Fast Regular Matching without Tiers

Robertas Norkus

4th Year Project Report
Computer Science
School of Informatics
University of Edinburgh

2023

Abstract

Regular expressions are widely used for pattern matching in programming languages, but many of production-grade engines are susceptible to evil regular expression patterns, which are patterns that are specifically crafted to dwarf the performance of systems or even crash them. This includes Links language, which is the source of motivation for this dissertation. In addition to being susceptible to evil regular expression patterns, it also has mismatched regular expression engines between the server and client sides. We make an effort of solving both of these problems by providing our own implementation of regular expressions.

We restrict the scope of regular expressions to those that were originally defined by Kleene to make objectives more tractable. We implement regular expressions using two different approaches in OCaml and then make an attempt of integrating our code into Links. OCaml is our choice of language because Links' implementation language is OCaml as well and there exists a compiler from OCaml to JavaScript for the client side. The first approach of implementing regular expressions is based on nondeterministic finite automata. For the other approach, we consider a lowering of the nondeterministic finite automata into a virtual machine with its own bytecode format. Then, we make a prototype of integrating virtual machine approach into Links language for regular matching. Finally, we compare the performance of our implementations of regular expressions to a widely used V8 JavaScript engine.

Research Ethics Approval

This project was planned in accordance with the Informatics Research Ethics policy. It did not involve any aspects that required approval from the Informatics Research Ethics committee.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(*Robertas Norkus*)

Acknowledgements

I would like to express my deepest gratitude to my supervisor Daniel Hillerström for all the incredible support and guidance you have provided me during my dissertation journey. Your dedication and commitment to my success have truly made a difference, and I cannot thank you enough. Your expertise and willingness to go above and beyond to ensure that I had all the resources I needed to succeed have been invaluable. Despite the challenges I faced in my personal life, you remained patient, understanding, and supportive throughout. Your belief in me helped me push through difficult times and stay focused on my goals. Your insightful feedback and thoughtful suggestions have been instrumental. I am truly grateful for the time and effort you invested in me and I feel honoured to have had you as my supervisor.

Table of Contents

Chapter 1

Introduction

Regular expressions are a powerful tool used by developers to match and manipulate text. They are used in a wide range of applications such as searching for and replacing strings, data validation, and parsing input from users. The ability to define patterns that match text using regular expressions makes them a crucial tool in modern programming languages and operating systems. For example, regular expressions are the cornerstones of widely used search programs such as `grep` [10] and `sed` [25].

However, it is challenging to implement regular expressions in a way that yields good performance for all possible inputs. This is especially true for *evil regular expression* (evil regex) [31] patterns, which are patterns that are specifically crafted to dwarf the performance of systems or even crash them. These evil expression patterns often exploit a similar weakness present in the implementation of regular expression engines, namely, that their implementations rely on some form of backtracking. For many cases the backtracking is not a problem as many regular expression patterns do not cause repeated backtracking to occur. Backtracking makes the asymptotic worst case run time complexity exponential in the size of the input. Evil regular expression patterns trigger this exponential time behaviour.

Many production-grade engines such as V8 [12] are susceptible to evil regular expression patterns [31]. The problem is also present in research-grade engines such as the one that powers the regular expressions of the Links programming language [4]. In the case of Links, there is an extra dimension to the problem, as Links is a tierless programming language that allows code to flow seamlessly between the client and the server. Thus, a malicious client can craft an evil regular expression pattern and send it to the server to cause it to stall. In fact, it can be even worse as Links supports multiple clients connected via the same server [9]. Consequently, it is possible for a malicious client to send code, that embeds an evil regular expression pattern, via a common server to another client, whose execution environment will freeze, or crash, upon executing the received code.

There are multiple ways to implement regular expression matching. In this dissertation we will consider two distinct approaches. First, we consider an implementation technique based on nondeterministic finite automata. This technique is well studied [28]. It is a fairly high-level technique in the sense that it works over transition graphs. Second, we consider a lowering of the nondeterministic finite automata into a virtual machine with its own bytecode format [7]. This allows to more efficiently do regular matching by being more low-level, and it is more versatile: it is easier to implement new features by adding new bytecode instructions in the future.

1.1 Goals, scope and contributions

The goal is to implement regular expressions in multiple ways in functional programming language OCaml. We choose OCaml because it is the implementation language of Links. Following that, another goal is to evaluate the performance of these implementations to already existing implementations that are used in the real world. More specifically, the goal is to test the performance of these implementations against inputs that are specifically crafted to exploit the weakness of those implementations.

The resulting code from this dissertation can then be integrated into Links programming language to better support regular expressions. Code is self-contained, does not have any dependencies and is easy to use.

Production-grade regular expressions often tend to be large and complex by supporting many features that go beyond formal definition of regular expressions. In this dissertation, we restrict our attention to implementing regular expressions as originally defined by Kleene [19]. These regular expressions are well-understood and make the objectives of this dissertation more tractable. This means that such constructs as anchors or backreferences, that are widely implemented in production-grade applications, are not in the scope of this dissertation.

The following objectives were identified and achieved:

- Identification of evil regular expressions that break some implementations of regular expressions, more specifically V8 JavaScript engine.
- Implementing regular expressions using NFA and Thompson's construction in OCaml.
- Implementing regular expressions using Virtual Machine approach in OCaml.
- Measuring the performance of both of the implementations and comparing it to the V8 JavaScript engine. This includes testing evil regular expressions and their effect on the implementations.
- Making a prototype of our implementation integration into Links.

1.2 Outline

This dissertation is structured as follows.

- Chapter 2: describes the background of the work taken in this dissertation by introducing Links programming language and regular expressions.

- Chapter 3: discusses and provides the implementation details of the NFA and VM approaches for implementing regular expressions.
- Chapter 4: evaluates the performance of the implementations through a series of experiments.
- Chapter 5: concludes the dissertation and discusses future work.

Chapter 2

Background

In this chapter we will discuss Links programming language and how it tries to solve the impedance mismatch problem in web development. In addition, we will define what regular expressions are, how they relate to finite automata, a current implementation of regular expression matching in Links, and how some implementations can be broken by carefully crafted regular expressions and inputs.

2.1 Links

This section highlights the main features of Links, the programming language that is at the centre of this dissertation, and its problems with regular expressions.

2.1.1 Impedance mismatch problem

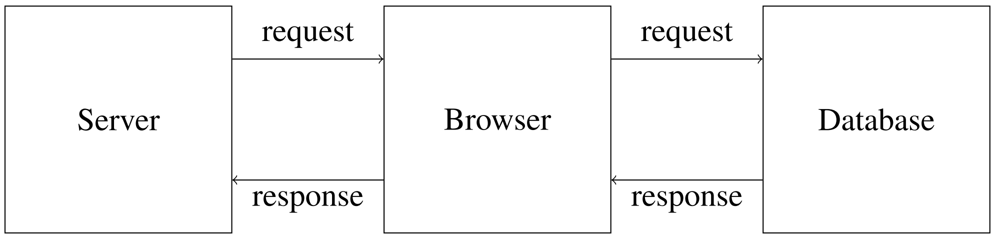


Figure 2.1: Three tiers of the web

In traditional web programs there are three tiers: the client tier, the server tier, and the database tier (as seen in Figure 2.1). The client tier's purpose is to serve the user interface and handle user's input. Most often it uses web technologies such as HTML, CSS, and JavaScript. The server tier is for processing the user's input, handling business logic, and providing the response. This tier is usually implemented with object-oriented programming languages such as Java, Python, and PHP. The database tier is responsible for storing and fetching the data. It most often uses relational databases such as MySQL and PostgreSQL.

The use of different programming languages for the source codes of each web tier leads to impedance mismatch problem. This is because each programming language has their own interfaces and represents data in their own ways using different data structures. As a result, it becomes challenging to transport data between each of the tiers effectively. This leads to the use of ad hoc and error-prone serialization schemes to transfer data between the different tiers. In addition, it often leads to poor performance.

Stonebraker et al. [29] describes one of the most notable examples of the problem of impedance mismatch between the server tier that uses an object-oriented programming language to access data from a relational database. These two paradigms have fundamentally different data models and structures, which make it difficult to map objects to tables and vice versa.

Similarly, Jacobsen [18] has described the impedance mismatch problem between the server and client tiers. Due to the use of different data structures to represent data, there is a need for some sort of serialization and deserialization of data while transporting it between the tiers. This incurs a run time performance penalty that could be avoided. To avoid this penalty and bridge the gap between the server and client tiers, a middleware, as described by Jacobsen [18] in the same paper, could be used.

2.1.2 Links

Links is a programming language whose goal is to solve the impedance mismatch problem in web development [4]. It eliminates the potential problem of having mismatched data types between tiers by providing a single language for all three web tiers. Code for all the tiers gets generated from a single source: it gets compiled into JavaScript for the client side and SQL for the database, while the server side is run with OCaml.

Links is a strict, statically typed, functional language. It incorporates many common features found in functional programming languages such as first-class functions, immutable data, algebraic data types, it also includes features tailored for web programming such as formlets as an abstraction of webforms [5], code mobility between tiers, an integrated database query language, a message-passing concurrency model to enable modular compartmentalization of interactive components. More recently, Links has been extended with effect handlers to support user-definable computational effects in direct-style [14, 15], session types to type communication protocols [20, 9], incremental relational lenses for updatable views in databases [17].

Links also includes support for regular expressions, which is the main topic of this thesis. Regular expressions are second class citizens in Links, meaning that they cannot be passed as arguments to functions or returned from functions. Instead, a regular expression is defined inline when testing some string, for instance the expression `str =~ /ab*a/`.

Currently, a naïve implementation is used for the regular expressions for string matching in Links, which means it is susceptible to particular kinds of regular expressions which can lead to evaluations running in exponential time or potentially cause stack overflows. In addition, the current implementation of regular expressions in Links does not guarantee the same evaluation on all the tiers, which contributes to the impedance mismatch problem as discussed in Section 2.1.1.

More on regular expressions and potential problems with their naïve implementation in the next sections.

2.2 Regular expressions

The notion of regular expressions was first introduced by mathematician Stephen Kleene in 1956 [19]. Computer scientist Ken Thompson adapted the mathematical theory as presented by Kleene and popularized the use of regular expressions for string searching while developing Unix operating system in 1960s and 1970s. This later led to the development of now widely popular string searching programs such as `grep` [10] and `sed` [25], for which regular expressions are the cornerstones of their functionality. These types of regular expressions, that go beyond the original theory as laid out by Kleene, are now publicly called as *regex* in short. In this dissertation, we will focus more on the regular expressions as defined by Stephen Kleene, which have rich mathematical theory behind them.

A regular expression is a compact notation for describing sets of characters [6]. Rather than listing strings one by one that are contained in some set, it is more convenient to describe this set using regular expressions. As an example, three strings such as “Edinburgh”, “Hamburg”, and “Tilburg” could be represented as a regular expression $(Edin \parallel Ham \parallel Til)burgh^?$. When a string is in the set defined by a regular expression, it is commonly said that the regular expression *matches* the string. So $(Edin \parallel Ham \parallel Til)burgh^?$ matches “Edinburgh” in this example. Here $\parallel$ symbol means matching either “Edin”, “Ham”, or “Til”. Then, the matching expects string “burg” and finally $h^?$ is for optionally matching character h. In addition, multiple different regular expressions can describe the same set of strings. For $(Edin \parallel Ham \parallel Til)burgh^?$ case, $(Edi \parallel Ham \parallel Til)(burg \parallel nburgh)$ would also be a valid representation of the same set of strings.

Regular expressions are closely related to finite automata, as they are often used to recognize a specific language, which is also a set of strings [28]. More details on this later in the chapter.

2.2.1 The syntax

The following grammar defines the syntax of fairly standard regular expression language.

$$R ::= \varepsilon \mid \mathsf{char} \mid R_1 \circ R_2 \mid R_1 \parallel R_2 \mid (R) \mid R^* \tag{2.1}$$

Informally, the language constants and operators are interpreted as follows.

- The empty expression ε matches an empty string. It means that it has no actual pattern to match, but still can be used as a valid regular expression for building more complex regular expressions.

- The basic building block of a regular expression is a character, char, which is drawn from some finite alphabet Σ. A character matches only itself.
- The concatenation operator ($\circ$) forms an expression which matches the sequence of the subexpressions R_1 and R_2. Often the operator is implicit, that is, we often write R_1R_2 to mean $R_1 \circ R_2$.
- The choice operator ($||$) forms an expression which matches both R_1 and R_2.
- The notation (R) means grouping, it has the highest precedence of all the operators.
- The asterisk operator $(-^*)$ matches zero of more occurrences of the preceding expression R. It is also known as the *Kleene star* [28].

The precedence order of the operators from the strongest binding to lowest: grouping, then Kleene star, alternation, and finally concatenation.

This syntax given above is enough to describe all regular languages [28].

Production-grade regular expression languages often offer more operators such as $R^?$ and R^+, which means optionally match R and match one or more repetitions of R, respectively. These two operators happen to be expressible in terms of basic language given above, i.e. $R^? = R \mid\mid \varepsilon$ and $R^+ = RR^*$. Nevertheless, they may be included in a language for reasons of efficiency. Another thing one might find in a production-grade regular expression languages is finite repetition notation, e.g. $R\{n\}$ means match n repetitions of R. But with such additions we begin to leave realm of formal regular expression languages and then start to enter the realm of what is usually referred to as *regexes*.

2.2.2 Finite Automaton

As mentioned before, regular expressions can also be expressed as finite automaton [28]. Regular expressions and finite automata have the same expressive power. Regular expressions can be converted to finite automaton and vice versa. There are two types of finite automata that are involved in conversions: nondeterministic finite automata (NFA) and deterministic finite automata (DFA). The main difference is that NFA can have more than one possible next state for a particular input character, while for DFA there is only one possible next state for a given symbol from each state.

There are several algorithms for doing conversions between finite automata and regular expressions. To convert from a regular expression to a nondeterministic finite automaton, the two most common algorithms are used: Thompson's construction algorithm [30] and Glushkov's construction algorithm [11]. To convert a finite automaton to a regular expression, Kleene's algorithm [19] can be used. An adaptation of this algorithm for conversion between deterministic finite automata and regular expressions has been given by Hopcroft et al. [16], while conversion between nondeterministic finite automata and regular expressions has been described by Gross and Yellen [13].

As an example, regular expression $a(bb)^+a$ is equivalent to the finite automaton shown in Figure 2.2, which in this case is deterministic. The state machine starts in the state

s_0 and then transitions to the state s_1 when it reads an a character. Similarly, the state machine transitions to state s_2 when b is read and then s_3 when another b is read. Now from state s_3 there are two options: the state machine will transition to the accepting state s_4 if it reads the character a, while it will loop back to the state s_2 if it reads the character b. As it can be seen, there are no transitions with the same label from any of the states, so the given finite automaton is deterministic.

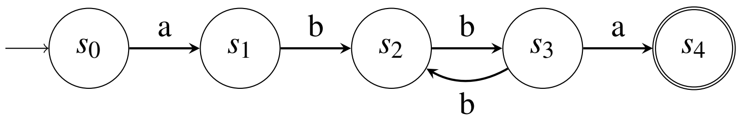


Figure 2.2: Deterministic finite automaton for $a(bb)^+a$

As another example, the same $a(bb)^+a$ regular expression can also be expressed as a nondeterministic finite automaton (Figure 2.3). Once once again, the state machine starts in state s_0, transitions to state s_1 after it reads character a, and then transitions to state s_2 after it reads b. Now when it reads character b again, it transitions to two states simultaneously: s_3 and back to s_1. Being in more than one state at a time makes this finite automaton nondeterministic. From s_1 it can go back to state s_2 when it reads character b, or it can end up in the accepting state s_4 when it reads character a.

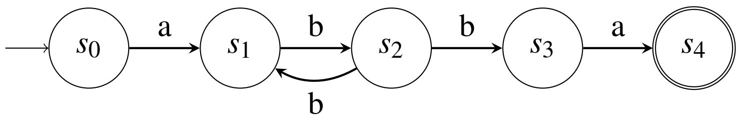


Figure 2.3: Nondeterministic finite automaton for $a(bb)^+a$

2.2.3 Naïve algorithm for search

A naïve algorithm implementing regular expression search would convert a regular expression into a nondeterministic finite automaton and then simulate it. For guessing the next state when there are multiple options, one of the choices can be explored first and then later backtracked if it does not work.

This naïve algorithm works in most cases, but for some specific inputs (described in the next subsection) it can take long time evaluate due to backtracking, as all the possible paths must be tried and the number of paths can be exponential.

2.2.4 Evil regular expressions

A regular expression, that could get stuck or break on a specially crafted input, could be called as a evil regular expression, or evil regex in short [31]. These kinds of regular expressions can be exploited in Regular expression Denial of Service (ReDoS) attacks, as supplying a crafted input can hang or crash the system. So, there are two types of regular expressions: those that run indefinitely given some simple and short input, or those that crash given a large enough input.

2.2.4.1 Running indefinitely

As previously discussed, inputs on particular regular expressions can run slowly due to exponential number of paths that have to be tried. For example, regular expression $\hat{}(a^+)^+\ \$$ [31], which can be expressed with NFA (Figure 2.4), has an exponential number of paths.

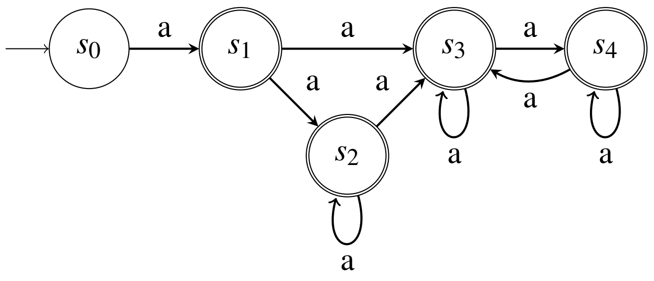


Figure 2.4: NFA for $\hat{}(a^+)^+\ \$$

There are exactly 2^n paths for an input containing n characters of a. This leads to the runtime of $O(2^n)$. The performance of this regular expression on V8 engine can be seen in 2.5 Figure.

2.2.4.2 Stack overflow

Some programming languages have implemented memoization to avoid backtracking for particular regular expressions, but they are still vulnerable to stack overflow issue. Stack space is used for backtracking, and it is linear to the size of the input. This causes certain implementations to run out of stack space. An example can be seen in V8 engine:

```
d8> const re = /^(ab?)*$/
d8> re.exec("a".repeat(10000000))
(d8):1: RangeError: Maximum call stack size exceeded
```

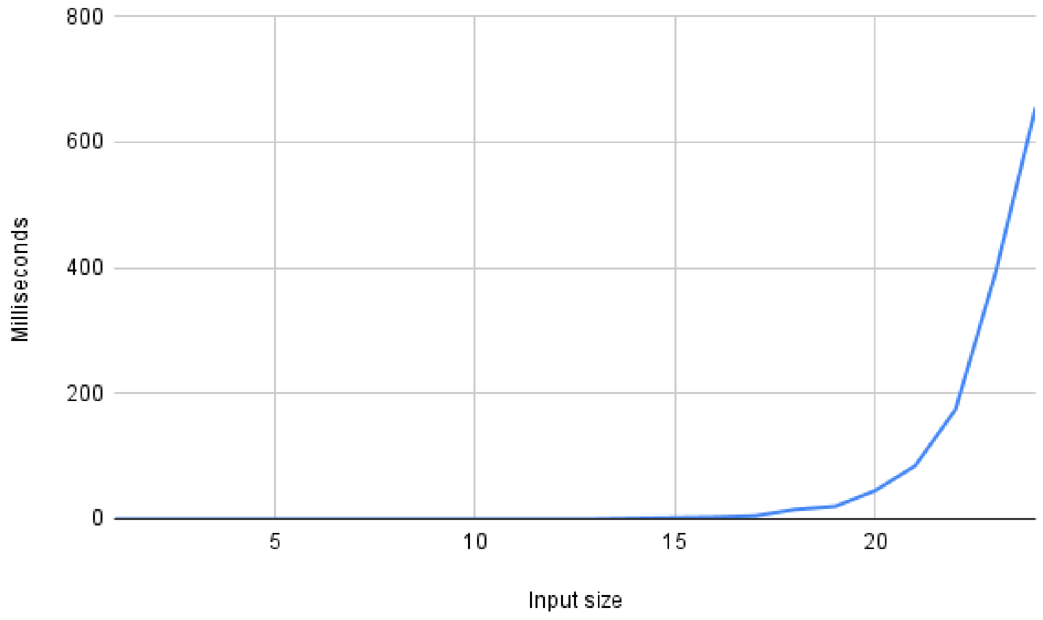


Figure 2.5: Performance of ^$(a^+)^+$ \$ in milliseconds based on input size in V8

Chapter 3

Implementation

We use OCaml as the language to implement regular expressions, since Links uses OCaml for the server side, JavaScript for the client side, and there exists a compiler from OCaml to JavaScript. In this chapter we apply two main approaches for implementing regular expressions: one based on nondeterministic finite automata and another based on virtual machines. The goal of using virtual machine approach is to generate bytecode that is almost like a representation of nondeterministic finite automata at a lower level. These both approaches are then compared to each other and to widely used JavaScript engine V8 in Chapter 4.

3.1 Regular expression parser

To start with, we use an algebraic data type to represent the syntax of regular expressions:

```
type t =
  | Eps              (* Empty string *)
  | Char of char     (* Character, e.g. a, b, c, ... *)
  | Alt of t * t     (* Choice, e.g. a|b, ab|ba, ... *)
  | Seq of t * t     (* Concatenation, e.g. aa, abba, ... *)
  | Star of t        (* Kleene closure, e.g. a*, ab*a, ... *)
  | Plus of t        (* Transitive closure, e.g. a+, ab+a, ... *)
  | Opt of t         (* Optional, e.g. a?, ab?a, ... *)
```

This representation is almost a direct transliteration of the grammar given in Equation 2.1 extended with the transitive closure operation ($-^{+}$) and the optional operator ($-^{?}$).

Following that, we have a recursive parser (adapted from [24]) to parse a string representation of a regular expression into a tree-like structure that is more suitable for further compilations by both of the mentioned approaches of implementing regular expression matching. It works over a stream of tokens (in this case a list of characters), which gets generated by a simple lexer, which in this case just explodes a string into a list of characters.

```
let rec seq = function
    [] -> assert false
  | [x] -> x
  | x :: xs -> Seq (x, (seq xs))

let rec parse' str ps = match (str, ps) with
  | ([], _) | (')' :: _, _) -> (seq (List.rev ps), str)
  | ('(' :: rest, _) ->
      begin match parse' rest [] with
      | (r, ')' :: rest) -> parse' rest (r :: ps)
      | _ -> raise Unbalanced
      end
  | ('|' :: rest, p :: ps) ->
      let (other, rest) = parse' rest [] in
      parse' rest (Alt (p, other) :: ps)
  | ('*' :: chs, p :: ps) -> parse' chs (Star p :: ps)
  | ('?' :: chs, p :: ps) -> parse' chs (Opt p :: ps)
  | ('+' :: chs, p :: ps) -> parse' chs (Plus p :: ps)
  | (c :: chs, _)         -> parse' chs (Char c :: ps)
```

The recursive parser `parse'` takes in a list of characters (given by the lexer) and a list of currently processed regular sub-expressions, and matches them against these cases:

1. If either the remaining list of characters is empty or the end of the group is reached (given by the character `)`), put all currently processed sub-expressions into a regular expression that represents a sequence by calling `seq` function.

2. If the current character is `(` (an indication of the start of a group), then recursively process that group and expect to find the end of the group character `)`. If the end of the group character is not found, then an exception (unbalanced parenthesis) is raised.

3. If the current character is an alternation character `|`, then recursively process the rest of the characters and combine the results into a regular expression that represents alternation.

4. If the current character is a repetition character `*, ?, +`, then wrap a currently processed regular sub-expression into an appropriate regular expression.

5. If the current character is not one of the special characters, then add a regular expression that matches that character.

Since each token (or a character) is only processed once, it is apparent that the asymptotic complexity of this parser is $O(n)$, where n is the length of the token stream. The lexer explodes the string into a list of characters. It has the asymptotic complexity of $O(n)$ as well, where n is the length of the string.

3.2 Nondeterministic finite automaton

Thompson's construction is an algorithm for transforming a regular expression into an equivalent nondeterministic finite automaton. The algorithm is named after Ken Thompson, who wrote down the algorithm in the late 60s [30]. However, the algorithm is sometimes also refereed to as McNaughton-Yamada-Thompson algorithm [1], as McNaughton and Yamada developed a similar algorithm in the early 60s [22].

We use this data type to represent an NFA:

```
module StateSet = Set.Make(Int)
module AlphSet = Set.Make(Char)

type transition = { q : int; sym : char; q' : int }
type etransition = { eq : int ; eq' : int }

type nfa = {
  states : StateSet.t;
  alph   : AlphSet.t;
  trans  : transition list;
  etrans : etransition list;
  start  : StateSet.t;
  final  : StateSet.t;
}
```

Here `states` is a set of all reachable states indexed by integers, `alph` is a set of input alphabet, `trans` is the list of transitions with starting and accepting state indexes and a transition symbol, `etrans` is the list of empty transitions with starting and accepting state indexes, `start` is a set of starting states, `final` is a set of accepting states.

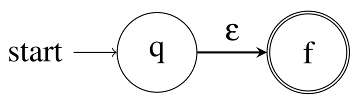
3.2.1 Building the NFA

The algorithm recursively splits expressions into sub-expressions and then constructs NFA for each of those sub-expressions using predetermined rules. The constructed NFA follows these properties:

- NFA has only one starting state, which does not have any incoming transitions.
- NFA has only one accepting state, which does not have any outgoing transitions.
- Each state in the NFA has at most two outgoing transitions.
- The number of states is equal to $2n - m$, where n is the number of symbols apart from parentheses and m is the number of concatenations.

There are five rules for converting any regular expression into NFA, as described by Aho et al. [1]. Let $N(R)$, $N(R_1)$ and $N(R_2)$ correspond to nondeterministic finite automata for regular expressions R, R_1, and R_2, respectively.

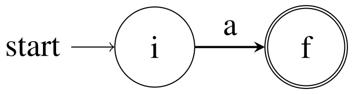
Empty expression ε gets converted to



In code, we add both of the new states to the set of states, set the first state as starting state, second state as final state, and add an empty transition between those two states:

```
let nullNfa = {
  states = StateSet.of_list [0; 1];
  alph = AlphSet.empty;
  trans = [];
  etrans = [{eq = 0; eq' = 1}];
  start = StateSet.of_list [0];
  final = StateSet.of_list [1];
}
```

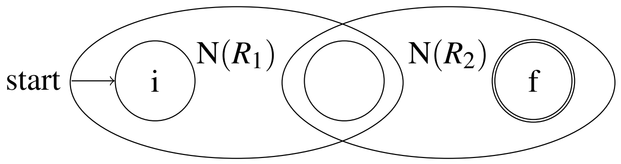
Single character a gets converted to



In code, similar to the conversion of empty transition, we add a new starting and a new final state to the list of states, add a transition between these states that match the given character, and add that character to the alphabet.

```
let charNfa a = {
  states = StateSet.of_list [0; 1];
  alph   = AlphSet.of_list [a];
  trans  = [{q = 0; sym = a; q' = 1}];
  etrans = [];
  start  = StateSet.of_list [0];
  final  = StateSet.of_list [1];
}
```

Sequence $R_1 \circ R_2$ gets converted to

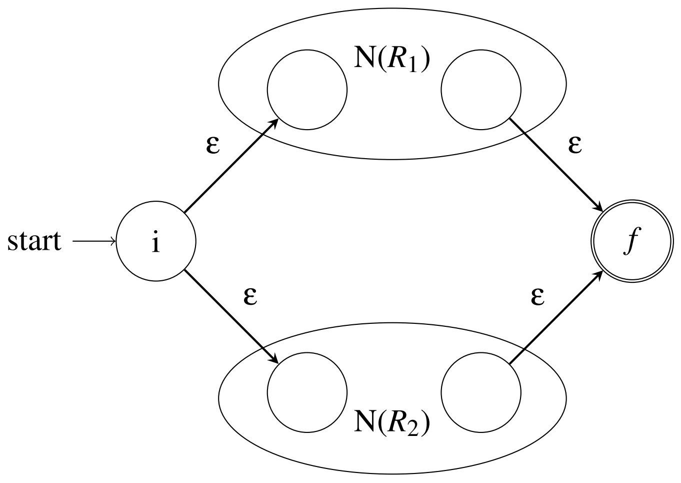


The starting state of $N(R_1)$ becomes the starting state q of the newly generated NFA. The accepting state of $N(R_1)$ becomes the starting state of $N(R_2)$. The accepting state of $N(R_2)$ becomes the accepting state f of the newly generated NFA.

In code, first, the states of both of the NFAs are reindexed, so that there are no collisions of state indexes. Then, all the states, transitions and empty transitions are combined. In addition, an empty transition is added between the start state of the first NFA and the accepting state of the second NFA. The starting state of the first NFA becomes the new starting state of the newly generated NFA. Similarly, the accepting state of the second NFA becomes the new accepting state of the whole NFA.

```
let concatNfa a b =
  let concatNfa' a b = {
    states = StateSet.union a.states b.states;
    alph = AlphSet.union a.alph b.alph;
    trans = List.append a.trans b.trans;
    etrans = List.append (List.map2 (fun x y -> {eq = x; eq' = y})
          (StateSet.elements a.final) (StateSet.elements b.start))
          (List.append a.etrans b.etrans);
    start = a.start;
    final = b.final;
  }
  in concatNfa' (reindexNfa a 0)
              (reindexNfa b (StateSet.cardinal a.states))
```

Alternation $R_1 \parallel R_2$ gets converted to



A new starting state i is added with two outgoing empty transitions to both starting states of $N(R_1)$ and $N(R_2)$. The accepting states of both $N(R_1)$ and $N(R_2)$ become regular states and have outgoing empty transitions to the new accepting state f.

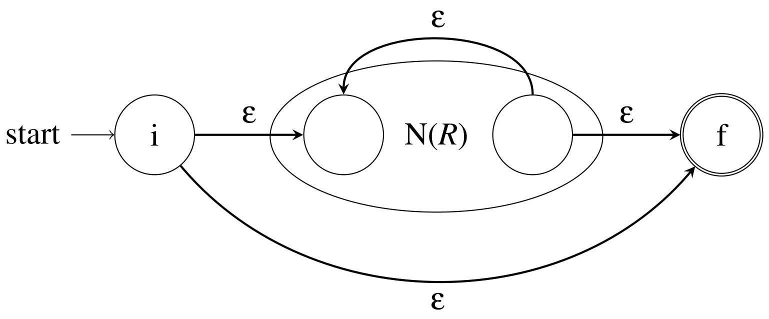
In code, the conversion starts at the function `unionNfa`. First, once again, the states of both NFAs get reindexed. Then, all the states, transitions, and empty transitions are combined by the function `unionNfa'`. This results in an NFA that has more than one start and final state, which violates the properties stated at the start of this section, so a function called `thompson` is used to convert the NFA into a right one. It creates a new starting and a new accepting state, and adds empty transitions between each pair of the new starting state and old starting states. The same is done between each pair of the new accepting state and old accepting states.

```
let thompson a =
  let thompson' a start final = {
    states = StateSet.add start (StateSet.add final a.states);
    alph = a.alph;
    trans = a.trans;
```

```
      etrans =
        List.append a.etrans
          (List.append (List.map (fun x -> {eq = start; eq’ = x})
          (StateSet.elements a.start))
            (List.map (fun x -> {eq = x; eq’ = final})
            (StateSet.elements a.final)));
      start = StateSet.singleton start;
      final = StateSet.singleton final; }
    in
    let start = StateSet.cardinal a.states in
    let final = StateSet.cardinal a.states + 1 in
    thompson’ a start final

  let unionNfa a b =
    let unionNfa’ a b = {
      states = StateSet.union a.states b.states;
      alph = AlphSet.union a.alph b.alph;
      trans = List.append a.trans b.trans;
      etrans = List.append a.etrans b.etrans;
      start = StateSet.union a.start b.start;
      final = StateSet.union a.final b.final;
    }
    in thompson (unionNfa’ (reindexNfa a 0)
                (reindexNfa b (StateSet.cardinal a.states)))
```

Kleene star R^* gets converted to



New starting state i and accepting state f are created with empty transitions from and to $N(R)$. An empty transition between i and f is added for the case when there are no repetitions. An empty transition between the accepting and starting states of $N(R)$ is added for the case when there are repetitions.

In code, a new starting state and a new accepting state are created and added to the set of already existing states. For empty transitions, three different ones are created. First, an empty transition is added between the newly created start and accepting states. Second, an empty transition is added between the newly created start state and already existing start state in the given NFA. Similarly, an empty transition is added between the new accepting state and the old accepting state.

```
let starNfa a =
  let starNfa' a start final = {
    states = StateSet.add start (StateSet.add final a.states);
    alph = a.alph;
    trans = a.trans;
    etrans =
      {eq = start; eq' = final} ::
        (List.append (List.map2 (fun x y -> {eq = x; eq' = y})
        (StateSet.elements a.final) (StateSet.elements a.start))
          (List.append a.etrans
            (List.append (List.map (fun x -> {eq = start; eq' = x})
            (StateSet.elements a.start))
              (List.map (fun x -> {eq = x; eq' = final})
              (StateSet.elements a.final)))));
    start = StateSet.singleton start;
    final = StateSet.singleton final;
}
in starNfa' a (StateSet.cardinal a.states)
              (StateSet.cardinal a.states + 1)
```

3.2.2 Compilation

The Thompson's construction algorithm starts at the function `compile`. It takes in a parsed regular expression (as described in Section 3.1), uses the construction rules as described in previous section to recursively construct the NFA, which can then be used for matching.

Compilation function:

```
let rec compile = function
    Eps        -> nullNfa
  | Char a     -> charNfa a
  | Alt (a, b) -> unionNfa (compile a) (compile b)
  | Seq (a, b) -> concatNfa (compile a) (compile b)
  | Star a     -> starNfa (compile a)
  | Plus a     -> compile (Seq (a, Star a))
  | Opt a      -> compile (Alt (a, Eps))
```

The operators $(-^?)$ and $(-^+)$ get desugared to equivalent regular expressions that only use sequence, alternation and star symbol. That way there is no need for any custom construction rules for these symbols.

3.2.3 Simulating the NFA

The input string is read one symbol at the time and all the active states are advanced to the next state according to the read symbol. In addition, all empty transitions are taken as well. This means that the state machine can be in multiple states at once. This allows to avoid backtracking, when only one state at the time could be active and backtracking would be used to go back if the state does not end up at the accepting state.

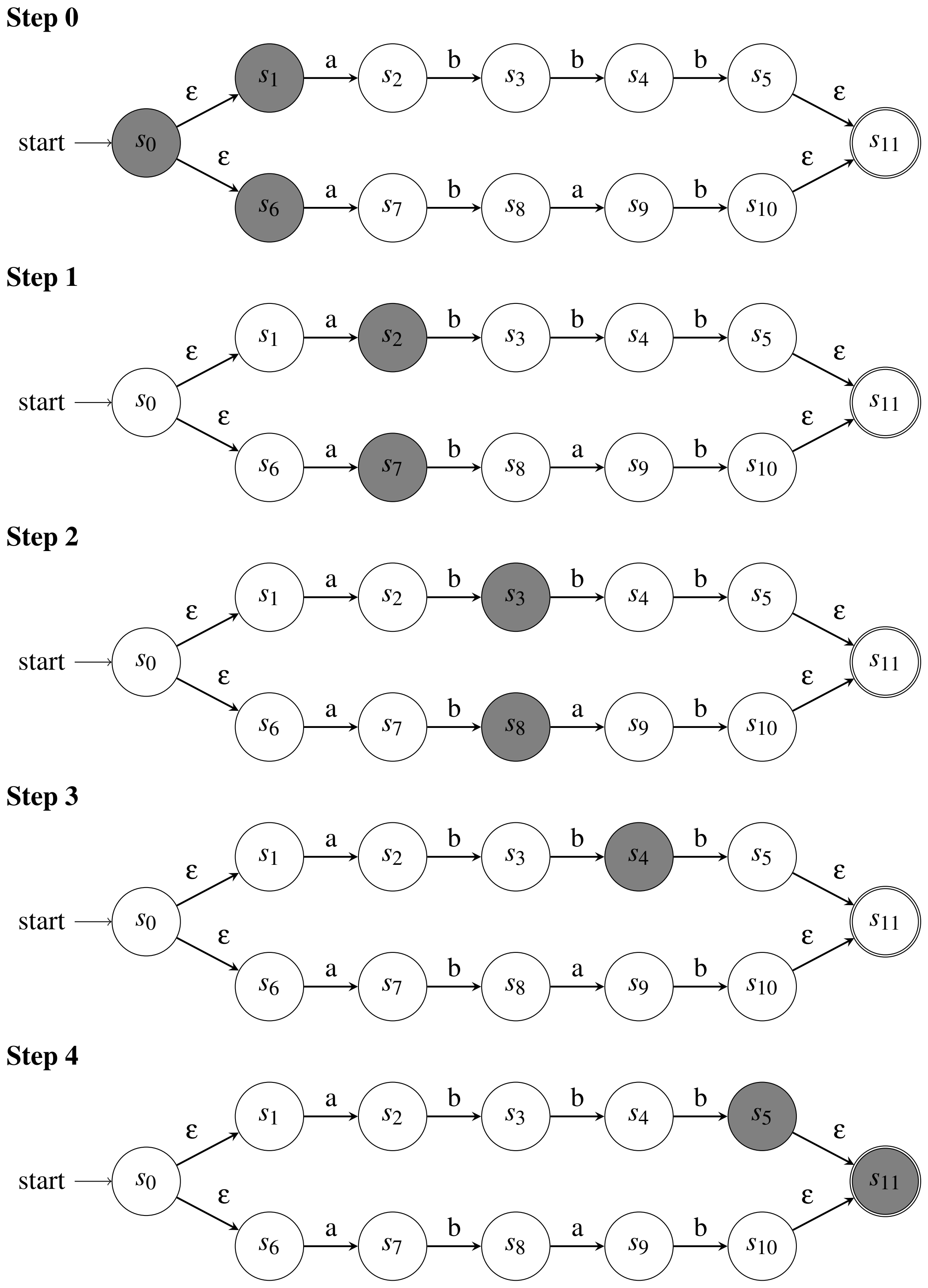


Figure 3.1: Example of NFA simulation

An example (adapted from [6]) for the regular expression *abbb* || *abab* and input string *abbb* can be seen in Figure 3.1. The simulation starts at the starting state and all the states that are reachable from the starting state via empty transitions, as can be seen in step 0. In steps 1 and 2, machine gets advanced while being in two states simultaneously. In step 3 only one active state remains, with step 4 reaching accepting state.

Code

```
let rec accepts nfa str =
  match str with
  | [] -> (* Check that the accepting state is active *)
  | h :: hs -> accepts (move nfa h) hs
```

The matching algorithm begins at the `accepts` function, which recursively advances the states in the NFA one character at the time. For each active state, it takes all appropriate transitions according to the read symbol. In addition, it takes all the appropriate empty transitions (multiple times if needed), until all the states reachable by empty transitions are in the active list.

When the input string is fully processed, it checks if the accepting state is active. If it is active, then the regular expression matches the provided input string, otherwise it does not match.

3.2.4 Asymptotic complexity

In the worst case, the NFA can be at all the states at each step, but this only results in a constant time of work, as it only depends on the size of the regular expression rather than the input string. So this improves the backtracking solution, which required exponential time to process particular input strings. It is efficient because only the active states are tracked rather than the set of paths that have been taken at any point. There are only $2n - m$ states, where n is the number of symbols in the regular expressions and m is the number of concatenations, so overall the asymptotic complexity is $O(n)$.

3.3 Virtual machine

The second approach of implementing regular expressions, that we compare to the NFA approach, is based on implementing a virtual machine (VM) and is adapted from the description of Cox [7]. The basic idea is to implement instructions at a lower level, which would encode the NFA graphs. The VM executes a list of threads, each thread maintains its own program counter.

Definition of instructions:

- **Char a**: if the current character is not *a*, kill the thread. Otherwise, read the next character and advance the program counter.
- **Match**: the thread found the match, stop it.
- **Jump x**: jump to the instruction *x*.

- **Split x, y**: split the current thread. Current thread continues with the instruction x. New thread is created with the program counter pointing to instruction y.

We use this representation of an instruction in code, matching the description above, with `ptr` denoting a pointer into the instruction stream:

```
module Inst = struct
  type ptr = int          (* Instruction pointer *)
  type t =
    | Char of char        (* Match a character *)
    | Match               (* Match found *)
    | Jump of ptr         (* Move pc to ptr *)
    | Split of ptr * ptr  (* Fork two threads *)
end
```

For representing a thread in the virtual machine, we use `Thread` that basically encapsulates the instruction pointer:

```
module Thread = struct
    type t = Inst.ptr

    let make : Inst.ptr -> t
      = fun inst -> inst
  end
```

3.3.1 Translating into bytecode instructions

$$
\begin{aligned}
\llbracket - \rrbracket \;&:\; \mathsf{Nat} \times \mathsf{Regex.t} \to \mathsf{Inst.t\ list} \\
\llbracket \varepsilon \rrbracket_o &= [\,] \\
\llbracket a \rrbracket_o &= [\texttt{Char a}] \\
\llbracket R_1 \circ R_2 \rrbracket_o &= \llbracket R_1 \rrbracket_o \mathbin{+\!\!+} \llbracket R_2 \rrbracket_{o+|R_1|} \\
\llbracket R_1 \parallel R_2 \rrbracket_o &= \quad [\texttt{Split}\ (o+1),\ (o+2+i)] \\
&\quad \mathbin{+\!\!+} \llbracket R_1 \rrbracket_{o+1} \\
&\quad \mathbin{+\!\!+} [\texttt{Jump}\ (o+2+i+j)] \\
&\quad \mathbin{+\!\!+} \llbracket R_2 \rrbracket_{o+2+i} \qquad \textbf{where } i = |R_1| \text{ and } j = |R_2| \\
\llbracket R^* \rrbracket_o &= \quad [\texttt{Split}\ (o+1),\ (o+2+i)] \\
&\quad \mathbin{+\!\!+} \llbracket R \rrbracket_{o+1} \\
&\quad \mathbin{+\!\!+} [\texttt{Jump}\ o] \qquad \textbf{where } i = |R| \\
\llbracket R^? \rrbracket_o &= \quad [\texttt{Split}\ (o+1),\ (o+1+i)] \\
&\quad \mathbin{+\!\!+} \llbracket R \rrbracket_{o+1} \qquad \textbf{where } i = |R| \\
\llbracket R^+ \rrbracket_o &= \quad \llbracket R \rrbracket_o \\
&\quad \mathbin{+\!\!+} [\texttt{Split}\ o,\ (o+1+i)] \qquad \textbf{where } i = |R|
\end{aligned}
$$

Figure 3.2: Translation rules for regular expressions

$$\begin{aligned}
|-| &: \mathsf{Regex.t} \rightarrow \mathsf{Nat} \\
|\varepsilon| &= 0 \\
|a| &= 1 \\
|R_1 \circ R_2| &= |R_1| + |R_2| \\
|R_1 \parallel R_2| &= 2 + |R_1| + |R_2| \\
|R^*| &= 2 + |R| \\
|R^?| &= 1 + |R| \\
|R^+| &= 1 + |R|
\end{aligned}$$

Figure 3.3: The length of the instruction sequence for a particular regular expression

Figure 3.2 and Figure 3.3 formally define the translation of a regular expression into bytecode instructions based on the grammar in Equation 2.1. Function in Figure 3.2 is parameterized by an offset, o, into the instruction stream and a sub-expression, R.

Going through all the definitions one by one:

- Empty expression ε gets converted into an empty list of instructions and thus the length of the instruction sequence for it is equal to 0.
- Single character a only generates `Char` instruction and thus, the length of the instruction sequence is equal to 1.
- Sequence $R_1 \circ R_2$ is simply a concatenation of instruction sequences generated from R_1 and R_2. Thus, the lengths of the instruction sequences get added together as well. Note, index of the first instruction generated from R_2 follows the index of the last instruction from R_1.
- Alternation $R_1 \parallel R_2$ gets generated as follows: first instruction in the sequence is a `Split`, with $o+1$ and $o+2+i$ as targets. Here $o+1$ is the first instruction of R_1 and $o+2+i$ is the first instruction of R_2. Then, the `Split` instruction is followed by the instruction sequences of R_1 and R_2 with a `Jump` instruction in between. This `Jump` has a target to the instruction that follow right after the last instruction of R_2. This achieves the goal of alternation: either only instructions of R_1 or R_2 are executed, but not both. Regarding the length of alternation regular expression, it equals to the lengths of instruction sequences of both R_1 and R_2 plus 2 (for added `Split` and `Jump` instructions).
- Kleene star R^* has three parts: first instruction in the sequence is a `Split` with targets $o+1$ and $o+2+i$. Here $o+1$ is the first instruction of R and $o+2+i$ is right after the last instruction of R^*. Then, the instruction sequence of R follows the `Split` instruction. Finally, the instruction sequence of R^* ends with a `Jump` instruction, pointing back to the first instruction of R^*, which in this case is the `Split`. So, this `Jump` achieves the goal of executing R more than once and the `Split` instruction with the target outside R^* allows for R not to be run at all.

The length of the instruction sequence of R^* equals to the length of instruction sequence of R plus 2 (for added `Split` and `Jump` instructions).

- $R^?$ is generated as follows: first instruction in the sequence is a `Split` with targets $o+1$ and $o+1+i$, which is then followed by the instruction sequence of R. Here $o+1$ is the first instruction of R and $o+1+i$ is the instruction that follows the last instruction of $R^?$. This `Split` achieves the intended goal of the operator: either instruction sequence of R is run or it gets skipped. The length of the instruction sequence of $R^?$ equals to the length of instruction sequence of R plus 1 (for the added `Split` instruction).
- R^+ is similar to $R^?$ and has two parts as well: instruction sequence of R is followed by a `Split` instruction with targets o and $o+1+i$. Here o is the first instruction of R and $o+1+i$ is the instruction that follows the last instruction of R^+. Having this `Split` instruction ensures that the instruction sequence of R is run either once or more than once. The length of the resulting instruction sequence equals to the length of instruction sequence of the sub-expression R plus 1 (for the added `Split` instruction).

3.3.1.1 Code

The translation from the regular expression to the instructions happens in the function `compile`. It takes in a regular expression in the format described in Section 3.1 and outputs an array of instructions.

```
let compile : Regex.t -> Inst.t array
  = fun regexp ->
  let dummy = Inst.Char '0' in
  let insts = Array.make (program_length regexp + 1) dummy in
  let ptr = ref 0 in
  let rec compile : Regex.t -> Inst.ptr ref -> Inst.t array -> unit
  = match regexp with
    (* Recursive compilation *)
  in
  compile regexp ptr insts;
  Array.set insts !ptr Inst.Match;
  insts
```

First, it creates an appropriately sized array with dummy values, which then will be replaced by actual instructions during compilation. It is filled with dummy values because OCaml does not allow uninitialised data. To calculate the needed size of the array based on the given regular expression, `program_length` is used, which is the transliteration of the definition in Figure 3.3:

```
let rec program_length : Regex.t -> int = function
  | Regex.Eps -> 0
  | Regex.Char _ -> 1
  | Regex.Seq (e, e') -> program_length e + program_length e'
  | Regex.Alt (e, e') -> 2 + program_length e + program_length e'
  | Regex.Star e -> 2 + program_length e
  | Regex.Opt e -> 1 + program_length e
  | Regex.Plus e -> 1 + program_length e
```

In addition, `compile` initializes a pointer that points to the next available slot in the array of instructions. Since during compilation the array of instructions is not filled in linear fashion, this pointer allows avoiding backtracking.

The actual compilation happens in the recursive function `compile`, which is parameterized by the regular expression being compiled, an instruction pointer, and the instruction sequence array. It matches the regular expression compiled with all possible options and compiles them individually.

First, empty expressions are skipped, exactly as formally defined in Figure 3.2:

```
  | Regex.Eps -> ()
```

A single character gets directly translated to `Char` instruction, exactly as formally defined in Figure 3.2:

```
  | Regex.Char ch ->
    Array.set insts !ptr (Inst.Char ch);
    incr ptr
```

For sequence, both of the sub-expressions are compiled in turn, exactly as formally defined in Figure 3.2:

```
  | Regex.Seq (e, e') ->
    compile e ptr insts;
    compile e' ptr insts
```

As for alternation, it is a bit more complicated. First, both sub-expressions of the alternation are compiled separately and the positions of their instructions are saved. Then, there is enough information to finally generate the `Split` and `Jump` instructions at their appropriate positions in the instruction array. This slightly differs from the formal translation rule in Figure 3.2. We choose not to compute the length of the instruction sequences of the sub-expressions, because it would lead to wasteful recomputation all the time. Instead, we imperatively keep track of the offset pointer, remembering the required positions for the `Split` and `Jump` instructions, whose targets might not be known at first.

```
  | Regex.Alt (e, e') ->
    let split_pos = !ptr in
    incr ptr;
    (* Compile e. *)
    compile e ptr insts;
    let jump_pos = !ptr in
    incr ptr;
    (* Compile e'. *)
    let e'_pos = !ptr in
    compile e' ptr insts;
    (* Generate the Split instruction. *)
    let split = Inst.Split (split_pos+1, e'_pos) in
    Array.set insts split_pos split;
    (* Generate the Jump instruction. *)
    let jump = Inst.Jump !ptr in
    Array.set insts jump_pos jump
```

For Kleene star, first the position of the `Split` instruction is saved. Then, the regular expression is compiled and the `Split` instruction is generated at the saved position together with the `Jump` instruction. Once again, it closely follows the formal definition in Figure 3.2, but instead saving the position for `Split` and generating it later, rather than wastefully calculating the instruction sequence length of the sub-expression.

```
  | Regex.Star e ->
    let split_pos = !ptr in
    incr ptr;
    (* Compile e. *)
    compile e ptr insts;
    (* Generate the Jump instruction *)
    let jump = Inst.Jump split_pos in
    Array.set insts !ptr jump;
    incr ptr;
    (* Generate the Split instruction at the correct position *)
    let split = Inst.Split (split_pos+1, !ptr) in
    Array.set insts split_pos split
```

Similarly, for both `?` and `+` operators, a position required for the `Split` instruction is first saved, then the regular expression is compiled and `Split` instruction is generated. Once again it slightly differs from the formal definition in Figure 3.2. Required position used by `Split` instruction is calculated later rather than computing it immediately by unnecessary calculating the length of the sub-expression.

```
  | Regex.Opt e ->
    (* Save the position for split instruction *)
    let split_pos = !ptr in
    incr ptr;
    (* Compile e *)
    compile e ptr insts;
    (* Generate the Split instruction *)
    let split = Inst.Split (split_pos+1, !ptr) in
    Array.set insts split_pos split


  | Regex.Plus e ->
    (* Save the position for split instruction *)
    let split_beg = !ptr in
    (* Compile e *)
    compile e ptr insts;
    incr ptr;
    (* Generate the Split instruction *)
    let split = Inst.Split (split_beg, !ptr) in
```

```
      Array.set insts (!ptr - 1) split
```

3.3.1.2 Asymptotic complexity

Since each regular expression symbol is only processed once in constant time, it is apparent that the asymptotic complexity depends on the length of the given regular expression and is $O(n)$, where n is the number of symbols in the regular expression, except for the brackets.

Our particular implementation also exhibits no wasteful computation. It avoids recursively calculating the lengths of sub-expressions (or reindexing all the instructions at the end all together) by leaving the gaps in the instruction sequence during compilation when the required indexes of instructions are not known yet. Gaps get filled in immediately when the required indexes get computed.

3.3.2 Executing the instructions

The main idea is that the VM maintains the list of all threads at any given point. The VM reads characters one by one and for each of them executes all the threads.

For maintaining the list of threads, we implement `ThreadList`:

```
module ThreadList = struct
  module ThreadSet = Set.Make(Int)
  (* A ThreadList is a pair:
     - container contains Thread objects. We wrap Thread objects in
       an Option type to be able to represent the absence of a thread.
     - next keeps track of the next free cell in ‘container’.
     - added keeps track of all the threads that have already been
       added to the container *)
  type t = { container: Thread.t option array
           ; mutable next: int
           ; mutable added: bool array}

  let make : int -> t
    = fun size ->
    { container = Array.make size None
    ; next = 0
    ; added = Array.make size false }

  let add : t -> Thread.t -> unit
    = fun ts t ->
    if not (Array.get ts.added t) then
      (Array.set ts.container ts.next (Some t);
      ts.next <- ts.next + 1;
      Array.set ts.added t true)

 (* This function provides an iterator for a ThreadList. It relies on
  the fact that ‘container’ and ‘next’ are mutably updated, as the list
  might growing during the iteration *)
```

```
    let iter : (Thread.t -> unit) -> t -> unit
      = fun f ts ->
      let rec iter i f ts =
        if i < ts.next
        then ( f (Option.get (Array.get ts.container i))
             ; Array.set ts.container i None
             ; iter (i+1) f ts )
      in
      iter 0 f ts

    let clear : t -> unit
      = fun ts ->
      ts.next <- 0;
      Array.fill ts.added 0 (Array.length ts.added) false;
  end
```

ThreadList essentially keeps a list of threads and provides a way to add a new thread to the list, iterate over the list or simply clear it. It is filled with None at first and next is used to keep track where the next free space is in the list (in this case a cell with value None). To make sure that for each character each instruction is only run once, added map is used.

The function add in ThreadList first checks if the thread that is being added has not been already added to the list. This is done by checking added map. If it was not added before, then the new thread is added to the list at the place as indicated by next. next and added are updated accordingly.

The iterator for ThreadList is provided by the function iter. It applies the given function to each element of the list. By starting at index 0, it goes up the list until a free slot in the list is reached as indicated by next. Since the list of threads can grow during the iteration, it is important that both the list and next are mutably updated.

The matching happens in the function matches:

```
  let matches : Inst.t array -> string -> bool
    = fun insts str ->
    let exception Match in
    let rec exec : ThreadList.t -> ThreadList.t -> Inst.t array -> string
     -> int -> unit
      = fun clist nlist insts str sp ->
      if sp <= String.length str then
        (ThreadList.iter
           (fun thread ->
             let pc = thread in
             match Array.get insts pc with
             | Inst.Char ch ->
                if (sp < String.length str) && (String.get str sp) = ch
                then ThreadList.add nlist (Thread.make (pc+1))
             | Inst.Match ->
                if sp = String.length str then raise Match
```

```
            | Inst.Jump ptr ->
               ThreadList.add clist (Thread.make ptr)
            | Inst.Split (ptr, ptr') ->
               ThreadList.add clist (Thread.make ptr);
               ThreadList.add clist (Thread.make ptr'))
         clist;
       ThreadList.clear clist;
       exec nlist clist insts str (sp+1))
  in
  try
    let pc = 0 in
    let clist = ThreadList.make (Array.length insts) in
    let nlist = ThreadList.make (Array.length insts) in
    ThreadList.add clist (Thread.make pc);
    exec clist nlist insts str 0;
    false
  with Match -> true
```

The function `matches` takes in a list of instructions and an input string and outputs boolean value if the given string matches the regular expression that is represented by the given list of instructions. It calls the recursive function `exec` with the current list of threads (which at the start contains the first instruction in the list), a list of instructions for the next character in the input string (which initially is empty), a list of instructions, the input string itself and the index of the current character that is being processed (at the start set to zero). For each of the characters, it iterates over the current list of threads and executes the instructions one by one. Important thing is that the current list of threads continues growing during the execution of instructions.

So, for `Char` instruction, if the current character in the input string matches the character that is expected by the instruction, then a new thread is added to the list of threads that will be executed on the next iteration.

For `Match` instruction, if the whole input string has been processed, it means there is a match and the loop is exited with an exception.

For `Jump` instruction, a new thread is added to the current list of threads with the appropriate program counter (index to the instruction in the instruction list).

For `Split` instruction, two new threads are added to the current list of threads with appropriate program counters, as defined by the `Split` itself.

At the end of each iteration, the `nlist` list of threads (generated by `Char` instructions) becomes the new current list of threads, string pointer is advanced by one, and a new iteration happens.

3.3.2.1 Asymptotic complexity

Having at most n instructions means at most n threads can be active at once. In addition, by making sure that a thread can only appear in the thread list once, each thread can only be executed once for each character. Given that there are m characters in the input string and processing single character takes $O(1)$ of time, it means that the running time of this VM approach is $O(nm)$.

3.4 Integration into Links

We make a prototype of our code integration into Links, as our implementation does not currently support all the required features to fully replace Links' implementation of regular expressions. We choose to integrate our VM implementation rather than NFA, as it is more efficient and easier to extend later by adding more bytecode instructions.

First, we package our implementation as a library `vmregex` and include it in Links. Then, we make a runtime flag `--virtual-machine-regex` for using our regular expression implementation. When Links desugars regular expressions, it chooses one of the primitive binary functions to do the regular matching, and their names end in `tilde`. So, we also add a new function and name it `vtilde`. When our newly created flag is set, it is the function that gets used.

3.4.1 Server tier

For the server tier, we implement the function `vtilde` similarly as `tilde` is implemented. First, the Links regular expression is compiled into our representation of regular expressions using `Regex.compile_foreign`. The resulting `regex` is then compiled into instruction list `regex'` using our library function `Vmregex.compile`. Finally, `Vmregex.matches` from our library is called with the instruction list `regex'` and unboxed string input. The resulting bool value is then boxed.

```
(* core/lib.ml *)
("vtilde", (p2 (fun s r ->
  let regex = Regex.compile_foreign (Linksregex.Regex.ofLinks r) in
  let regex' = Vmregex.compile regex in
  let string = Value.unbox_string s in
    Value.box_bool (Vmregex.matches regex' string)),
  datatype "(String, Regex) -> Bool", PURE));
```

Here is the function `compile_foreign` that compiles from the Links representation of regular expressions (defined here as `regex`) to our representation. For this prototype, we throw an error if we encounter regular expressions that are currently not supported by our implementation. For most cases there is a direct conversion from Links representation to ours, except for `Simply` strings that have to be exploded into characters and combined using `Seq`.

```
(* core/regex.ml *)
type repeat = Star | Plus | Question
and  regex = | Range of (char * char)
           | Simply of string
           | Quote of regex
           | Any
           | StartAnchor
```

```
            | EndAnchor
            | Seq of regex list
            | Alternate of (regex * regex)
            | Group of regex
            | Repeat of (repeat * regex)
            | Replace of (regex * string)

(* function to compile from Links regex representation to ours *)
let compile_foreign : regex -> Vmregex__Regex.t = fun r ->
  let rec compile_simply = function
    | [] -> Vmregex__Regex.Eps
    | c :: [] -> Vmregex__Regex.Char c
    | c :: cs -> Vmregex__Regex.Seq
                (Vmregex__Regex.Char c, compile_simply cs) in
  let rec compile = function
    | Simply s -> compile_simply
                  (List.init (String.length s) (String.get s))
    | Seq [] -> Vmregex__Regex.Eps
    | Seq (r :: rs) -> Vmregex__Regex.Seq (compile r, compile (Seq rs))
    | Alternate (r1, r2) -> Vmregex__Regex.Alt(compile r1, compile r2)
    | Group s -> compile s
    | Repeat (Star, r) -> Vmregex__Regex.Star(compile r)
    | Repeat (Plus, r) -> Vmregex__Regex.Plus(compile r)
    | Repeat (Question, r) -> Vmregex__Regex.Opt(compile r)
    | _ -> failwith "Not implemented"
  in compile r
```

3.4.2 Client tier

For the client tier, we first compile our OCaml implementation into JavaScript code using `Js_of_ocaml` compiler [23] and include the code while rendering webpages in Links.

We implement the function `_vregexCompile` to convert Links representation of regular expressions back to their string representation.

```
function _vregexCompile(expr) {
  switch (expr._label) {
    case "Simply": return expr._value;
    case "Group": return "(" + _vregexCompile(expr._value) + ")";
    case "Seq": {
      const regexes = _$List.toArray(expr._value);
      let acc = "";
      for (let regex of regexes) {
        acc = acc + _vregexCompile(regex);
      }
      return acc;
    }
    case "Repeat": {
      let op;
      switch (expr._value[1]._label) {
        case "Plus": op = "+"; break;
        case "Star": op = "*"; break;
        case "Question": op = "?"; break;
      }
      const regex = _vregexCompile(expr._value[2]);
      return regex + op;
    }
    case "Alternate": {
      const first = _vregexCompile(expr._value[1]);
      const second = _vregexCompile(expr._value[2]);
      return first + "|" + second;
    }
    default: return "";
  }
  return null; // Can never be reached
}
```

Then, similarly as with server tier, we implement our newly added function `vtilde` in JavaScript. It calls our newly created `_vregexCompile` to get the string representation of a regular expression. After that, we first find all our library functions for parsing, compilation, and matching in the global `jsoo_runtime`. All these functions are registered in the `caml_global_data` under the library name `Vmregex`. To get a specific function from the library, we also use their position in the original code as index. In addition, we make a reference of a helper function `caml_call_gen` that is used to call our library functions and `caml_string_of_jsbytes` for converting strings into a right form. Finally having all these functions, we then parse the regular expression, compile it into a list of instructions and do the matching on the given input string.

```
(* lib/js/jslib.js *)
function _vtilde(s, regex) {
  const r = _vregexCompile(regex);
  const caller = globalThis.jsoo_runtime.caml_call_gen;
  const parser =
    globalThis.jsoo_runtime.caml_global_data['Vmregex__Regex'][2];
  const compiler =
    globalThis.jsoo_runtime.caml_global_data['Vmregex'][2];
  const matcher =
    globalThis.jsoo_runtime.caml_global_data['Vmregex'][3];
  const to_string = globalThis.jsoo_runtime.caml_string_of_jsbytes;
  const parsed = caller(parser, [to_string(r)])[1]
  const compiled = caller(compiler, [parsed])
  const matched = caller(matcher, [compiled, to_string(s)])
  return matched
}
const vtilde = _$Links.kify(_vtilde);
```

3.4.3 Example

We use the example Links program below for testing our implementation of regular expressions in Links. It uses the evil regex $\hat{}(a^+)^+\ \$$ as introduced in Chapter 2. It renders a simple page with an input box for the input string and a button to run the matching. The result of the matching is then printed in the developer console. While running with the original implementation, the browser hangs on the specifically crafted input. When we set the flag to use our implementation of regular expressions, the matching completes successfully. Despite that, our implementation is still vulnerable to evil regex that would exploit our recursive compilation function. To avoid that, we consider rewriting the parser into continuation-passing style in the future.

```
# Input: aaaaaaaaaaaaaaaaaaaaaaaaaaaaab
fun testRegex(s) client {
  var result = s =~ /^(a+)+$/;
  if (result) print("True")
  else print("False")
}
fun testRegexForm(_) {
  page <#>
    <input id="regexp" value="" />
    <button l:onclick="{testRegex(getInputValue("regexp"))}">Click
me!</button>
  </#>
}
fun main() {
  addRoute("/", testRegexForm);
  servePages()
}

main()
```

Chapter 4

Benchmarks

In this chapter we first start by discussing the methodology of executing the benchmarks and what measures were taken to reduce the noise in the results. Then, we compare the performance of two implementations as described in Chapter 3 (the NFA implementation and VM implementation) against the implementation of regular expressions in the widely used V8 JavaScript engine [12]. Version 4.08.1 of OCaml together with version 10.8.95 of V8 are used for these benchmarks.

4.1 Methodology

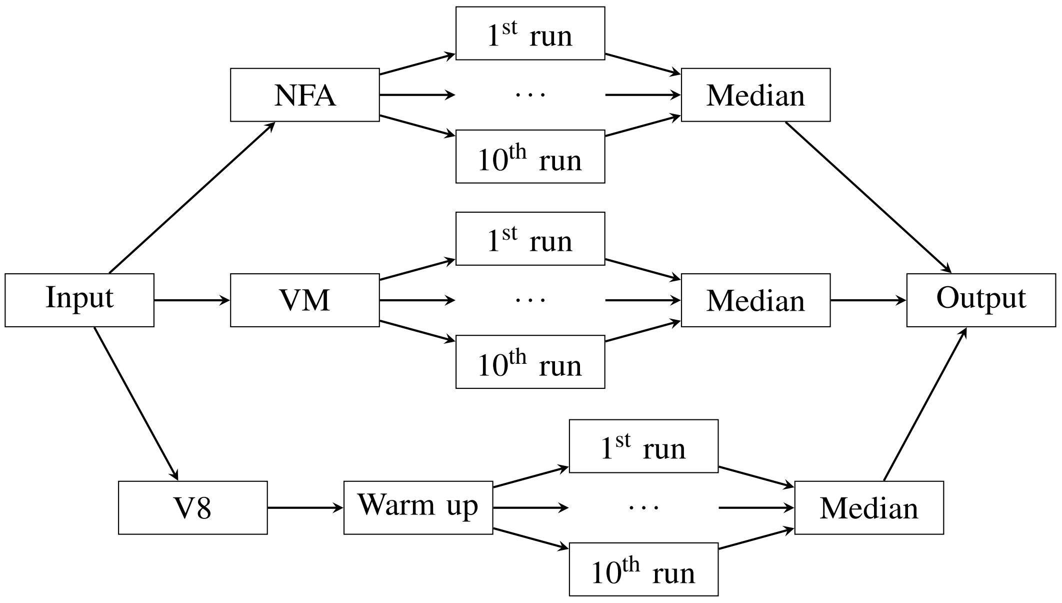


Figure 4.1: Execution of a single benchmark

There are in total 40 benchmarks, with regular expressions and string inputs stored in separate files. The experiment's pipeline is as follows (Figure 4.1): a Python script runs the benchmark programs and output their results in a CSV file. Each benchmark is run 10 times. On each run we measure both the compilation time and the execution time for the respective regex. We then summarize the result by computing the median compilation time and median execution time. We choose to use the median as our metric as it is robust against outliers (in contrast the average is sensitive to outliers).

We take some measures to eliminate potential noise from the results. Some sources of noise and the solutions:

- **Garbage collection:** both OCaml and JavaScript have garbage collection, which inadvertently increase the time of execution. Currently, both OCaml [21] and JavaScript [3] use mark-and-sweep garbage collector. The collector periodically scans the memory to identify the objects that are still in use by the program. Then, the algorithm sweeps through the memory and safely reclaims all the objects not used by the program. To minimize the impact of garbage collection on the results, we decided to considerably increase the size of the heap. Ideally, with a large enough heap size the garbage collector would not need to collect any garbage during the run time of the experiments. In OCaml, we fix the minor heap size to 4 GB by setting the environment variable `OCAMLRUNPARAM="s=4G"` [21]. In JavaScript, we fix the heap size to 4 GB by setting the runtime flag `--max-old-space-size=4096`.
- **JIT warm-up:** both OCaml and JavaScript have pretty different execution environments: OCaml gets compiled into native machine code, while JavaScript is typically executed in a virtual machine that uses a Just-in-time (JIT) compiler. We try to bridge this gap by "warming up" [2] the JIT compiler. Running the program multiple times allows for the JIT compiler to "learn" more about how the program behaves, which leads to optimizations that reduce the execution time. Thus, we run each benchmark three times before taking any measurements, which should be sufficient for the JIT compiler to warm-up and for the execution of benchmarks to be optimized as much as possible.
- **Operating system activity:** due to different background tasks that might be running together with the benchmarks, it is important to run the benchmarks multiple times. We choose to run the benchmarks ten times and take the median value for execution and compilation times. Taking a median instead of an average prevents from outliers (that appear for various reasons) significantly skewing the results.

4.2 Results

In this section we examine the performance of regular expression matching on three different kinds of expressions: simple regular expressions, evil regex as discussed in Section 2.2.4, and regular expressions that often appear in real-world use.

4.2.1 Simple regular expressions

First, we used some simple regular expressions to test the performance of implementations solely on the size of input string. This includes a^*, $[a-z]^*$, $[a-$

$z]^*somerandomwords[a-z]^*$ with results in Tables 4.1, 4.2, 4.3 respectively. We use $[a-z]$ notation for easier reading here, although in benchmarks an alternation of all the characters is used ($a \parallel b \parallel c \parallel d \parallel ... \parallel z$) across all the implementations.

It is apparent that the execution time for both NFA and VM implementations grow linearly together with the input string size, while the execution time of V8 implementation only grows logarithmically. In addition, it can be seen that the execution time of NFA starts of large in comparison to VM implementation and stays about 100 to 1000 times larger.

VM implementation has the best compilation time across all these benchmarks and the best execution time for input sizes up to around 100 characters. For larger inputs, V8 has the best execution time.

	NFA		VM		V8	
	Compile	Match	Compile	Match	Compile	Match
1 character	4	5	3	2	25	37
10 characters	4	18	3	2	25	38
100 characters	5	137	3	10	26	38
1000 characters	5	1086	3	85	24	71
10000 characters	4	7399	2	821	25	76
100000 characters	4	64798	3	5866	24	126

Table 4.1: Performance of a^* based on input size (in microseconds)

	NFA		VM		V8	
	Compile	Match	Compile	Match	Compile	Match
1 character	559	953	4	4	29	41
10 characters	555	5378	4	12	29	43
100 characters	532	46916	4	96	31	44
1000 characters	526	449908	4	942	29	86
10000 characters	>1 second		5	59466	36	313
100000 characters	>1 second		4	97536	28	1600

Table 4.2: Performance of $[a-z]^*$ based on input size (in microseconds)

	NFA		VM		V8	
	Compile	Match	Compile	Match	Compile	Match
1 character	2130	17759	6	28	32	56
10 characters	2134	30390	7	44	35	59
100 characters	2088	182292	8	227	34	66
1000 characters	>1 second		7	1840	38	123
10000 characters	>1 second		7	20322	33	348
100000 characters	>1 second		8	187103	28	2040

Table 4.3: Performance of $[a-z]^*somerandomwords[a-z]^*$ based on input size (in microseconds)

4.2.2 Evil regex

As described in the Section 2.2.4, some implementations of regular expression matching perform poorly given carefully crafted inputs. Two different regular expressions have been chosen to test this. First regular expression is $(a^+)^+$. As it can be seen in Table 4.4 and Figure 4.2, V8 performs poorly, with its execution time growing exponentially given increasing input size. In this case, 26 characters are enough for the execution time to be greater than one second. In comparison, VM implementation's execution time does not grow and stays very minimal: only a few microseconds. NFA execution time also stays the same, although it is about 100 times slower than VM implementation. So for this benchmark, the VM implementation has the best both compilation and execution time.

Another evil regex is $((((((((a \parallel a^?)^+ \parallel a^?)^+ \parallel a^?)^+ \parallel a^?)^+ \parallel a^?)^+ \parallel a^?)^+ \parallel a^?)^+ \parallel a^?)+$. It was artificially crafted to specifically break the NFA implementation. It can be seen from Table 4.5 and Figure 4.3 that NFA implementation indeed breaks on this regular expression: even on small input sizes it takes more than one second to finish execution. Comparing VM and V8 implementations, it can be seen that the execution time for V8 implementation starts high at around 5 milliseconds, while for VM implementation the execution time starts small and grows larger as the input size increases. At around 1000 characters, the execution time of VM reached the execution time of V8 implementation and then grows beyond it. So, it can be said that the VM approach has the best compilation time once again across all the benchmarks, with execution time being the best until the input size gets to around 1000 characters, where V8 then takes the lead.

	NFA		VM		V8	
	Compile	Match	Compile	Match	Compile	Match
1 character	20	4	4	2	29	46
5 characters	20	104	3	2	29	48
10 characters	20	262	4	2	32	71
15 characters	18	310	4	3	29	835
20 characters	18	417	4	3	30	26305
21 characters	20	484	4	3	30	50090
22 characters	20	509	3	4	29	108534
23 characters	20	504	4	5	30	211592
24 characters	20	532	4	5	30	435571
25 characters	21	558	3	4	29	888308
26 characters	19	560	4	5	>1 second	

Table 4.4: Performance of $(a^+)^+$ based on input size (in microseconds)

	NFA		VM		V8	
	Compile	Match	Compile	Match	Compile	Match
1 character	35078	195274	5	3	29	6948
5 characters	31820	572784	6	8	29	6842
10 characters	>1 second		5	12	29	6744
20 characters	>1 second		4	18	35	5790
30 characters	>1 second		6	43	45	6675
100 characters	>1 second		5	90	34	7549
1000 characters	>1 second		6	910	30	1191
10000 characters	>1 second		6	8952	31	1324
100000 characters	>1 second		6	128608	56	8187

Table 4.5: Performance of $((((((((a \parallel a^?)^+ \parallel a^?)^+ \parallel a^?)^+ \parallel a^?)^+ \parallel a^?)^+ \parallel a^?)^+ \parallel a^?)^+ \parallel a^?)+$ based on input size (in microseconds)

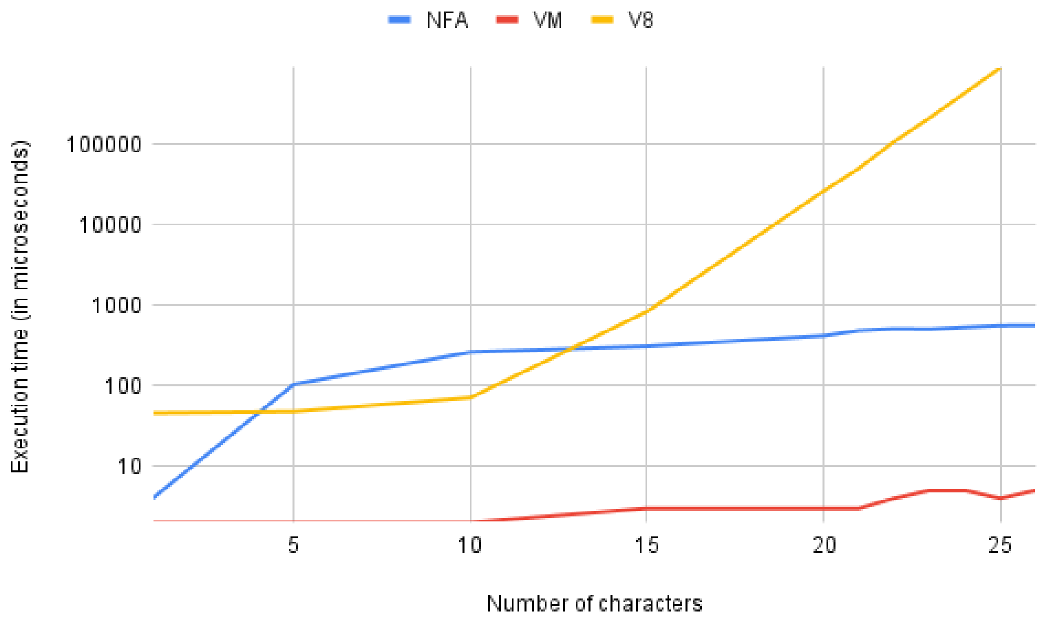


Figure 4.2: Performance of $(a^+)^+$ based on input size (in microseconds, logarithmic scale)

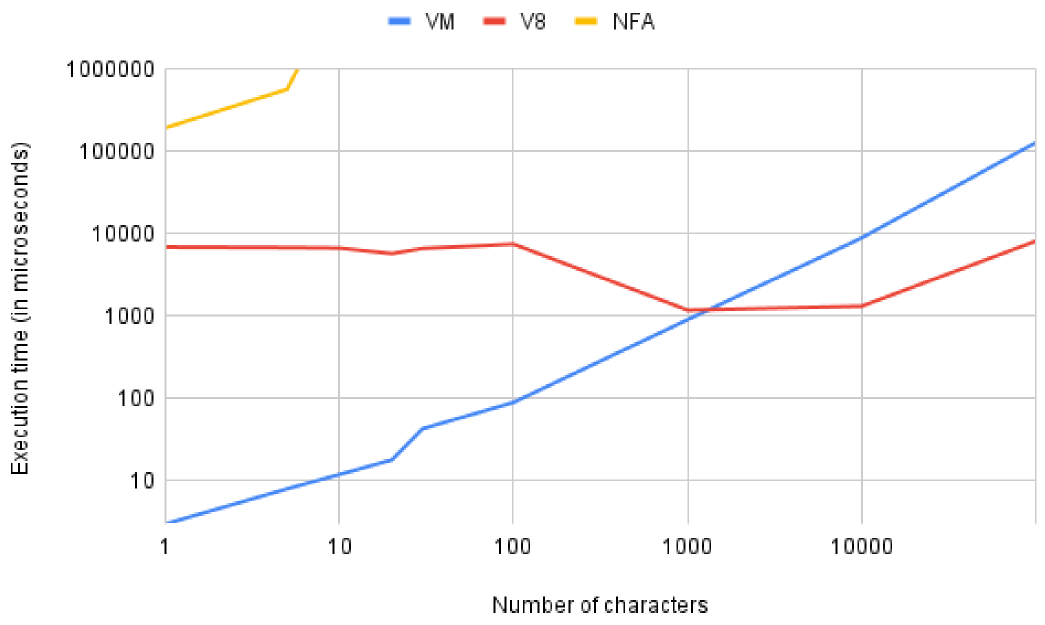


Figure 4.3: Performance of $(((((((((a \parallel a^?)^+ \parallel a^?)^+ \parallel a^?)^+ \parallel a^?)^+ \parallel a^?)^+ \parallel a^?)^+ \parallel a^?)^+ \parallel a^?)+$ based on input size (in microseconds, logarithmic scale)

4.2.3 Real world examples

Lastly, some real world examples have been used for testing the performance. This includes regular matching URLs and email addresses. As it can be seen from Table 4.6, compilation time is similar between VM and V8 implementations, while NFA implementation takes about 100 times longer to compile. In terms of the execution time, once again VM and V8 implementations are similar in performance, while NFA implementation is about 100 to 1000 times worse than both. Overall, it is apparent that the VM implementation outperforms other implementations in both compilation and execution time.

	NFA		VM		V8	
	Compile	Match	Compile	Match	Compile	Match
Email address	3388	54664	10	69	70	109
URL	3853	30018	6	36	38	61

Table 4.6: Performance of real world examples (in microseconds)

Chapter 5

Conclusion

In this chapter, we will summarize both implementations of regular expressions and how they perform relative to each other and V8 JavaScript engine. In addition, we will outline the potential future work and what could be improved.

5.1 Summary

This report presents two implementations of regular expressions in OCaml. First approach is based on NFA and Thompson's construction. Thompson's construction algorithm converts the given regular expression into an NFA using the five construction rules. The resulting NFA's number of states is proportional to the number of symbols in the regular expression. By simulating NFA and only tracking the states that are active rather than the paths being taken, it is possible to reach the asymptotic complexity that is linear to the length of the regular expressions times the length of input string.

Second approach to implementing regular expressions is essentially lowering the NFA approach as a virtual machine, where each thread corresponds to an exploration of a distinct choice in the NFA transition graph. Regular expressions get compiled into instructions given predefined rules. Then, VM executes the instructions while keeping track of the threads that are spawned by the instructions. By executing the threads in lock-step while reading characters one-by-one, it is possible to reach the asymptotic complexity that is linear to the length of the regular expressions times the length of input string.

After implementing the regular expressions in both of these approaches, we measured and compared the performance of both of them in comparison to the widely used V8 JavaScript engine. It appears that for simple regular expressions, such as a^*, NFA performs poorly, VM performs the best until the input size gets to around 1000 characters, from then on V8 has the best execution time from all the implementations, with VM approach's execution time growing linearly with the input string length. The poor scaling of VM approach might be due to how we implemented the handling of the input string. For evil regex, it can be seen that there exists some regular expressions, for example $(a^+)^+$, that are able to break the V8 engine, while both NFA and VM continue to work correctly. Nevertheless, there exist regular expressions that also break the NFA approach, so overall VM is the most resilient against evil regular expressions. Lastly, for real-world examples, such as matching emails and URLs, it appears that VM approach is quite competitive in comparison to V8 implementation, as it has its execution time matching or lower than V8.

Finally, as a proof of concept, we integrated our implementation into Links. By creating an example Links program with an evil regex $(a^+)^+$, we show that our implementation can successfully finish regular matching, while the original implementation would hang the browser.

5.2 Future work

There are multiple interesting future directions to explore following from this dissertation work.

Alternative implementation techniques Having more implementations of regular expressions, especially ones that have a completely different approach to solving regular expression matching and does not involve NFA. This could include Brzozowski derivative, which was introduced at the end of 1950s by multiple authors [27, 26, 8]. Brzozowski derivative can be used to construct regular expression first without generating NFA. The idea is to define a derivative for a language that could be repeatedly applied for each symbol in the alphabet. This leads to an algorithm that can be used to decide whether an input string is in the given language described by the regular expression.

Performance analysis Investigating the cause of poorer performance than expected of the VM approach. It was expected that the performance of VM approach would stay closer to the V8 implementation for larger inputs. This might be due to inefficiencies in handling the string input in our implementation. In addition, we expected for the NFA implementation to be closer to the VM implementation. This discrepancy could be possibly explained by the inefficient implementation in how transitions in the NFA are calculated.

Database tier Implementing regular expressions in the database tier and making sure that the engine matches what both server tier and client tier have. This could be achieved by compiling the OCaml source code not only to JavaScript for the client tier, but also to SQL for the database tier. The compilation for the database tier would be more complex, in contrast to using the existing compiler from OCaml to JavaScript for the client tier.

Extending the regular expression language Expanding the feature set of the current implementations. Current implementations as described in Chapter 3 only support a limited number of special symbols. There is a potential for such constructs:

- **Character classes:** character classes are used to more easily match a specific set of characters without the need to use alternations. For example $[a-z]$ could be used to match all lower case ascii letters. Another popular symbol is a dot ., which matches against any character.

- **Anchors:** anchors allow matching specific position in the input string. For example, an anchor symbol ˆ matches the beginning of the line, while anchor symbol $ matches the end of the line.
- **Backreferences:** backreferences allow to reference a previously matched group within the same expression. The matched text can then be used as a variable. For example $(abc)\backslash 1$ would match *abcabc*, as $\backslash 1$ references (abc).
- **Lookarounds:** lookarounds are used for matching a pattern if and only if it is followed or preceded by other pattern. For example $((? <= abc)def)$ only matches def if it is preceded by *abc*.
- **Escaped characters:** currently, special characters such as $*$ and $+$ are reserved, but it would be useful to have an ability to actually match these characters in text as well.

Bibliography

[1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers, Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[2] Edd Barrett, Carl Friedrich Bolz-Tereick, Rebecca Killick, Sarah Mount, and Laurence Tratt. Virtual Machine Warmup Blows Hot and Cold. *Proceedings of the ACM on Programming Languages*, 2017.

[3] MDN contributors. JavaScript memory management reference. `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Memory_Management`, 2023. Accessed March 23, 2023.

[4] Ezra Cooper, Sam Lindley, Philip Wadler, and Jeremy Yallop. Links: Web Programming Without Tiers. In *FMCO*, volume 4709 of *LNCS*, pages 266–296. Springer, 2006.

[5] Ezra Cooper, Sam Lindley, Philip Wadler, and Jeremy Yallop. The Essence of Form Abstraction. In *APLAS*, volume 5356 of *Lecture Notes in Computer Science*, pages 205–220. Springer, 2008.

[6] Russ 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`, 2007. Accessed March 13, 2023.

[7] Russ Cox. Regular Expression Matching: the Virtual Machine Approach. `https://swtch.com/~rsc/regexp/regexp2.html`, 2007. Accessed March 13, 2023.

[8] Calvin C Elgot and Joseph D Rutledge. Operations on finite automata. In *2nd Annual Symposium on Switching Circuit Theory and Logical Design (SWCT 1961)*, pages 129–132. IEEE, 1961.

[9] Simon John Fowler. *Typed concurrent functional programming with channels, actors and sessions*. PhD thesis, The University of Edinburgh, Scotland, UK, 2019.

[10] Free Software Foundation, Inc. Gnu grep manual. `https://www.gnu.org/software/grep/manual/grep.html`, 2023. Accessed March 23, 2023.

[11] Victor Mikhaylovich Glushkov. The abstract theory of automata. *Russian Mathematical Surveys*, 16(5):1, 1961.

[12] Google. V8. `https://v8.dev/`, 2023. Accessed March 23, 2023.

[13] Jonathan L Gross and Jay Yellen. *Handbook of graph theory*. CRC press, 2003.

[14] Daniel Hillerström and Sam Lindley. Liberating effects with rows and handlers. In *Proceedings of the 1st International Workshop on Type-Driven Development*, pages 15–27, 2016.

[15] Daniel Hillerström. *Foundations for Programming and Implementing Effect Handlers*. PhD thesis, The University of Edinburgh, Scotland, UK, 2022.

[16] John E Hopcroft, Rajeev Motwani, and Jeffrey D Ullman. Introduction to automata theory, languages, and computation. *ACM Sigact News*, 32(1):60–65, 2001.

[17] Rudi Horn. *Language integrated relational lenses*. PhD thesis, The University of Edinburgh, Scotland, UK, 2023.

[18] Hans-Arno Jacobsen. *Middleware 2004: ACM/IFIP/USENIX International Middleware Conference, Toronto, Canada, October 18-20, 2004, Proceedings*, volume 3231. Springer, 2004.

[19] Stephen Cole Kleene. Representation of events in nerve nets and finite automata. In Claude Shannon and John McCarthy, editors, *Automata Studies*, pages 3–41. Princeton University Press, Princeton, NJ, USA, 1956.

[20] Sam Lindley and J Garrett Morris. A semantics for propositions as sessions. In *Programming Languages and Systems: 24th European Symposium on Programming, ESOP 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015, Proceedings 24*, pages 560–584. Springer, 2015.

[21] Anil Madhavapeddy, Jason Hickey, and Yaron Minsky. *Real World OCaml: Functional Programming for the Masses*. Cambridge University Press; 2nd edition, 2022.

[22] R. McNaughton and H. Yamada. Regular Expressions and State Graphs for Automata. *IEEE Transactions on Electronic Computers*, 1960.

[23] Ocsigen. Js_of_ocaml. `https://ocsigen.org/js_of_ocaml/latest/manual/overview`, 2023. Accessed March 23, 2023.

[24] University of Cambridge. Advanced Functional Programming. `https://www.cl.cam.ac.uk/teaching/1718/L28/`, 2017. Accessed March 23, 2023.

[25] Ken Pizzini, Paolo Bonzini, Jim Meyering, and Assaf Gordon. GNU sed, a stream editor. *Abgerufen am*, 7, 2018.

[26] Michael O Rabin and Dana Scott. Finite automata and their decision problems. *IBM journal of research and development*, 3(2):114–125, 1959.

[27] George N Raney. Sequential functions. *Journal of the ACM (JACM)*, 5(2):177–180, 1958.

[28] Michael Sipser. *Introduction to the theory of computation. Third edition.* Australia : Cengage Learning, 2013.

[29] Michael Stonebraker, Paul Brown, and Dorothy Moore. *Object-relational DBMSs: Tracking the Next Great Wave*. Morgan Kaufmann Publishers, 1999.

[30] Ken Thompson. Programming Techniques: Regular expression search algorithm. *Communications of the ACM*, 1968.

[31] Adar Weidman. Regular expression Denial of Service - ReDoS. `https://owasp.org/www-community/attacks/Regular_expression_Denial_of_Service_-_ReDoS`, 2020. Accessed March 23, 2023.