp} F \sqsubseteq \bigsqcup_n F^n(T)$. We first remark that by definition, $\operatorname{lfp} F = \bigsqcup_n F^n(\bot)$ is the least upper bound of $\{F^n(\bot)\}$. Observe that for all n we have $F^n(\bot) \sqsubseteq F^n(T) \sqsubseteq \bigsqcup_n F^n(T)$. Indeed, the last inequality follows by definition of least upper bounds. The former holds by induction, since we have $\bot \sqsubseteq T$ and by monotonicity of F, $F^m(\bot) \sqsubseteq F^m(T)$ implies $F^{m+1}(\bot) \sqsubseteq F^{m+1}(T)$. Since we have shown that $\bigsqcup_n F^n(T)$ is an upper bound of $\{F^n(\bot)\}$, we have $\operatorname{lfp} F \sqsubseteq \bigsqcup_n F^n(T)$.

Claim: $\bigsqcup_n F^n(T) \sqsubseteq \operatorname{lfp} F$. Observe that for all n we have $F^n(T) \sqsubseteq \operatorname{lfp} F$. Indeed we have $T \sqsubseteq \operatorname{lfp} F$ by assumption, and by monotonicity $F^m(T) \sqsubseteq \operatorname{lfp} F$ implies $F^{m+1}(T) \sqsubseteq F(\operatorname{lfp} F) = \operatorname{lfp} F$. Since we have shown that $\operatorname{lfp} F$ is an upper bound of $\{F^n(T)\}$ it follows that $\bigsqcup_n F^n(T) \sqsubseteq \operatorname{lfp} F$. □

Theorem D.3.2 (Fundamental theorem). *Let $u \in \Sigma^*$ and consider $T(u)$ as defined above. For all $(i, j) \in \mathsf{Index}$:*

1. *$j \leq |u|$ and $T(u)_{ij} = \mathsf{f}$ iff $(A_i, u_j) \Rightarrow_P \mathsf{f}$; and*

2. *$j \leq |u|$ and $T(u)_{ij} = m \in \omega$ iff $(A_i, u_j) \Rightarrow_P u_{j+m}$; and*

3. *$j \leq |u|$ and $T(u)_{ij} = \bot$ iff $(A_i, u_j) \not\Rightarrow_P$;*

4. *if $j > |u|$ then $T_{ij} = T_{i|u|}$*

The converse also holds: for any T satisfying the above, we have $T = T(u)$.

Proof. We start by proving Property 4. Let $w = u\#^{\omega}$ and observe that for all (i, j) where $j > |u|$ we have $w_j = w_{|u|}$, and it follows that $F^u(\perp)_{ij} = F^u(\perp)_{i|u|} \in \{\perp, \mathsf{f}\}$. By a simple induction we obtain that $j > |u|$ implies $(F^u)^k(\perp)_{ij} = (F^u)^k(\perp)_{i|u|}$ for all $k \geq 0$. Therefore $j \geq |u|$ implies that $T(u)_{ij} = \bigsqcup_k (F^u)^k(\perp)_{ij} = \bigsqcup_k (F^u)^k(\perp)_{i|u|} = T(u)_{i|u|}$.

Before proving the remaining, we make the claim that for all (i, j) such that $j \leq |u|$ we have

1. If $(A_i, u_j) \Rightarrow u_{j+m}$ then $\exists k. (F^u)^k(\perp)_{ij} = m$.

2. If $(A_i, u_j) \Rightarrow \mathsf{f}$ then $\exists k. (F^u)^k(\perp)_{ij} = \mathsf{f}$.

3. If $(A_i, u_j) \not\Rightarrow$ then $\forall k. (F^u)^k(\perp)_{ij} = \perp$.

If the claim holds, then one direction of Properties 1,2,3 follow. For example, if $(A_i, u_j) \Rightarrow u_{j+m}$ then there is a k such that $m = (F^u)^k(\perp)_{ij} \sqsubseteq \bigsqcup_k (F^u)^k(\perp)_{ij} = T(u)_{ij}$. For the converse directions we use the fact that for all (i, j) we have $(\exists m. (A_i, u_j) \Rightarrow u_{j+m}) \vee ((A_i, u_j) \Rightarrow \mathsf{f}) \vee ((A_i, u_j) \not\Rightarrow)$. Using the previous claims, the value of T_{ij} will be in contradiction with all but one of the three disjuncts.

The first two claims follow by induction on derivations. In the inductive cases we use monotonicity of F^u to pick a large enough k. For the third claim we prove that $\exists k. (F^u)^k(\perp)_{ij} = \mathsf{f}/m$ then $(A_i, u_j) \Rightarrow \mathsf{f}/u_{j+m}$ by induction on k. The contrapositive of this matches the third claim.

Using determinacy of the parsing relation, it is easily seen that the properties of the Theorem uniquely determines $T(u)$. □

Prefix Tables

Theorem D.4.3 (Approximation)**.** *For all $u \in \Sigma^*$, we have $T^<(u) \sqsubseteq T^{\sqcap}(u)$. In particular, if $T^<(u)_{ij} = m$ or $T^<(u)_{ij} = \mathsf{f}$, then $\forall v. T(uv)_{ij} = m$ or $\forall v. T(uv)_{ij} = \mathsf{f}$, respectively.*

Proof. Let $u \in \Sigma^*$. It suffices to show that for any $v \in \Sigma^*$, we have $T^<(u) \sqsubseteq T(uv)$.

We first remark that for all $J \subseteq \mathsf{Index}$ and $T \in \mathsf{Table}$ we have $F^u_J(T) \sqsubseteq F^u(T)$. Furthermore, for all $v \in \Sigma^*$ we have $F^u_{J_u}(T) = F^{uv}_{J_u}(T)$. By these two remarks, we obtain via induction that for all $n \geq 0$, we have $(F^u_{J_u})^n(\perp) = (F^{uv}_{J_u})^n(\perp) \sqsubseteq (F^{uv})^n(\perp) \sqsubseteq \bigsqcup_n (F^{uv})^n(\perp)$. Hence $T(uv)$ is an upper bound of $\{(F^u_{J_u})^n(\perp) \mid n \geq 0\}$, but since $T^<(u)$ is the *least* upper bound of this set, we obtain $T^<(u) \sqsubseteq T(uv)$. □

Correctness of algorithm

Lemma D.5.1 (Work set characterization). *For all T we have*

$$\begin{aligned}\Delta_u(T) = \{(i,j) \in J_u \mid\ & T_{ij} = \bot \\ & \wedge (g_i \textit{ complex} \Rightarrow D^T_{ij} \neq \bot \neq T_{D^T_{ij}})\}\end{aligned}$$

Proof. Let $(i,j) \in \mathsf{Index}$. For the forward direction, assume $T_{ij} \sqsubset F^{(u)}(T)_{ij}$. Then $T_{ij} = \bot$, and since $T_{ij} \neq F^{(u)}(T)_{ij} = F_{J_u}(T)_{ij}$, we must have $(i,j) \in J_u$. It remains to prove the implication. Assume $A_i{\leftarrow}A_x[A_y, A_z]$. By cases on the definition of F^u and the fact $F^u(T)_{ij} \neq \bot$, we have three possible cases: $T_{xj} = m$ and $T_{y(j+m)} = m'$; or $T_{xj} = m$ and $T_{y(j+m)} = \mathsf{f}$; or $T_{xj} = \mathsf{f}$ and $T_{zj} \neq \bot$. In the first two cases we have $D^T_{ij} = (y, j+m)$ and $T_{y(j+m)} \neq \bot$. In the last case we have $D^T_{ij} = (z,j)$ and $T_{zj} = \mathsf{f} \neq \bot$, and we are done.

For the converse direction, assume $(i,j) \in J_u$, $T_{ij} = \bot$ and (g_i complex $\Rightarrow$ $D^T_{ij} \neq \bot \neq T_{D^T_{ij}}$). Since $(i,j) \in J_u$, we have $F^{(u)}(T)_{ij} = F^u(T)_{ij}$ and we need to show $F^u(T)_{ij} \neq \bot$. If g_i is simple it is easy to check that $F^u(T)_{ij} \neq \bot$ in all cases. If $A_i{\leftarrow}A_x[A_y, A_z]$, then by assumption we have $D^T_{ij} \neq \bot \neq T_{D^T_{ij}}$. We have three possible cases which are handled analogously. For the first case $T_{xj} = m$, $D^T_{ij} = (y, j+m)$ and $T_{y(j+m)} = m'$ for some m, m'. By definition $F^u(T)_{ij} = m + m' \neq \bot$, and we are done. □

Dependency monotonicity says that the dependency map seen as a table operator is monotone.

Lemma D.5.2 (Dependency monotonicity). *If $T \sqsubseteq T'$, then for all $(i,j) \in \mathsf{Index}$, we have $D^T_{ij} \sqsubseteq D^{T'}_{ij}$.*

Proof. Let $(i,j) \in \mathsf{Index}$ and assume $D^T_{ij} \neq \bot$ (the case $D^T_{ij} = \bot$ is trivial). Then $A_i{\leftarrow}A_x[A_y, A_z]$ and either $D^T_{ij} = (y, j+m)$ and $T_{xj} = m$; or $D^T_{ij} = (z,j)$ and $t_{xj} = \mathsf{f}$. In the first case we get $T_{xj} \sqsubseteq T'_{xj} = m$ by assumption, so $D^{T'}_{ij} = (y, j+m) = D^T_{ij}$. The latter case is analogous. □

The following shows that upon updating a single entry in a table, the set of dependencies that will go from being undefined to being defined can be determined statically.

Lemma D.5.3 (Dependency difference). *Let $S \in \mathsf{Table}(\mathsf{Res})$ such that $S \sqsubseteq T^{<}(u)$ and $(p,q) \in \Delta_u(S)$, and define $T = F^{(u)}_{pq}(S)$. Then $\{(i,j) \mid D^S_{ij} \sqsubset D^T_{ij}\} = C^{-1}_p \times \{q\}$.*

Proof. For the converse direction, let $i \in C^{-1}_p$, which implies $A_i{\leftarrow}A_p[A_y, A_z]$. Since $S_{pq} = \bot \neq T_{pq}$, we must have $\bot = D^S_{iq} \sqsubset D^T_{iq} \neq \bot$, and we are done.

For the forward direction, assume $\bot = D^S_{ij} \sqsubset D^T_{ij} \neq \bot$. By the latter equality it follows that $A_i \leftarrow A_x[A_y, A_z]$ where $C_i = x$, so $i \in C_x^{-1}$. By the first equality and definition, we have $S_{xj} = \bot$; by the latter equality we have $T_{xj} \neq \bot$. But then $S_{xj} \sqsubset T_{xj}$, which implies $(x, j) = (p, q)$. Since $i \in C_p^{-1}$ and $j = q$, we are done. □

We can now prove the main lemma of the correctness proof:

Lemma D.5.4 (Work set update). *Let $S \sqsubseteq F^{(u)}(S) \sqsubseteq T^{<}(u)$, $(p, q) \in \Delta_u(S)$ and $T = F^{(u)}_{pq}(S)$. Then*

$$\begin{aligned} \Delta_u(T) = {} & \Delta_u(S) \setminus \{(p,q)\} \\ & \cup (D^S)^{-1}_{pq} \\ & \cup \{(i', q) \mid i' \in C_p^{-1} \wedge \bot \neq T_{D^T_{i'q}}\} \end{aligned}$$

Proof. We initially remark that $S \sqsubseteq T$ by definition of T, and hence that $D^S \sqsubseteq D^T$ by Lemma D.5.2. Since $(p, q) \in \Delta_u(S)$ we have $S_{pq} \sqsubset F^{(u)}(S)_{pq} = T_{pq}$, so in particular $T_{pq} \neq \bot$.

Forward direction. Assume $(i, j) \in \Delta_u(T)$. By Lemma D.5.1 we have $T_{ij} = \bot$ and (g_i complex $\Rightarrow D^T_{ij} \neq \bot \neq T_{D^T_{ij}}$). So, $(i, j) \neq (p, q)$ and $S_{ij} = \bot$.

Assume g_i complex. Then $D^T_{ij} \neq \bot \neq T_{D^T_{ij}}$. Since $D^S_{ij} \sqsubseteq D^T_{ij}$, we have either (a) $D^S_{ij} = \bot \sqsubset D^T_{ij}$; or (b) $D^S_{ij} = D^T_{ij} \neq \bot$.

In case (a), we apply Lemma D.5.3 to obtain $(i, j) \in C_p^{-1} \times \{q\}$, which implies $(i, j) \in \{(i', q) \mid i' \in C_p^{-1} \wedge \bot \neq T_{D^T_{i'q}}\}$, and we are done.

In case (b) we consider the subcases (α) $S_{D^S_{ij}} = \bot \sqsubset T_{D^T_{ij}}$; and ($\beta$) $S_{D^T_{ij}} = T_{D^T_{ij}} \neq \bot$. In subcase ($\alpha$), we must have $D^S_{ij} = (p, q)$ so $(i, j) \in (D^S)^{-1}_{pq}$ and we are done. In subcase (β), observe that we have $D^S_{ij} \neq \bot \neq S_{D^S_{ij}}$ which by Lemma D.5.1 implies $(i, j) \in \Delta_u(S) \setminus \{(p, q)\}$.

Converse direction. Assume that (i, j) is in the set on the right hand side. By Lemma D.5.1 it suffices to show $T_{ij} = \bot$ and (g_i complex $\Rightarrow D^T_{ij} \neq \bot \neq T_{D^T_{ij}}$). We have three possible cases:

Case $(i, j) \in \Delta_u(S) \setminus \{(p, q)\}$. Since $(i, j) \neq (p, q)$ we have $S_{ij} = T_{ij}$ by definition of T. By Lemma D.5.1 we obtain $S_{ij} = \bot = T_{ij}$ and (g_i complex $\Rightarrow D^S_{ij} \neq \bot \neq S_{D^S_{ij}}$). Assuming g_i complex, we thus have $D^S_{ij} \neq \bot \neq S_{D^S_{ij}}$, and since $S \sqsubseteq T$ and $D^S \sqsubseteq D^T$, this implies $D^T \neq \bot \neq T_{D^T_{ij}}$, and we are done.

Case $(i, j) \in (D^S)^{-1}_{pq}$. Then $(p, q) = D^S_{ij} = D^T_{ij}$. By Lemma D.9.1 and $S_{pq} = \bot$ we obtain $T_{ij} = \bot$. Since $D^T_{ij} \neq \bot \neq T_{pq} = T_{D^T_{ij}}$, we are done.

Case $i \in C_p^{-1}$, $j = q$ and $\bot \neq T_{D^T_{iq}}$. We have $D^T_{iq} \neq \bot \neq T_{D^T_{iq}}$, so it suffices to show $T_{iq} = \bot$. Since $C_i = p$, have $A_i \leftarrow A_p[A_y, A_z]$. By $S_{pq} = \bot$, we therefore have $F^{(u)}(S)_{iq} = \bot$. By $S \sqsubseteq F^{(u)}(S)$, this implies $S_{iq} = \bot$. It suffices to show $i \neq p$, as this implies $T_{iq} = S_{iq} = \bot$.

Assume $i = p$. Then by $S_{pq} = \bot$ we have $T_{pq} = F^{(u)}(S)_{pq} = \bot \neq T_{pq}$, so $T_{pq} \neq T_{pq}$ a contradiction. Thus $i \neq p$, and we are done. □

The previous proof uses the following, which shows that the entry for a complex expression cannot be resolved if its dynamic dependency is undetermined.

Lemma D.9.1 (Dependency strictness)**.** *Let $T \in \mathsf{Table}$, $(i,j) \in \mathsf{Index}$ and $u \in \Sigma^*$. If $T \sqsubseteq T^<(u)$; g_i complex and $D^T_{ij} = \bot$, then $F^{(u)}(T)_{ij} = \bot$.*

Proof. If $(i,j) \notin J_u$ then $F^{(u)}(T)_{ij} = T_{ij}$. Since $T_{ij} \sqsubseteq T^<(u)_{ij}$, the result follows by showing $T^<(u)_{ij} = \bot$. By $(i,j) \notin J_u$ we have $F^{(u)}(T)_{ij} = T_{ij}$ for all T, and by induction we obtain $(F^{(u)})^n(\bot)_{ij} = \bot$ for all $n \geq 0$. We must therefore have $T^<(u)_{ij} = \bot$, since $T^<(u)$ is the least upper bound of all $(F^{(u)})^n(\bot)$.

In the other case, assume $(i,j) \in J_u$, so $F^{(u)}(T) = F^u(T)$. We must have $A_i {\leftarrow} A_x[A_y, A_z]$. By $D^T_{ij} = \bot$ and definition, we have $T_{xj} = \bot$ and hence $F^u(T)_{ij} = \bot$. □

Upon updating a single entry in a table, each entry in the updated reverse dependency map is obtained by appending a predetermined set of indices to the corresponding entry in the old reverse dependency map:

Lemma D.5.5 (Dependency update)**.** *Let $S \sqsubseteq T^<(u)$, $(p,q) \in \Delta_u(S)$ and $T = F^{(u)}_{pq}(S)$. Then for all $(k,\ell) \in \mathsf{Index}$, we have $(D^T)^{-1}_{k\ell} = (D^S)^{-1}_{k\ell} \cup \{(i',q) \mid i' \in C^{-1}_p \wedge (k,\ell) = D^T_{i'q}\}$.*

Proof. Let $(i,j) \in \mathsf{Index}$ such that $D^T_{ij} = (k,\ell)$. Since $S \sqsubseteq T$ we have $D^S_{ij} \sqsubseteq D^T_{ij}$. We have $D^S_{ij} = D^T_{ij} = (k,\ell)$ if and only if $(i,j) \in (D^S)^{-1}_{k\ell}$. The other case, $D^S_{ij} \sqsubset D^T_{ij}$, holds if and only if $(i,j) \in C^{-1}_p \times \{q\}$ by Lemma D.5.3. □

Correctness of Fix

Invariant 1 (Work loop)**.** Assuming variables $u \in \Sigma^*(\# + \varepsilon)$; $T \in \mathsf{Table}$; $R : \mathsf{Index} \to \mathbf{2}^{\mathsf{Index}}$; and $W \subseteq \mathsf{Index}$:

1. $T \sqsubseteq F^{(u)}(T) \sqsubseteq T^<(u)$
2. $R = (D^T)^{-1}$
3. $W = \Delta_u(T)$

Lemma D.9.2 (Initialization)**.** *When entering line 2 in Fix, Invariant 1 holds.*

Proof. Let $u' = a_m a_1 ... a_{n-1}$ and $u = u' a_n$. By the precondition, $T = T^<(u')$ and $R = (D^T)^{-1}$.

Property 1. We first show $T \sqsubseteq F^{(u)}(T)$. Let $(i,j) \in \mathsf{Index}$. We either have $(i,j) \in J_{u'}$ or $(i,j) \notin J_{u'}$.

In the first case we also have $(i,j) \in J_u$, so $F^{(u)}(T)_{ij} = F^{(u')}(T)_{ij} = T_{ij}$, where the last equality follows from the fact that T is a fixed point of $F^{(u')}$.

In the second case $(i,j) \notin J_{u'}$ we have $\forall T'. F^{(u')}(T')_{ij} = T'_{ij}$. Hence $\forall n \geq 0. (F^{(u')})^n(\bot)_{ij} = \bot$, and since $T = T^<(u')$ is the least upper bound of all $(F^{(u')})^n(\bot)$, we have $T_{ij} = \bot \sqsubseteq F^{(u)}(T)_{ij}$.

From the above we conclude $T \sqsubseteq F^{(u)}(T)$, and it remains to show $F^u(T) \sqsubseteq T^<(u)$.

Since $F^{(u)}(T^<(u)) = T^<(u)$, this follows by monotonicity of F and $T = T^<(u') \sqsubseteq T^<(u)$, which in turn follows from Proposition D.4.4.

Property 2. Follows by assumption.

Property 3. Since $T = T^<(u')$, we have $\Delta_{u'}(T) = \emptyset$. Thus $\Delta_u(T) = \Delta_u(T) \setminus \Delta_{u'}(T)$, and by Lemma D.5.1 $(i,j) \in T^<(u)$ if and only if $j = |u| - 1$, $T_{i(|u|-1)} = \bot$ and either $D^T_{i(|u|-1)} \neq \bot \neq T_{D^T_{i(|u|-1)}}$ or g_i simple. But $D^T_{i(|u|-1)} = \bot$ for all $i < |P|$, so $\Delta_u(T) = \{(i, |u|-1) \mid g_i \text{ simple}\}$. □

Lemma D.9.3 (Preservation of consistency). *If $F : \mathsf{Table} \to \mathsf{Table}$ is monotone and $T \sqsubseteq F(T)$, then for all $(p,q) \in \mathsf{Index}$, we have $F_{pq}(T) \sqsubseteq F(F_{pq}(T))$.*

Proof. Since $T \sqsubseteq F(T)$ then in particular $T_{pq} \sqsubseteq F(T)_{pq}$, so $T \sqsubseteq F_{pq}(T)$. By monotonicity, we have $F(T) \sqsubseteq F(F_{pq}(T))$. But then

$$\forall (i,j) \in \mathsf{Index}. T_{ij} \sqsubseteq F(T)_{ij} \sqsubseteq F(F_{pq}(T))_{ij}$$

We now prove $\forall (i,j) \in \mathsf{Index}. F_{pq}(T)_{ij} \sqsubseteq F(F_{pq}(T))_{ij}$. If $(i,j) = (p,q)$, then $F_{pq}(T)_{ij} = F(T)_{ij} \sqsubseteq F(F_{pq}(T))_{ij}$; and if $(p,q) \notin (i,j)$, then $F_{pq}(T)_{ij} = T_{ij} \sqsubseteq F(F_{pq}(T))_{ij}$. □

Lemma D.9.4 (Maintenance). *Invariant 1 is maintained for each iteration of lines 2-10 in* Fix.

Proof. Assume that Invariant 1 holds, and let S refer to the configuration of T at the beginning of the iteration. When the iteration has finished, some $(p,q) \in \Delta_u(S)$ has been picked such that

(a) $T = F^{(u)}_{pq}(S)_{ij}$

(b) $S \sqsubseteq F^{(u)}(S) \sqsubseteq T^<(u)$

(c) $\forall k, \ell. R_{k\ell} = (D^S)^{-1}_{k\ell} \cup \{(i',q) \mid i' \in C_p^{-1}\} \cap (D^T)^{-1}_{k\ell}$

(d) $$\begin{aligned} W = {} & \Delta_u(S) \setminus \{(p,q)\} \\ & \cup (D^S)^{-1}_{pq} \\ & \cup \{(i',q) \mid i' \in C^{-1}_p \wedge D^T_{i'q} \neq \bot \neq T_{D^T_{i'q}}\} \end{aligned}$$

By Lemma D.9.3 on (a), (b), Property 1 is reestablished.
By Lemma D.5.5 on (a), (b), (c), Property 2 is reestablished.
By Lemma D.5.4 on (a), (b) and (d), Property 3 is reestablished. □

Lemma D.9.5 (Termination). *Invariant 1 entails the postcondition of* Fɪx *when the loop in lines 2-10 terminates.*

Proof. When the loop terminates we have $W = \emptyset$. By the invariant we have both $W = \Delta_u(T) = \emptyset$ and $T \sqsubseteq T^<(u)$, so $T = T^<(u)$. □

Bibliography

[1] A. V. Aho and J. D. Ullman. *The Theory of Parsing, Translation, and Compiling*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1972.

[2] R. Becket and Z. Somogyi. DCGs + Memoing = Packrat Parsing but Is It Worth It? In P. Hudak and D. S. Warren, editors, *Practical Aspects of Declarative Languages*, number 4902 in Lecture Notes in Computer Science, pages 182–196. Springer Berlin Heidelberg, Jan. 2008. DOI: 10.1007/978-3-540-77442-6_13.

[3] A. Birman and J. D. Ullman. Parsing Algorithms with Backtrack. In *Proceedings of the 11th Annual Symposium on Switching and Automata Theory (Swat 1970)*, SWAT '70, pages 153–174, Washington, DC, USA, 1970. IEEE Computer Society.

[4] P. Cousot and R. Cousot. Automatic synthesis of optimal invariant assertions: Mathematical foundations. *SIGPLAN Notices*, 12(8):1–12, Aug 1977.

[5] B. Ford. Packrat parsing: Simple, Powerful, Lazy, Linear Time. In *ACM SIGPLAN Notices*, volume 37, pages 36–47. ACM, Sept. 2002.

[6] B. Ford. Parsing Expression Grammars: A Recognition-Based Syntactic Foundation. *ACM SIGPLAN Notices*, 39(1):111–122, Jan. 2004.

[7] B. B. Grathwohl, F. Henglein, U. T. Rasmussen, K. A. Søholm, and S. P. Tørholm. Kleenex: Compiling Nondeterministic Transducers to Deterministic Streaming Transducers. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL 2016, pages 284–297, New York, NY, USA, 2016. ACM.

[8] N. B. r. 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.

[9] R. Ierusalimschy. A Text Pattern-matching Tool Based on Parsing Expression Grammars. *Softw. Pract. Exper.*, 39(3):221–258, Mar. 2009.

[10] K. Kuramitsu. Packrat Parsing with Elastic Sliding Window. *Journal of Information Processing*, 23(4):505–512, 2015.

[11] L. Lee. Fast Context-free Grammar Parsing Requires Fast Boolean Matrix Multiplication. *J. ACM*, 49(1):1–15, Jan. 2002.

[12] S. Medeiros and R. Ierusalimschy. A Parsing Machine for PEGs. In *Proceedings of the 2008 Symposium on Dynamic Languages*, DLS '08, pages 2:1–2:12, New York, NY, USA, 2008. ACM.

[13] S. Medeiros, F. Mascarenhas, and R. Ierusalimschy. From regexes to parsing expression grammars. *Science of Computer Programming*, 93, Part A:3–18, Nov. 2014.

[14] K. Mizushima, A. Maeda, and Y. Yamaguchi. Packrat Parsers Can Handle Practical Grammars in Mostly Constant Space. In *Proceedings of the 9th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, PASTE '10, pages 29–36, New York, NY, USA, 2010. ACM.

[15] T. Parr, S. Harwell, and K. Fisher. Adaptive LL(*) Parsing: The Power of Dynamic Analysis. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '14, pages 579–598, New York, NY, USA, 2014. ACM.

[16] R. R. Redziejowski. Some Aspects of Parsing Expression Grammar. *Fundam. Inf.*, 85(1-4):441–451, Jan. 2008.

[17] R. R. Redziejowski. Mouse: From parsing expressions to a practical parser. In *Concurrency Specification and Programming Workshop*. Citeseer, 2009.

[18] R. R. Redziejowski. Cut Points in PEG. *Fundamenta Informaticae*, 143(1-2):141–149, Feb. 2016.

(The following pages have intentionally been left blank)