A Crash-Course in Regular Expression Parsing and Regular Expressions as Types

Built: August 18, 2014

Contents

1 Introduction

Suppose we need to proof-read a large piece of text. After a little while, we begin to notice a few recurring mistakes:

- The text frequently mentions the former Libyan ruler Muammar Gaddafi, but spells his name inconsistently: for example, by spelling it "Moammar Qadafi", or one of the many other possible variations due to the ambiguity of Arabic romanization [1].
- The text contains several numerals, but arbitrarily switches between using comma and dot as a decimal mark (the symbol separating the decimal part from the fractional part of a number). For example, switching between the two styles "123,5" and "123.5".

We would like to correct the text such that it uses a single spelling of "Muammar Gaddafi", and to use "." as a decimal mark, which is the usual convention in written English.

Manually going through the manuscript to correct all occurrences of the above mistakes can be a labor-intensive task, depending on the size of the text. If we are using a text editor with search/replace features, we might be able to replace all occurrences of "Qadafi" with "Gaddafi". But, since that is just one out of more than a hundred possible spellings [1], replacing them all would require many applications of the search/replace procedure. Finding and fixing all occurrences of "," used as a decimal mark is even more work, as we only want to replace the commas that occur as part of a numeral, and not those that occur as a grammatical separator of text clauses.

Fortunately, *regular expressions* can be used to solve the above problems. A regular expression can be seen as a formal *description* of a set of strings satisfying certain criteria. For example, we might write a regular expression which describes the set of all strings that begins with one of the letters "Q", "K" or "G", followed by either "adafi" or "addafi". This already describes 6 different ways of spelling "Gaddafi", including the one that we consider to be correct. If our text editor supported search/replace by regular expressions instead of just constant strings, we can ask it to replace all substrings that are matched by our Gaddafi-expression with the string "Gaddafi" – but now six variations are corrected at once, saving us from a lot of work.

Even though regular expressions are not expressive enough to describe *all* sets of strings in the universe, they still cover a lot of practical use cases similar to the ones mentioned above. Additionally, they have some appealing theoretical properties that leads to very efficient software implementations. In the following section, we will describe them in more detail.

2 Regular Expressions

The notion of regular expressions arose from the field of theoretical computer science, and was first described in 1956 by Stephen Cole Kleene [11] in his work on finite automata theory. The ubiquitous interpretation of regular expressions is as *patterns* denoting (possibly infinite) sets of strings. Under this interpretation, we can formulate the problem of determining whether a given string is contained in the underlying set specified by some regular expression. This generalizes the simpler string matching problem of determining whether a specific string occurs as a substring in a larger text. Concrete text search implementations emerged a relatively long while after Kleene's introduction, with one of the earliest appearances being Thompson's description of a text search algorithm based on regular expressions from 1968 [18]. The motivation for using regular expressions for simple text search is that they are computationally weak enough to provide strong guarantees of efficient running time and memory use, which is not the case for more expressive formalisms such as *context free grammars* [5].

Regular expressions are now well-known among programmers and power-users, and have found use in many applications such as text editing, lexical analysis in compilers and scripting. Popular implementations include UNIX tools such as `sed`, `grep` and `awk`, text editors such as `emacs`, and some programming languages even include them as first-class constructions (Perl [19] being a notable example, although not the only one). What these implementations have in common is that they all implement the traditional regular expressions of Kleene, although some choose to add *extensions*, yielding a much more expressive (and thus computationally much more complex) formalism. A notable example is again Perl which adds *back-references* [8], making the matching problem NP-complete! Additionally, the various implementations have slightly different conventions for the external syntax, mostly with regard to escaping. To avoid confusion when talking about regular expressions, we will therefore not talk about a specific implementation, but instead use a more mathematical notation.

Definition 1. *We say that E is a* regular expression *(RE) if either:*

1. *E is an* atomic expression*, consisting of a single letter a from some set of letters Σ, or the special symbol 1 which does not occur in Σ.*

2. *Given REs E_1 and E_2, E is a* compound expression *of the form $E_1 + E_2$, E_1E_2, or $E_1^\star$.*[1]

[1] These compound expressions are often written `E|F`, `EF`, `E*` in actual software implementations, where `E` and `F` are regular expressions.

□

The set Σ, called an *alphabet*, is assumed to be a finite set of "letters", which varies depending on our application. For example, for text search in a document encoded using ASCII [2], Σ will be defined as the set of printable characters in the ASCII table. We will sometimes add parentheses to REs to make their meaning unambiguous: to distinguish parentheses from letters, we will write all literal letters underlined with a `typewriter font`.

Given an RE E, we want to be able to talk about the set of strings that it represents, the so-called *language* of E, denoted by $\mathcal{L}(E)$. $\mathcal{L}(\cdot)$ can simply be thought of as a function sending a regular expression to its interpretation as a set of strings. For the atomic expressions, the interpretation is straightforward:

$$\mathcal{L}(a) = \{a\},$$
$$\mathcal{L}(1) = \{\epsilon\}.$$

Thus, the RE a (for some $a \in \Sigma$) represents a singleton set containing the one-letter string a. In the above, we use ϵ to denote the *empty string*, that is, the string consisting of no letters at all. Thus, the RE 1 represents the singleton set containing just the empty string. Note that this is *not* the same as the empty set!

The interpretation of a compound expression of the form $E_1 + E_2$ is defined directly in terms of set union and the interpretations of E_1 and E_2:

$$\mathcal{L}(E_1 + E_2) = \mathcal{L}(E_1) \cup \mathcal{L}(E_2).$$

So, the $+$ operator simply represents the merging of the languages it operates on. The situation for expressions of the form E_1E_2 is a little bit more involved:

$$\mathcal{L}(E_1E_2) = \{w_1w_2 \mid w_1 \in \mathcal{L}(E_1), w_2 \in \mathcal{L}(E_2)\}.$$

In the above, we use w_1, w_2 to range over strings (or *words*) in the languages for E_1 and E_2. Writing two words w_1, w_2 side-by-side denotes their concatenation w_1w_2. Thus, informally, the interpretation of E_1E_2 is the set of all words that can be formed by prepending all words from the language of E_1 to all the words from the language of E_2.

We will get back to defining the meaning of REs of the form $E_1^\star$ in a moment, but first we present an example of a simple RE which provides the first step towards solving the first problem from the introduction: finding misspellings of the name "Gaddafi". Consider the following RE, whose interpretation is a set of 192 different ways to spell "Gaddafi" (including the correct spelling):

$$E_g = (\underline{\texttt{Q}} + \underline{\texttt{G}} + \underline{\texttt{K}})(\underline{\texttt{a}} + \underline{\texttt{u}})(\underline{\texttt{d}} + \underline{\texttt{t}})(\underline{\texttt{h}} + 1)(\underline{\texttt{d}} + \underline{\texttt{t}})(\underline{\texttt{h}} + 1)\underline{\texttt{af}}(\underline{\texttt{i}} + \underline{\texttt{y}})$$

For example, we have $\underline{\texttt{Gaddafi}} \in \mathcal{L}(E_g)$ and $\underline{\texttt{Qutthafy}} \in \mathcal{L}(E_g)$, which can easily be seen by using the rules introduced above.

There is a reason why the above notation is not used in practical implementations: it is very verbose. Using the syntax of Perl, the above RE can be written `[QGK][au][dt]h?[dt]h?af[iy]`. To get a detailed explanation of the meaning of this expression, try pasting it into a tool such as Regex101 [15] (make sure "PCRE" is set as syntax). Note that this is still the exact same expression as E_g, but using the syntactic sugar of Perl to make it shorter.

The second problem from our introduction, finding uses of "," used as a decimal mark, requires the last compound expression form: REs of the form $E_1^\star$. The reason for this is that the set of numerals containing "," is *infinite* (since the set of fractional numbers is infinite), and up until now, we have only seen how to represent *finite* sets of strings using REs.

First, we define the notion of *exponentiation* of an RE: for any RE E and number $n > 0$, we define

$$E^n = \underbrace{EE...E}_{n \text{ times}},$$

that is, E^n is simply the concatenation of E with itself, n times. Just as exponentiation of real numbers is "repeated multiplication", this exponentiation is repeated concatenation. Furthermore, just as $a^0 = 1$ for any $a \in \mathbb{R}$, $E^0 = 1$. Hence, specifying something as an expression repeated zero times is the same as specifying the empty string. Given an RE of the form $E_1^\star$, we then define

$$\mathcal{L}(E_1^\star) = \{\epsilon\} \cup \bigcup_{n=1}^{\infty} \mathcal{L}(E_1^n) = \bigcup_{n=0}^{\infty} \mathcal{L}(E_1^n).$$

That is, the interpretation of $E_1^\star$ is the union of the interpretations of any number of repetitions of E_1, plus the empty string. Let us put this to use, and define the set of numbers which use "," as a decimal mark:

$$E_d = (\underline{0} + \underline{1} + ... + \underline{9})^\star \underline{,} (\underline{0} + \underline{1} + ... + \underline{9})(\underline{0} + \underline{1} + ... + \underline{9})^\star$$

Again, we can easily use the definition to verify that $\underline{\texttt{123,5}} \in \mathcal{L}(E_d)$ and $\underline{\texttt{00000,0}} \in \mathcal{L}(E_d)$. However, we do *not* have $\underline{\texttt{123,}} \in \mathcal{L}(E_d)$: if the "," is not followed by a digit, it is not necessarily in a wrong context.

For easy reference, we repeat here the full definition of the syntax and language interpretation of regular expressions:

Definition 2. *The syntax of regular expressions is described by the grammar:*

$$E ::= \underline{a} \mid 1 \mid E_1 + E_2 \mid E_1 E_2 \mid E_1^\star,$$

where it is assumed that a finite alphabet Σ is given and that $\underline{a} \in \Sigma$. □

Definition 3. *The* language interpretation $\mathcal{L}(\cdot)$ *of a regular expression is defined as:*

$$\begin{aligned}
\mathcal{L}(\underline{a}) &= \{\underline{a}\} \\
\mathcal{L}(1) &= \{\epsilon\} \\
\mathcal{L}(E_1 + E_2) &= \mathcal{L}(E_1) \cup \mathcal{L}(E_2) \\
\mathcal{L}(E_1 E_2) &= \{w_1 w_2 \mid w_1 \in \mathcal{L}(E_1), w_2 \in \mathcal{L}(E_2)\} \\
\mathcal{L}(E^\star) &= \{\epsilon\} \cup \bigcup_{n=1}^{\infty} \mathcal{L}(E)^n = \bigcup_{n=0}^{\infty} \mathcal{L}(E)^n
\end{aligned}$$

$$\mathcal{L}(E)^n = \underbrace{\mathcal{L}(E)\mathcal{L}(E)\ldots\mathcal{L}(E)}_{n \text{ times}}.$$

□

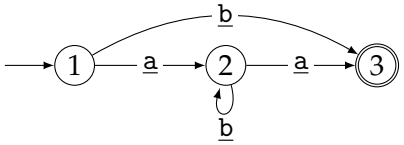
We will use summation notation as a shorthand for large alternatives. For example, given a finite subset of symbols $S \subseteq \Sigma$ where $S = \{a_1, a_2, ..., a_n\}$, we will write $\sum_{a \in S} a$ for the expression $a_1 + a_2 + ... + a_n$. To reduce notational clutter in later examples, we introduce the following definitions:

$$\begin{aligned}
E_{digit} &= \sum_{a \in \{\underline{0},\underline{1},\ldots,\underline{9}\}} a \\
E_{alpha} &= \left(\sum_{a \in \{\underline{\mathtt{a}},\underline{\mathtt{b}},\ldots,\underline{\mathtt{z}}\}} a \right) + \left(\sum_{a \in \{\underline{\mathtt{A}},\underline{\mathtt{B}},\ldots,\underline{\mathtt{Z}}\}} a \right)
\end{aligned}$$

3 Regular Expression Matching

We have now seen the definition of regular expressions, and their interpretation as sets of strings. If we want to be able to use regular expressions as patterns in text search applications, we need a procedure which, given any RE E and string w can answer the question of whether $w \in \mathcal{L}(E)$ – also called the *acceptance problem*. In practical applications, we need a bit more information than that, namely also information about *how* w is matched by E. For now, we will just focus on the acceptance problem, however, and will then proceed to show how the methods can be extended.

Most algorithms for solving the acceptance problem are based on *finite state machines* (FSMs) also called *finite automata*, which are closely related to regular expressions. A simple example of a finite automaton can be seen below:

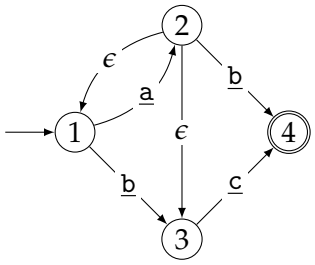


Every circle in the diagram above is called a *state*, with the numbers acting as (unique) identifiers. Arrows are called *transitions*, and are marked by *labels*, which in this case constitute letters from some alphabet. In the above, the state 1 is the *initial state*, which is denoted by the orphaned incoming arrow, and the state 3 is the *accepting state*, denoted by a double circle.

Like regular expressions, finite automata can be seen as describing sets of strings. A string w is said to be in the set described by a given automaton if it *accepts* w. Acceptance is determined by *running* the automaton with w as input, as described in the following. An automaton is always in one of its states, which is referred to as the *active state*. Before the automaton starts to process the input string, it will be in the initial state. The automaton then repeatedly removes the first symbol of w, using its value to determine the next active state. If the active state has an outgoing transition labeled with the current input symbol, the automaton transitions to the destination state, which then becomes the active state for the next input symbol. If, when there are no more symbols to be read, the active state is the accepting state, then the automaton is said to *accept* w, meaning that w is in the set of strings described by the automaton. Otherwise, w is *rejected*. If, at any point, the currently active state does not have a transition labeled with the next input symbol, the automaton halts and rejects the whole string.

For example, the automaton above accepts the string `aba`: we can get from 1 to 2, reading `a`, from 2 to 2, reading `b`, and finally from 2 to 3, reading `a`. Similarly, we can check that the automaton also accepts the strings `b` and `aa`, but not `aab`, since the only way to read `aa` will take us to state 3, and there is no way to get from 3 to any other state reading `b`.

An automaton like the above is called a *deterministic finite automaton* (DFA), since the active state can transition to at most one successor state for any given input symbol. In general though, a state may have multiple outgoing transitions with the same label. Additionally, transitions may be labeled using the special label ϵ (*epsilon*), which denotes a "free transition" which can be taken regardless of the current input symbol, and which does not consume it either. For example, the following is also an automaton:



Such a construction is called a *non-deterministic finite automaton* (NFA), since we may have to choose between multiple state candidates when performing a state transition. Since not all of the possible candidates might lead us to the accepting state, we will have to try them all, in parallel. We do this by generalizing the concept of an active state to *state sets*: in each step of the NFA simulation, we maintain a set S of states, all considered to be active *at once*. The initial state set is just singleton set containing only the initial state, and successor state sets are calculated by finding *all* possible successor states resulting from reading the current symbol while in any of the states from the current state set. The NFA accepts if the final state set contains the accepting state: this implies that there must exist at least one path through the NFA which reads the input string.

For example, to verify that abc we simulate the NFA as follows:

1. Let the initial state set be $\{1\}$.
2. Read a: Then $1 \xrightarrow{\mathtt{a}} 2$ is the only possible transition, so the next state set is $\{2\}$.
3. Read b: We can take the transitions $2 \xrightarrow{\mathtt{b}} 4$ and $2 \xrightarrow{\epsilon} 1 \xrightarrow{\mathtt{b}} 3$, yielding the state set $\{4, 3\}$.
4. Read c: We have $4 \not\to$ but $3 \xrightarrow{\mathtt{c}} 4$, yielding the state set $\{4\}$.
5. No more symbols. Since 4 is in the final state set: accept.

In the next section, we will describe a method for constructing finite state automata from REs. The method will not, in general, result in a DFA. However, any NFA can be converted to a DFA describing the same language, but the conversion might lead to an exponential blow-up in size. Hence, we have a trade-off situation between having a less efficient acceptance algorithm versus spending a lot of time (and memory!) on a preprocessing phase. In many cases, in particular the case where the input string is *very* large, the extra preprocessing might be worth it, though.

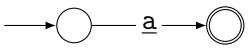
3.1 NFA Construction

There are many approaches to constructing a finite automaton from regular expressions. The one we will present here is Thompson's algorithm [18].

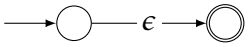
Thompson's algorithm works by constructing an NFA from an RE *bottom-up*: atomic expressions are converted first, and compound expressions are converted by combining the automata resulting from converting subexpressions.

For any RE E, the conversion proceeds by cases on the form of E. The cases for atomic expressions are simple:

- Case $E = \underline{\mathtt{a}}$. Then we construct the automaton

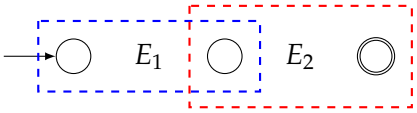


- Case $E = 1$. We construct the automaton



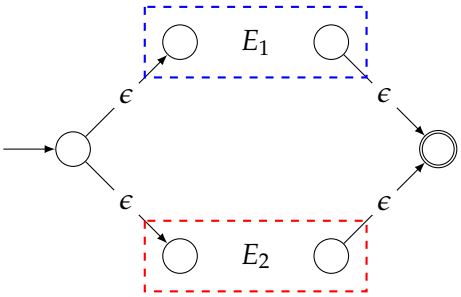
The translations are easily seen to be correct, in that the resulting NFA describes the same language as the original expression. For compound expressions, we will use boxes to denote the automata resulting from converting subexpressions.

- Case $E = E_1 E_2$. We construct:



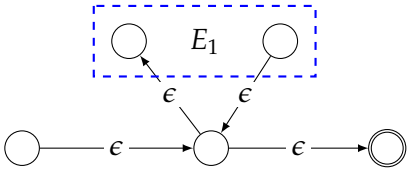
  That is, we merge the final state of the NFA for E_1 with the initial state for E_2, and let the initial state in E_1 be our new initial state, and the final state in E_2 be our new final state.

- Case $E = E_1 + E_2$. We construct:



The sum of two REs is handled by running their respective NFAs in parallel.

- Case $E = E_1^\star$. We construct:

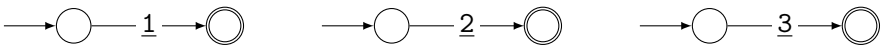


A Kleene star introduces a loop structure in the NFA.

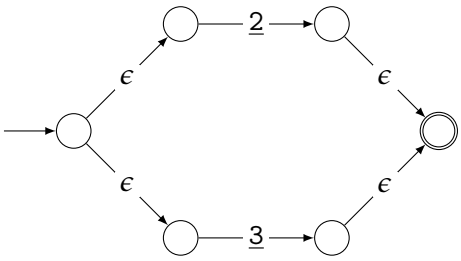
Example 1. Consider the expression

$$E_{123} = \sum_{a \in \{\underline{0},\underline{1},\underline{2}\}} a = \underline{1} + (\underline{2} + \underline{3}),$$

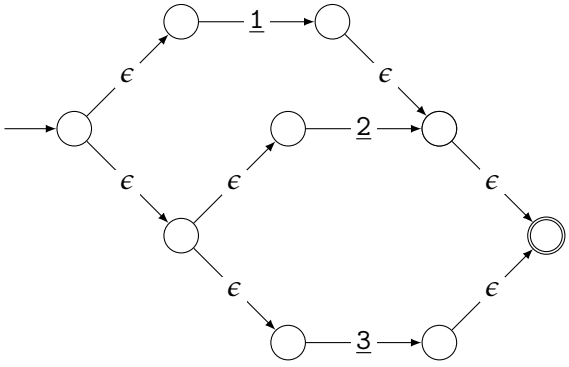
a simplified version of E_{digit}. The Thompson NFA for this expression is obtained by constructing the NFAs for $\underline{1}$ and $\underline{2} + \underline{3}$ and combining them using the rules above. In turn, the NFA for $\underline{2} + \underline{3}$ is constructed by combining the NFAs for $\underline{2}$ and $\underline{3}$. First we need the three "primitive" NFAs:



Combine number two and three using the rule for $+$:



Combine this with the automaton for $\underline{1}$ using the same rule:

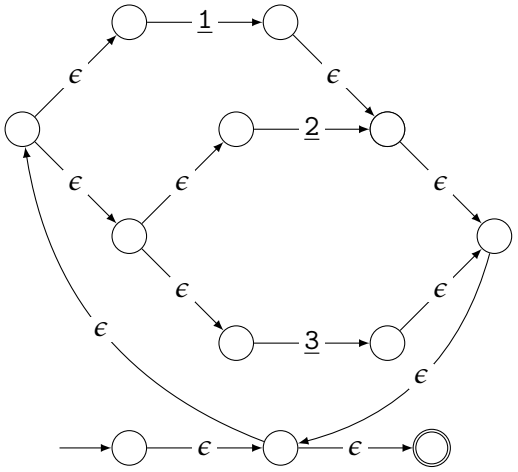


□

Example 2. Let us construct the NFA for the expression

$$E_{123s} = (E_{123})^\star = (\underline{1} + (\underline{2} + \underline{3}))^\star.$$

The NFA for the subexpression E_{123} is known from Example 1, so what remains to be done is simply to apply the rule for Kleene star:



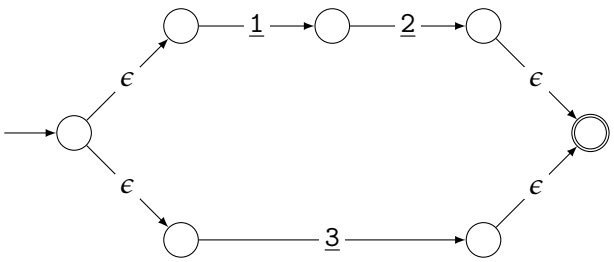
□

Example 3. The expression

$$E_{123'} = \underline{12} + \underline{3}$$

is constructed by taking the primitive NFAs for $\underline{1}$ and $\underline{2}$ and combining them with the rule for concatenation:

We can now build the NFA for $E_{123'}$ by combining this NFA with that of $\underline{3}$:



□

4 Structure in Regular Expressions

Being able to solve the more general problem of regular expression parsing allows for more advanced applications of regular expressions in text editing. Consider the scenario where we have a large text mixing numeral styles such as "1,234,567.89" and "1.234.567,89": that is, it inconsistently switches between writing numerals *with digit grouping* using the styles common in English and Scandinavian writing, respectively. Regular expressions for denoting the set of Scandinavian and English style numerals with digit grouping can be formulated as

$$E_{numS} = (\underbrace{(E_{digit}E^{\star}_{digit})}_{1}\ \underline{.})^{\star}\ \underbrace{(E_{digit}E^{\star}_{digit})}_{2}\ \underline{,}\ \underbrace{(E_{digit}E^{\star}_{digit})}_{3}$$

$$E_{numE} = (\underbrace{(E_{digit}E^{\star}_{digit})}_{1}\ \underline{,})^{\star}\ \underbrace{(E_{digit}E^{\star}_{digit})}_{2}\ \underline{.}\ \underbrace{(E_{digit}E^{\star}_{digit})}_{3}$$

Correcting all Scandinavian style numerals to use English style is not easily formulated as a regular expression matching problem, since there can be an arbitrary amount of thousands separator symbols in a single numeral, and we cannot relate specific parts of the input string to specific parts of the regular expression with a high enough precision. Solving the sub-matching problem gives us, for any sub-expression, the positions in the input string where the given sub-expression matched. Ambiguity in this answer arises when a sub-expression occurs under a Kleene star, as group 1 does in the example above. Matching E_{numS} against the input `1.234.567,89` yields two sub-matches for group 1:

$$\underbrace{1}_{1}\ \underline{.}\ \underbrace{234}_{1}\ \underline{.}\ \underbrace{567}_{2}\ \underline{,}\ \underbrace{89}_{3}$$

An ad-hoc fix would be to generalize the matching problem to return a *list* of match positions for each sub-expression. However, this would not work well in general, since ambiguity would again arise in the case where we are interested in multiple sub-expressions under one or more Kleene stars. For example, consider the following RE denoting a language of lists of key/value pairs with nullable values:

$$E_{pairs} = (\underline{(}\ \underbrace{E_{alpha}}_{1}\ \underline{,}\underbrace{(E_{digit}}_{2}\ +\underline{\texttt{NULL}})\underline{)})^{\star}$$

Performing sub-matching on the input `(a:1)(b:NULL)(c:2)` gives a list of matches `a`, `b`, `c` for group 1, and a list of matches `1`, `2` for group 2. However, the relationship between the two results is lost, since there is not way to tell

which of the keys a, b, c were associated with a null value. It can be argued that our choice of groups are a bit silly, since the problem disappears by extending group 2 to cover the right alternative as well. However, by doing that, a sub-matching implementation has to store the text NULL for each null value, which results in wasted memory. If we are parsing a huge key/value database where most values are null, the overhead is not insignificant.

As we can see, sub-matching inherently throws away useful information about the result of a match. To solve this, we generalize the problem of sub-matching to the problem of *full parsing*. Informally, the problem can be stated as follows: given an RE E and an input w, determine if w is in the language denoted by E. If it is, return a *parse tree* specifying exactly *how* w is matched by E.

Formulating this problem precisely requires us to introduce a bit more formalism, which will be covered in the following section.

5 Regular Expressions as Types

There is a correspondence between the structure of regular expressions and a certain group of *type constructors*. Intuitively, we can say that a regular expression contains some *structure*, and this structure corresponds to a *type*.

The structure described by a regular expression arises naturally when one thinks of the "meaning" of the different ways of building regular expressions:

- Concatenation expresses sequence, i.e., that something comes before something else.
- Alternation expresses a choice between the left hand side and the right hand side of the $+$.
- Kleene star expresses any finite number of repetitions of something.

The primitive symbols represent just themselves, and the special expression 1 represents "nothing", in a manner that will be specified later.

A *type* is something that describes the "shape" of objects in a set. This shape is exactly what is hinted at above, and considered like this the regular expressions can be regarded as types. We write the types in the same way as the normal regular expressions. Analogously to how a regular expression denotes a set of strings, a regular expression interpreted as a type denotes a set of *structured values*. These are denoted using the notation $\mathcal{T}(\cdot)$.

Definition 4. *The set of* structured values *denoted by an RE type E is the set $\mathcal{T}(E)$ defined as:*

$$\begin{aligned}
\mathcal{T}(1) &= \{()\} \\
\mathcal{T}(0) &= \emptyset \\
\mathcal{T}(\underline{a}) &= \{a\} \\
\mathcal{T}(E_1 E_2) &= \mathcal{T}(E_1) \times \mathcal{T}(E_2) \\
&= \{\langle v_1, v_2 \rangle \mid v_1 \in \mathcal{T}(E_1), v_2 \in \mathcal{T}(E_2)\} \\
\mathcal{T}(E_1 + E_2) &= \mathcal{T}(E_1) + \mathcal{T}(E_2) \\
&= \{\mathsf{inl}\ v_1 \mid v_1 \in \mathcal{T}(E_1)\} \cup \{\mathsf{inr}\ v_2 \mid v_2 \in \mathcal{T}(E_2)\} \\
\mathcal{T}(E_1^\star) &= \{[v_0, \ldots, v_n] \mid v_i \in \mathcal{T}(E_1)\}\,.
\end{aligned}$$

□

Note that there is a case dealing with the RE type 0, which we have not encountered before. It is also a valid "normal" RE, but not used in practice, as it denotes the empty language and is of limited interest in the traditional scenarios where REs are used. It does matter in the type interpretation, therefore we include it here.

Example 4. Consider the regular expression that denotes words formed from arbitrary combinations of $\underline{\mathtt{a}}$ and $\underline{\mathtt{b}}$:

$$E_{ab} = (\underline{\mathtt{a}} + \underline{\mathtt{b}})^\star\,.$$

If we build an automaton corresponding to E_{ab} as described in Section 3.1, the following words will all cause it to reach the accepting state:

$$\varepsilon \qquad \underline{\mathtt{a}} \qquad \underline{\mathtt{b}} \qquad \underline{\mathtt{ababba}} \qquad \underline{\mathtt{bbaabba}}.$$

By eye-balling the expression, it becomes clear that the language denoted is:

$$\mathcal{L}(E_{ab}) = \{v \mid v \text{ is either empty or consists of combinations of } \underline{\mathtt{a}}\text{s and } \underline{\mathtt{b}}\text{s}\}\,.$$

The type denoted by this expression is written in the same way as the expression itself: we have a *type* $E_{ab} = (\underline{\mathtt{a}} + \underline{\mathtt{b}})^\star$. This type denotes a set of objects of a particular shape. We can deduce how they look:

$$\begin{aligned}
\mathcal{T}(E_{ab}) &= \mathcal{T}((\underline{\mathtt{a}} + \underline{\mathtt{b}})^\star) \\
&= \{[t_0, \ldots, t_n] \mid t_i \in \mathcal{T}(\underline{\mathtt{a}} + \underline{\mathtt{b}})\} \\
&= \{[t_0, \ldots, t_n] \mid t_i \in \{\mathsf{inl}\ v \mid v \in \mathcal{T}(\underline{\mathtt{a}})\} \cup \{\mathsf{inr}\ w \mid w \in \mathcal{T}(\underline{\mathtt{b}})\}\} \\
&= \{[t_0, \ldots, t_n] \mid t_i \in \{\mathsf{inl}\ v \mid v \in \{a\}\} \cup \{\mathsf{inr}\ w \mid w \in \{b\}\}\} \\
&= \{[t_0, \ldots, t_n] \mid t_i \in \{\mathsf{inl}\ a\} \cup \{\mathsf{inr}\ b\}\} \\
&= \{[t_0, \ldots, t_n] \mid t_i \in \{\mathsf{inl}\ a, \mathsf{inr}\ b\}\}\,.
\end{aligned}$$

This is the set of lists formed from arbitrary combinations of the two elements $\mathsf{inl}\ a$ and $\mathsf{inr}\ b$. The two tags inl and inr adds information about the structure; it is encoded that a is the left and b the right choice in the sum. □

These structured values are said to *inhabit* the type; when a value "lives in" a type E, it is an *inhabitant of type* E.

5.1 Handling Lists

At first glance it seems like we have three different ways of combining types to form new types: $+$, $\cdot$, and $[\cdot]$. However, a list is defined to be either an empty list, written $()$, or an element combined with the remainder of the list (the head and the tail of the list). Consequently, the list type can be written:

$$E^\star = EE^\star + 1,$$

that is, either a designated end symbol of type 1 or an element v of type E along with the rest of the list of type $E^\star$. The only inhabitant of type 1 is $()$, and if we assume that v is an inhabitant of type E and l is an inhabitant of the remainder with type $E^\star$, we get an element $\langle e, l\rangle$ of type $EE^\star$. As these two elements are combined using the sum, a list is either the object $\mathsf{inr}\ ()$ or the object $\mathsf{inl}\ \langle v, l\rangle$, where l is again a list.

Example 5. The type $E_a = \underline{\mathtt{a}}^\star$ denotes the set of lists of $\underline{\mathtt{a}}$s. The element of $\mathcal{T}(E_a)$ with three repetitions can be written:

$$[a, a, a] = \langle \mathsf{inl}\ a, \langle \mathsf{inl}\ a, \langle \mathsf{inl}\ a, \mathsf{inr}\ ()\rangle\rangle\rangle.$$

Compare with the way the list `[a,a,a]` can be written in Haskell:

```
[a,a,a] = a:(a:(a:[]))
```

Thus, the Kleene star type is just the normal list type like it is used in Haskell or ML. □

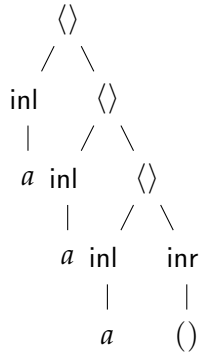
5.2 Values are Parse Trees

If we consider a tree structure in which all nodes are labeled with the constructors introduced in the structured values above and all leaves are labeled with the primitive objects a, b, etc., we can view the structured values as *parse trees*. Anything that does not take any arguments is a leaf, i.e., the primitive objects, and anything that takes $n > 0$ arguments is a node with $n > 0$ children. This is most easily illustrated with an example:

Example 6. The value

$$[a, a, a] = \langle \mathsf{inl}\ a, \langle \mathsf{inl}\ a, \langle \mathsf{inl}\ a, \mathsf{inr}\ () \rangle \rangle \rangle$$

from before can be written as the following parse tree:



□

In the following, we shall use the terms "value", "structured value", "element of type E", "inhabitant of type E", and "parse tree" interchangeably.

5.3 Flattening Trees

We can defined the *flattening* of a parse tree as the function that removes any structural information and leaves behind the underlying string, which is contained in the leaves.

Definition 5. *The* flattening *of a parse tree v is denoted $|\cdot|$ and is defined as:*

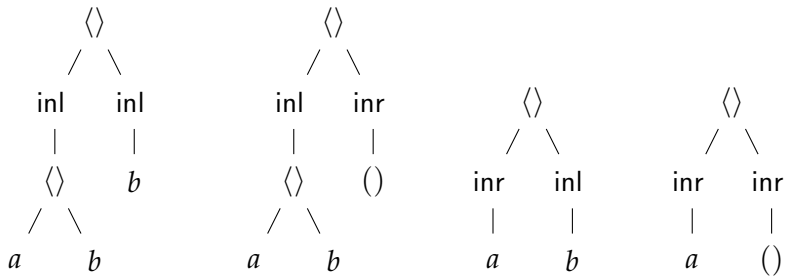
$$|()| = \epsilon, \quad |a| = \underline{a}, \quad |\langle v_1, v_2 \rangle| = |v_1||v_2|, \quad |\mathsf{inl}\ v| = |v|, \quad |\mathsf{inr}\ v| = |v|.$$

□

Example 7. Consider the expression $E = (\underline{\mathtt{ab}} + \underline{\mathtt{a}})(\underline{\mathtt{b}} + 1)$. The language interpretation is $\mathcal{L}(E) = \{\underline{\mathtt{a}}, \underline{\mathtt{ab}}, \underline{\mathtt{abb}}\}$, and the type interpretation is

$$\begin{aligned}
\mathcal{T}(E) &= \mathcal{T}((\underline{\mathtt{ab}} + \underline{\mathtt{a}})(\underline{\mathtt{b}} + 1)) \\
&= \mathcal{T}(\underline{\mathtt{ab}} + \underline{\mathtt{a}}) \times \mathcal{T}(\underline{\mathtt{b}} + 1) \\
&= \{\mathsf{inl}\ \langle a, b \rangle, \mathsf{inr}\ a\} \times \{\mathsf{inl}\ b, \mathsf{inr}\ ()\} \\
&= \{\langle \mathsf{inl}\ \langle a, b \rangle, \mathsf{inl}\ b \rangle, \langle \mathsf{inl}\ ab, \mathsf{inr}\ () \rangle, \langle \mathsf{inr}\ a, \mathsf{inl}\ b \rangle, \langle \mathsf{inr}\ a, \mathsf{inr}\ () \rangle\}\,.
\end{aligned}$$

Drawn as trees, these values look like the following:



Using the flattening function on each of these trees give the strings

$$\underline{\texttt{abb}} \qquad \underline{\texttt{ab}} \qquad \underline{\texttt{ab}} \qquad \underline{\texttt{a}}$$

respectively. □

5.4 Ambiguity in Regular Expressions

Notice in Example 7 that there are two different parse trees that flatten to the same string: $\langle \mathsf{inl}\ \langle a, b\rangle, \mathsf{inr}\ ()\rangle$ and $\langle \mathsf{inr}\ a, \mathsf{inl}\ b\rangle$ both flatten to $\underline{\texttt{ab}}$. This fact is not visible in the language interpretation because the structure of expressions is ignored from the beginning. This is an example of an *ambiguous* RE:

Definition 6. *An RE E is* ambiguous *exactly when two or more parse trees in $\mathcal{T}(E)$ flatten to the same string:*

$$\exists v_1, v_2 \in \mathcal{T}(E).\, v_1 \neq v_2 \wedge |v_1| = |v_2|.$$

□

6 Regular Expression Parsing

We are now in a position to formulate a more fine-grained way of "running" a regular expression on an input string. In Section 3, the task was to produce a yes/no answer to the question

"Does the string s match the expression E?"

This was adjusted in Section **??** to find *substrings* of the input string which match the E. The substring problem is more general, as we always know the answer to the basic question of matching if we know that there are substrings that match. If the problem is reformulated such that we want to find a parse tree for a given RE, we generalize even further: If we have a parse tree we have an answer to both the simple yes/no question and the sub-match question.

Definition 7. *Given an RE E,* to parse a string s under E *is to construct a parse tree v that inhabits $\mathcal{T}(E)$, and whose flattening is s: $|v| = s$.* □

When an RE is ambiguous the parser must use some strategy to pick between the possibilities. This strategy is called a *disambiguation strategy*.

We are now able to handle the problems encountered in the earlier examples from Section 4.

Example 8. Under the type interpretation of E_{numS} the input `1.234.567,89` corresponds to the following value:

$$\begin{aligned}&\langle[\langle\langle 1, []\rangle, \underline{.}\rangle, \langle\langle 2, [3,4]\rangle, \underline{.}\rangle],\\ &\langle\langle\langle 5, [6,7]\rangle, \underline{,}\rangle,\\ &\langle 8, [9]\rangle\rangle\rangle\end{aligned}$$

Here, 1 is abbreviation of the actual value of the digit `1` under the type interpretation of E_{digit}, which is $\mathsf{inr}\ \mathsf{inl}\ 1$. The entire structure of the original RE is preserved in the value, so the problems discussed on page 11 do not arise. Because we do not lose any structural information, we can write a program of the type $E_{numS} \longrightarrow E_{numE}$ that converts values using the Scandinavian-style thousands-separator to values using the English-style separators. □

Example 9. Consider the input `(a:1)(b:NULL)(c:2)` and the expression E_{pairs} from page 11. Recall that with the naïve submatch extraction we could not recreate a full dictionary from this string. The corresponding value of this string under the type interpretation is

$$\begin{aligned}&[\langle\langle a, \underline{,}\rangle, \mathsf{inl}\ 1\rangle,\\ &\langle\langle b, \underline{,}\rangle, \mathsf{inr}\ \mathit{NULL}\rangle,\\ &\langle\langle c, \underline{,}\rangle, \mathsf{inl}\ 2\rangle]\end{aligned}$$

where NULL is shorthand for the actual value associated with `NULL`. Clearly, this piece of data contains enough information to build a dictionary. □

6.1 Bit-Coding

There is a compact way of representing each value, by "splitting" the information contents between that which is universal to all values $\mathcal{T}(E)$, and that which is specific to a particular value $v \in \mathcal{T}(E)$. Take the value from before:

$$\langle \mathsf{inl}\ a, \langle \mathsf{inl}\ a, \langle \mathsf{inl}\ a, \mathsf{inr}\ ()\rangle\rangle\rangle,$$

and rewrite it by replacing occurrences of $\mathsf{inl}\ X$ and $\mathsf{inr}\ X$ with 0 and 1:e

$$\langle 0, \langle 0, \langle 0, 1\rangle\rangle\rangle.$$

We may do this because, given a subexpression $E_1 + E_2$, the fact that the left choice or the right choice was taken *uniquely* specifies the information contained in the particular usage of $+$ in a value v. Whatever information is required for a particular v in E_1 or E_2 must then be added, but this will no longer be information about that particular $+$. Specifically, for the expression $\underline{\mathtt{a}} + \underline{\mathtt{b}}$, there is exactly one possibility on either side of the $+$, hence there is no information content, and we may discard it. One can think of this as the fact that $\underline{\mathtt{a}}$ is on the left-hand side already is recorded in the expression itself, and there is therefore no reason to repeat it in the value.

Now, if we consider the sequential composition of REs, we notice that there is *no* information contents in the sequence – any value inhabiting the type *must* consist of the first part followed by the second part. Hence, the constructor $\langle\rangle$ is not really needed, as it does not encode any information specific to any particular value, leading us to reduce further:

$$0001.$$

This is just a bit-string! These four bits encode, in a quite compact manner, the entire value, *provided the RE is known*. All information contents specific to the value itself is represented in the bit-string, and any information that is general to every value inhabiting the type is left in the type itself.

Example 10. To illustrate why both the bit-string and the RE is needed, consider the bit-string

$$0001$$

again. Given RE $E_1 = (\underline{\mathtt{b}} + \underline{\mathtt{a}})(\underline{\mathtt{a}} + \underline{\mathtt{b}})(\underline{\mathtt{b}} + \underline{\mathtt{a}})(\underline{\mathtt{a}} + \underline{\mathtt{b}})$ this bit-string is the value

$$\langle \mathsf{inl}\ b, \langle \mathsf{inl}\ a, \langle \mathsf{inl}\ b, \mathsf{inr}\ b \rangle \rangle \rangle,$$

but given the RE $E_2 = a^\star$ the value is

$$\langle \mathsf{inl}\ a, \langle \mathsf{inl}\ a, \langle \mathsf{inl}\ a, \mathsf{inr}\ () \rangle \rangle \rangle.$$

□

We call the bit-string that encodes a parse tree a *bit-code*. To solve the parsing problem it suffices to construct a bit-code for a parse tree, as this can be converted to a "proper" data structure uniquely, given the RE type.

7 Coercions and Transformations

Because all information about the structure of the regular expression is retained in a structured value, we can perform transformations on the pieces

of data that can be reflected in transformations of the regular expressions themselves. In this section we will discuss two ways of transforming data: one that can change the underlying text string called *transformations*, and one that always preserves it, called *coercions*.

7.1 Transformations

Recall that in the example with thousand separators in Section 4 we wanted to change between two different styles of separators easily, but there was no easy way of specifying this change. When we have a full parse tree, this transformation can be easily formulated as a function between the two types designating the Scandinavian and English style, respectively.

To construct the transformation, we will at first need a way to transform a dot to a comma and vice versa. We therefore define the functions `fromComma` and `fromDot` with the types $\underline{,} \to \underline{.}$ and $\underline{.} \to \underline{,}$, respectively. Note that because we view the regular expressions as types, we can construct types that designate functions between regular expressions. These particular regular expressions are trivial because the each designate a singleton-set:

$$\mathcal{T}(\underline{,}) = \{,\}$$
$$\mathcal{T}(\underline{.}) = \{.\},$$

so the corresponding implementations of the functions should be simple too:

```
fromComma :: Comma -> Dot -- , → .
fromComma Comma = Dot
fromDot :: Dot -> Comma -- . → ,
fromDot Dot = Comma
```

The pseudo-code syntax we will use resembles that of Haskell. The above code snippet defines two functions, each of them taking a value of type $\underline{,}$ or $\underline{.}$ and producing a value of the other type. Because the types are both singletons the pattern matches are exhaustive.

7.2 Functorial Program Composition

These programs work as long as the comma/dot is the *only* symbol present. Clearly, this is not enough for a more complicated example, so there must be a way to compose programs. Because we are working with regular expression types, we need concatenation, sum, and Kleene star. These three constructs in combination with the base programs of the form described above will allow us to concisely express transformations on structured values.

Concatenation Given two programs p_1 and p_2 with the types $E_1 \to E_2$ and $F_1 \to F_2$, we can form a program $p = p_1 \times p_2$ with the type $E_1 \times F_1 \to E_2 \times F_2$. A straight-forward implementation in Haskell could look like the following:

```
p1 :: E1 -> E2 -- E1 → E2
p2 :: F1 -> F2 -- F1 → F2
p :: (E1, F1) -> (E2, F2) -- E1 × F1 → E2 × F2
p (v1, v2) = (p1 v1, p2 v2)
```

Note that the type $E_1 \times F_1$ and the data $\langle v_1, v_2 \rangle$ are written `(E1, F1)` and `(v1, v2)` in Haskell.

Sum Likewise, if programs p_1 and p_2 are as above, we can form their sum $p = p_1 + p_2$ which will have the type $E_1 + F_1 \to E_2 + F_2$. In Haskell this looks like:

```
p :: Either E1 F1 -> Either E2 F2 -- E1 + F1 → E2 + F2
p (Left v)  = Left (p1 v)
p (Right v) = Right (p2 v)
```

Again there is a small discrepancy between Haskell-style syntax and that used elsewhere, which should be self-explanatory.

Kleene Star The construct for Kleene stars will also be familiar to functional programmers. Given a program p_0 with the type $E_1 \to E_2$, we must construct a program $p = p_0^\star$ with the type $E_1^\star \to E_2^\star$. When you recall that the Kleene star is simply a list data type under the type interpretation, you easily see that this function is just an application of `map`!

```
p :: [E1] -> [E2] -- E1* → E2*
p = map p0
```

Constants To handle the constant transformation we use the identity function, and the failing transformation corresponds to converting something to the regular expression 0. As it does not have any elements, such a program will always fail:

```
id :: a -> a
id x = x
fail :: a -> b -- E → F
fail = undefined -- always fails
```

We are now able to formulate the transformation between Scandinavian and English style numbers from earlier examples.

Example 11. To transform numbers conforming to the format E_{numS} into the format of E_{numE} we construct the following program:

$$(\texttt{id} \times \texttt{fromDot})^\star \times \texttt{id} \times \texttt{fromComma} \times \texttt{id}$$

which has the type

$$\begin{aligned}&((E_{digit}E^\star_{digit})\ \underline{.})^\star(E_{digit}E^\star_{digit})\ \underline{,}\ (E_{digit}E^\star_{digit})\\ \to\ &((E_{digit}E^\star_{digit})\ \underline{,})^\star(E_{digit}E^\star_{digit})\ \underline{.}\ (E_{digit}E^\star_{digit})\end{aligned}$$

(The $\times$-symbol is omitted for brevity.) The function `id` is polymorphic and has the type $E_{digit} \times E^\star_{digit} \to E_{digit} \times E^\star_{digit}$ in this example. □

7.3 Coercions

The transformation presented in the example above does not preserve the underlying string. Another type of action which we call *coercions* do preserve strings, but they still transform the data from one type to another. For instance, values of the regular expression $\underline{\texttt{a}} + \underline{\texttt{b}}$ may be coerced into values of the regular expression $\underline{\texttt{b}} + \underline{\texttt{a}}$ and vice versa. In the language interpretation, this would have been a trivial statement, as the sum operator is commutative. Hence, there the two expressions are essentially the same. With the type interpretation this is different. Because we are interested in the *structure* and not just the set of words permitted by a finite automaton, changing the structure by swapping the order of the summands is not a trivial operation anymore.

One can think of the existence of a program converting values of one regular expression to another as a *proof* of the fact that one expression is "less than" another. This constructive interpretation of proofs is a very powerful notion, and is a complex field of study in itself. Four our purposes, the proof system is limited to only expressing inequalities between regular expressions. In the language interpretation, the fact that expression E is "less than" expression F means that the language of F is a superset of that of E:

$$E \leq F \iff \mathcal{L}(E) \subseteq \mathcal{L}(F).$$

It is a bi-implication because Kleene algebra is complete for the regular lanugages [12]. Consequently, if the underlying string of a value is not changed, it should be possible to transform the value from one of type E into one of type F. This is called a *coercion*, as it "forces" the parse tree into a different shape while still keeping the leaves intact.

As a first example, consider the expressions $\underline{\texttt{a}} + \underline{\texttt{b}}$ and $\underline{\texttt{b}} + \underline{\texttt{a}}$. They both represent the language $\{\underline{\texttt{a}}, \underline{\texttt{b}}\}$, so we expect

$$\underline{\texttt{a}} + \underline{\texttt{b}} \leq \underline{\texttt{b}} + \underline{\texttt{a}} \quad \text{and} \quad \underline{\texttt{b}} + \underline{\texttt{a}} \leq \underline{\texttt{a}} + \underline{\texttt{b}}$$

to hold, i.e., $\underline{a} + \underline{b} = \underline{b} + \underline{a}$. When thinking only in terms of the underlying languages, this is clearly the case. The added structure of our parse trees, however, highlights that this equivalence carries *computational contents*, that "something is done". In this example, the computational action is that which replaces `inl` tags with `inr` tags. We can write this as a Haskell-program:

```
retag :: Either a b -> Either b a
retag (Left v)  = Right v
retag (Right v) = Left v
```

This program should be thought of as a *proof* that the regular expression $\underline{a} + \underline{b}$ is contained in $\underline{b} + \underline{a}$. The program is polymorphic and it never inspects its input apart from the tag, so it states a much more general property:

$$E + F \leq F + E.$$

Instantiating E and F with $\underline{a}$ and $\underline{b}$ yields a proof of the inequality going one way, and switching the instantiation yields the opposite one. Hence, this program proves that

$$E + F = F + E,$$

and it does so by specifying *how* values with one type should be transformed into values with another type without altering the underlying strings.

Another instructive example is the equivalence of the expressions $\underline{a} \times (\underline{b} \times \underline{c})$ and $(\underline{a} \times \underline{b}) \times \underline{c}$. Only the structure and not the ordering of symbols is changed, so we can write a coercion that reorders the tuples inhabiting the two product types:

```
assoc :: (a, (b, c)) -> ((a, b), c)
assoc (x, (y, z)) = ((x, y), z)
assocInv :: ((a, b), c) -> (a, (b, c))
assocInv ((x, y), z) = (x, (y, z))
```

In this example there are two programs, one in each direction, witnessing the regular expression inequalities

$$E \times (F \times G) \leq (E \times F) \times G \quad \text{and}$$
$$(E \times F) \times G \leq E \times (F \times G),$$

respectively. Hence, combining the two represents a proof of

$$E \times (F \times G) = (E \times F) \times G.$$

$$E + (F + G) = (E + F) + G \tag{1}$$
$$E + F = F + E \tag{2}$$
$$E + 0 = E \tag{3}$$
$$E + E = E \tag{4}$$
$$E \times (F \times G) = (E \times F) \times G \tag{5}$$
$$1 \times E = E \tag{6}$$
$$E \times 1 = E \tag{7}$$
$$E \times (F + G) = (E \times F) + (E \times G) \tag{8}$$
$$(E + F) \times G = (E \times G) + (F \times G) \tag{9}$$
$$0 \times E = 0 \tag{10}$$
$$E \times 0 = 0 \tag{11}$$

Figure 1: Axioms for idempotent semirings.

7.4 Kleene Algebra

The algebra of regular expressions is called *Kleene algebra* (KA), and the small examples we have studied above are part of the axiomatization of this algebra. Kozen presented an axiomatization that is sound and complete for the regular sets of strings, i.e., an inclusion between any two regular sets of strings can be shown as an inequality in KA with these axioms, and any inequality in KA shown using these axioms represents an inclusion between regular sets of strings [12].

A Kleene algebra is an structure containing an underlying set, the *carrier set*, three operations, $+$, $\times$, and $\star$, and two special symbols 0 and 1. Hence, terms in KA are just regular expressions and the carrier set the alphabet. The structure satisfies all the axioms of *idempotent semirings*, which express the behaviour of the $+$ and $\times$ operations and how they interact. Furthermore, four extra axioms for the Kleene star are added to make it a Kleene algebra. Figures 1 and 2 show the axioms for idempotent semirings and the added axioms governing the Kleene stars.

The two examples from before correspond to axioms 2 and 5. Each axiom in Figure 1 can be given a direct computational interpretation – a set of two programs that transform data from left to right and vice versa [9]. By using a coinductive rule that allows "loops" in proofs, the Kleene star can be formulated as a fix point computation. The idea is, that if, under the assumption that some function f coerces E into F, it is possible to construct another function c that also coerces E into F, we may use the constructor

$$1 + E \times E^\star \leq E^\star \tag{12}$$
$$1 + E^\star \times E \leq E^\star \tag{13}$$
$$E + F \times X \leq X \to F^\star \times E \leq X \tag{14}$$
$$E + X \times F \leq X \to E \times F^\star \leq X \tag{15}$$

Figure 2: Axioms governing Kleene star.

`fix` to create the function `fix f.c` with the type $E^\star \to F^\star$. Obviously, if no restrictions are enforced we may trivially pick c to be the f that is assumed – this is clearly unsound, so a side-condition on the structure of f must be enforced [9].

Figure 3 shows the primitives in the inference system. Γ is an environment of hypotheses that is used with the `fix` rule:

$$\frac{\Gamma, f : E \leq F \vdash c : E \leq F}{\Gamma \vdash \text{fix } f.c : E \leq F}$$

Subproofs can be combined to form larger proofs using the method of functorial composition described above – remember that the proofs are just programs. Let us reformulate the composition rules in the style of inference rules to underline the fact that we are thinking of it as a *proof system*:

$$\frac{\Gamma \vdash c : E \leq E' \quad \Gamma \vdash d : F \leq F'}{\Gamma \vdash c + d : E + F \leq E' + F'} \qquad \frac{\Gamma \vdash c : E \leq E' \quad \Gamma \vdash d : F \leq F'}{\Gamma \vdash c \times d : E \times F \leq E' \times F'}$$

A proof usually consists of a sequence of steps performed in succession, so we need a way to combine subproofs sequentially as well:

$$\frac{\Gamma \vdash c : E \leq E' \quad \Gamma \vdash d : E' \leq E''}{\Gamma \vdash c; d : E \leq E''}$$

Each of the primitive functions in Figure 3 has a computational interpretation. Two of these we have already seen in the form of Haskell code above. Most of the rest are similarly simple functions. Of special note is `wrap`; compare the type signatures of the standard `foldr` function and the signature of `wrap`:

```
foldr :: (a -> b -> b) -> b -> [a] -> b
```

The function takes three elements as input:

- a function that inserts a new element of type `a` into the composite structure of type `b`,

$$
\begin{array}{lll}
\Gamma \vdash \text{shuffle} & : & E + (F + G) = (E + F) + G \\
\Gamma \vdash \text{retag} & : & E + F = F + E \\
\Gamma \vdash \text{untagL} & : & 0 + F = F \\
\Gamma \vdash \text{untag} & : & E + E \leq E \\
\Gamma \vdash \text{tagL} & : & E \leq E + F \\
\Gamma \vdash \text{assoc} & : & E \times (F \times G) = (E \times F) \times G \\
\Gamma \vdash \text{swap} & : & E \times 1 = 1 \times E \\
\Gamma \vdash \text{proj} & : & 1 \times E = E \\
\Gamma \vdash \text{abortR} & : & E \times 0 = 0 \\
\Gamma \vdash \text{abortL} & : & 0 \times E = 0 \\
\Gamma \vdash \text{distL} & : & E \times (F + G) = (E \times F) + (E \times G) \\
\Gamma \vdash \text{distR} & : & (E + F) \times G = (E \times G) + (F \times G) \\
\Gamma \vdash \text{wrap} & : & 1 + E \times E^\star = E^\star \\
\Gamma \vdash \text{id} & : & E = E
\end{array}
$$

Figure 3: Functions implementing Kleene algebra axioms.

- a start composite element of type `b`,
- a sequence of elements to be added, of type `[a]`.

This type signature is the same as the axiom 14 read as a type!

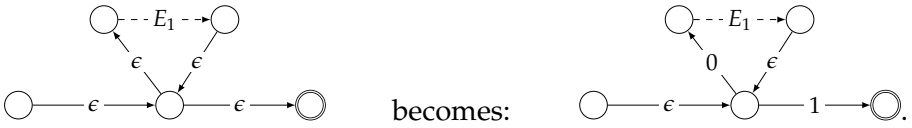
8 Two-Phase Regular Expression Parsing

In this Section we will introduce the principles behind the two-phase RE parsing algorithm from [7].

8.1 Bit-Codes as Oracles

Imagine that you have to guide a small pebble through an NFA by placing it on the states and moving it along the transition edges. The information contained in the bit-code can serve as an *oracle*, which can be queried each time there are two possible moves for the pebble. The pebble moves left or right depending on whether the "current" bit is 0 or 1. For the situations where there is only one outgoing edge, the pebble will simply move to the state pointed at regardless of the label. The idea that 0 and 1 control which edge should be picked can be encoded in the NFA itself. Occurences of the sum construction:

In the Kleene star construction there is also a notion of a choice; the pebble must either do another iteration in the loop or exit it. This is representable in the same way as above, and instances of the construction for $E_1^\star$:



Notice that these two cases are the only two in which a choice is introduced. This corresponds nicely to the previous observation that only the information concerning which side of a sum is "used" needs to be stored for a value; all other information is implied by the type itself.

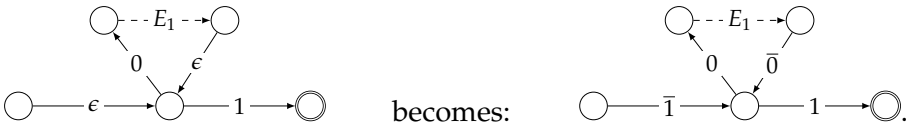
8.2 Handling Ambiguous REs

Knowing the labels on outgoing edges does not help when parsing a string, as it is impossible to know *which*, if any, of the two possible paths will succeed. As described in Section 3, this can be remedied by using state sets. Note that a successful parse must return exactly one bit-coded parse tree, hence a disambiguation strategy must be used at some point in the parsing process.

Observe that whenever a state has two *incoming* edges there is *exactly* one corresponding state with two *outgoing* edges. This can easily be verified by inspecting the cases in the NFA construction algorithm. Only two cases introduce states with two ingoing edges, namely the cases for $E_1 + E_2$ and $E_1^\star$, and the one-to-one correspondence is maintained in both. The remaining cases trivially preserves the symmetry. Note that in the Kleene star case, one state has both two incoming and two outgoing edges; this preserves the one-to-one correspondence.

The symmetrical structure of the NFAs can be included in the construction algorithm, along with the bit labels described previously. We use the symbols $\overline{0}$ and $\overline{1}$ as the duals to 0 and 1. Sum:

Likewise, Kleene star:



We have now removed all ϵ-edges from the NFA by labelling them with bits. This new kind of NFA is called an *augmented NFA* (aNFA). Its edges can be divided into three kinds:

- *transition edges* are labelled with letters from the alphabet;
- *logging edges* are labeled with $\overline{0}$ or $\overline{1}$;
- *choice edges* are labeled with 0 or 1.

If a state has two outgoing edges they will always be choice edges, and the state is called a *split state*. Conversely, states with two incoming edges are called *join states*, and the edges will always be logging edges.

8.2.1 Greedy Left-Most Disambiguation Strategy

The two-phase algorithm uses the *greedy left-most* disambiguation strategy. This strategy causes the algorithm to produce the same parse tree that a naïve recursive descent parser would produce; the left choice is always favored over the right choice, and if the left choice fails, go back in the RE structure and try the right alternative.[2]

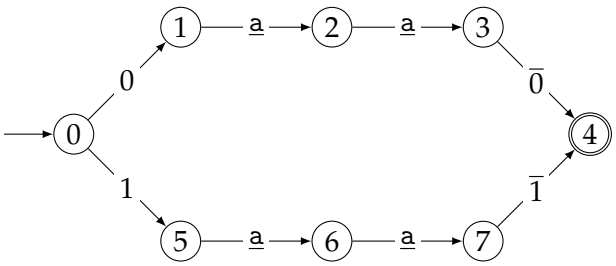
Clearly, when two parallel paths through the aNFA are followed, only if both paths succeed will they both reach the join state. But in that case we know which path was the leftmost, as it is the one entering through the $\overline{0}$-edge. Hence, this path is saved and the other one discarded, and the ambiguity is resolved.

[2]This type of parser is called a *backtracking parser*, and it is the way Perl implements regular expression matching (Perl does not perform full RE parsing). Deceptively simple, they suffer from an exponential blow-up in running time on inputs that causes the algorithm to backtrack a lot.

Example 12 (Unfinished algorithm)**.** The expression $E_{aa} = \underline{\mathtt{aa}} + \underline{\mathtt{aa}}$ is ambiguous, as there are two different parse trees that both flatten to the same string:

$$|\mathsf{inl}\ \langle a, a\rangle| = |\mathsf{inr}\ \langle a, a\rangle| = \underline{\mathtt{aa}}.$$

The aNFA for E_{aa} is (the nodes are labeled for easy reference):



Simulating the aNFA is very similar to how a normal NFA is simulated. Initially, the active state set is $\{0\}$. On the input $\underline{\mathtt{a}}$, the aNFA can move along both $\underline{\mathtt{a}}$-labeled edges, $1 \to 2$ and $5 \to 6$, making the new active state set $\{2, 6\}$. After another $\underline{\mathtt{a}}$, the transitions $2 \to 3$ and $6 \to 7$ are taken, and as $3 \to 4$ and $7 \to 4$ both are logging edges, the next (and final) active state set will be $\{4\}$. Both the $\overline{0}$ and the $\overline{1}$ edge have been taken, but due to the greedy left-most strategy, only the path that took $\overline{0}$ is stored. This path corresponds to the left choice, which has the bit-code 0. □

8.3 Logging

The procedure sketched so far implies that *all paths* taken throughout the simulation of the automaton be stored until it can be decided which one of them is the desired one. By exploiting the symmetries in the aNFA, we can do something much more space-efficient than simply storing the paths as we go along.

Each time an edge into a join-state is traversed, and a choice between left or right therefore is made, we always pick the left option if present. Due to the one-to-one correspondence between join and split states, disambiguating a join state only disambiguates one split state. It is therefore enough to store, for each join state, one bit indicating whether the left or right edge has been traversed to reach it. But this is exactly the dual bits $\overline{0}$ and $\overline{1}$ from the two incoming edges! Consequently, when a join state is reached, the algorithm must store the bit on the incoming edge followed into the join state, with preference for the $\overline{0}$-bit. This must be done for all join states in the aNFA, and we see that $n \cdot c$ bits must be stored, where n is the length of the input string and c the number of join states.

Storing these "log bits" does not constitute a parse tree. Once the input has been processed, a *backwards pass* is performed through the aNFA from

the accepting state to the initial state. The stored bits are used here as an oracle that guides through the automaton. There is no need to keep the input string, as the logged bits encode all the information specific to the particular run of the automaton. Because the aNFA is traversed backwards, the log bits are needed in reverse order of how they were stored – we use a stack for this.

8.4 State Lists

There is one more change to the NFA-simulation algorithm we have to introduce before the algorithm is complete. In the normal algorithm, the active states are stored in *sets*, and hence the ordering between them does not matter – the exact way in which a string is accepted by an automaton is irrelevant to the fact that it is accepted. In the parsing algorithm we are interested in one particular path through the aNFA, and therefore the order in which the states are reached matters. Therefore, the algorithm uses *lists* instead of sets to store the active states. The leftmost alternative is favored in choices, but the next state list is ordered based on the order the states are reached from the current state list.

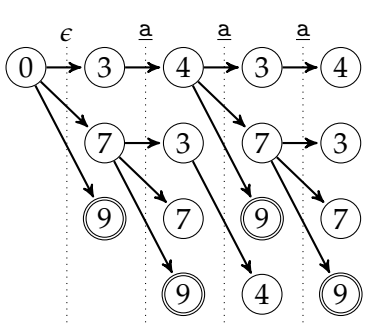
Example 13. The expression

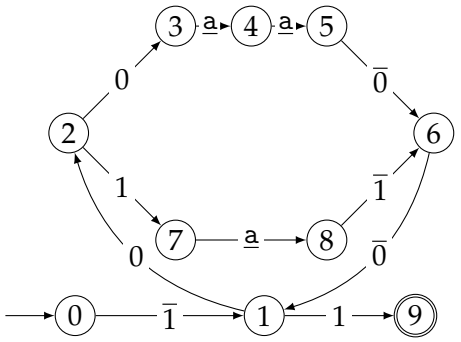
$$E_{ambig} = (\underline{\mathtt{aa}} + \underline{\mathtt{a}})^\star$$

is ambiguous. The input string $\underline{\mathtt{aaa}}$ may be parsed (at least) as either

$$\begin{aligned}
&\langle \mathsf{inl}\ \mathsf{inl}\ \langle a, a\rangle, \langle \mathsf{inl}\ \mathsf{inr}\ a, \mathsf{inr}\ ()\rangle\rangle && (00011),\\
&\langle \mathsf{inl}\ \mathsf{inr}\ a, \langle \mathsf{inl}\ \mathsf{inl}\ \langle a, a\rangle, \mathsf{inr}\ ()\rangle\rangle && (01001), \text{ or}\\
&\langle \mathsf{inl}\ \mathsf{inr}\ a, \langle \mathsf{inl}\ \mathsf{inr}\ a, \langle \mathsf{inl}\ \mathsf{inr}\ a, \mathsf{inr}\ ()\rangle\rangle\rangle && (0101011).
\end{aligned}$$

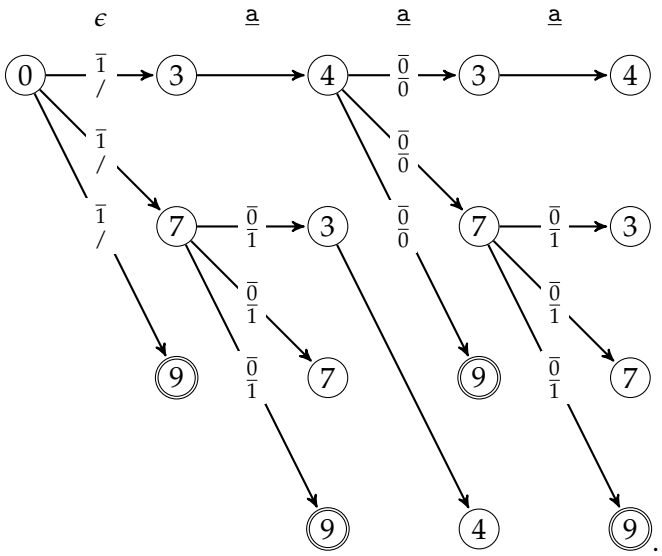
The correct one according to the greedy leftmost strategy is the first one. If we simulate the aNFA for E_{ambig}, the active state lists change as pictured below. The lists are ordered such that the topmost element at each instance is the highest prioritized (i.e., lefmost) state.





The final states are $[4, 3, 7, 9]$. This includes the accepting state, so the string matches. This representation corresponds to storing all the paths. We can reconstruct the parse tree by tracing backwards from the accepting state and emit all the bits on the traversed edges: $9 \to 7 \to 4 \to 3 \to 0$ yields 11000, the (reversed) correct bitcode.

States 1 and 6 are the only join states in the automaton. First, write down the log bit information associated with each transition between states, with the bit for state 1 above the bit for state 6:



We thus have enough information in the following bits to reconstruct the correct parse tree:

	ϵ	a	a	a
1	$\overline{1}$	$\overline{0}$	$\overline{0}$	$\overline{0}$
6	/	$\overline{1}$	$\overline{0}$	$\overline{1}$

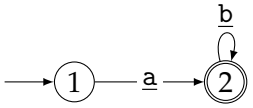
If we use these log bits as an oracle, we can trace deterministically backwards through the aNFA. Every time a symbol transition is traversed, the stack of log frames is popped (indicated with subscript p):

$$\underbrace{9 \to 1 \to_{\overline{0}} 6 \to_{\overline{1}} 8 \to_p}_{\frac{\overline{0}}{\overline{1}}} \underbrace{7 \to 2 \to 1 \to_{\overline{0}} 6 \to_{\overline{0}} 5 \to_p}_{\frac{\overline{0}}{\overline{0}}} \underbrace{4 \to_p}_{\frac{\overline{0}}{\overline{1}}} \underbrace{3 \to 2 \to 1 \to_{\overline{1}} 0}_{\frac{\overline{1}}{/}}.$$

The active log frames are written below the path, and the arrow subscripts indicate how choices have been determinized. This path yields the correct bitcode. □

9 Derivatives of Regular Expressions

One can think of the fact than an automaton is in an accepting state as if that there is an "end transition" out of that state, followed only in the situation when there is nothing left in the input. In a general automaton, i.e., not a Thompson NFA/aNFA, there might be other transitions from the final state(s) than this hypothetical end transition which corresponds to allowed input characters that are matched at that point in the automaton. For example, consider the following automaton for the regular expression $\underline{\texttt{ab}}^\star$:



We can think of the states of this automaton as representing languages, i.e., the sequences of characters that eventually lead to an accepting state. State 1 corresponds to the language

$$\mathcal{L}(\underline{\texttt{ab}}^\star) = \{\underline{\texttt{a}}, \underline{\texttt{ab}}, \underline{\texttt{abb}}, \cdots\},$$

as no input symbols have been read yet, and state 2 corresponds to what remains after an $\underline{\texttt{a}}$, which is the language

$$\mathcal{L}(\underline{\texttt{b}}^\star) = \{\epsilon, \underline{\texttt{b}}, \underline{\texttt{bb}}, \cdots\} = \{s \mid \underline{\texttt{a}}s \in \mathcal{L}(\underline{\texttt{ab}}^\star)\}.$$

Note that the language for state 2 contains the empty word ϵ: the empty sequence of characters leads to an accepting state, which is the state 2 itself. Two points are relevant in this example:

- there is a notion of *derivation* between the languages of states 1 and 2,
- a state is accepting if its language contains ϵ.

We can easily define a function ε on regular expressions that converts any regular expression E into either the expression 1 or the expression 0, depending on whether ϵ is contained in the language denoted by E.

Definition 8 (Empty word function)**.** *The* empty word function ε *is defined as:*

$$\begin{array}{ccc} \varepsilon(1) = 1 & \varepsilon(0) = 0 & \varepsilon(\underline{a}) = 0 \\ \varepsilon(E_1E_2) = \varepsilon(E_1)\varepsilon(E_2) & \varepsilon(E_1 + E_2) = \varepsilon(E_1) + \varepsilon(E_2) & \varepsilon(E_1^\star) = 1. \end{array}$$

Or, equivalently:

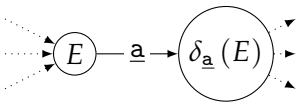
$$\varepsilon(E) = \begin{cases} 1 & \text{if } \epsilon \in \mathcal{L}(E) \\ 0 & \text{otherwise.} \end{cases}$$

□

The way of thinking in the example above can be formulated in a detailed manner, by inspecting regular expressions. The *derivative* $\delta_{\underline{a}}(E)$ of a regular expression E with respect to a symbol $\underline{a}$ is the expression whose language is all strings in Es language that start with an $\underline{a}$, with that $\underline{a}$ removed:

$$\mathcal{L}(\delta_{\underline{a}}(E)) = \{s \mid \underline{a}s \in \mathcal{L}(E)\}.$$

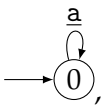
We can think of the relationship between an expression E and its derivative as a part of an automaton, where each state represents a particular regular expression:



When E is 0 its language is empty, and hence the derivative's language is also empty, regardless of which symbol the derivate is relative to:

$$\delta_{\underline{a}}(0) = 0.$$

Drawn as an automaton, this is:

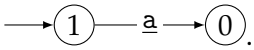


i.e., an automaton where there is no way of reaching the accepting state.

For $E = 1$, the language contains only the string ϵ. Hence, there are no strings that start with any symbol $\underline{a}$, and the derivative is empty:

$$\delta_{\underline{a}}(1) = 0.$$

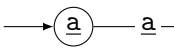
There is therefore also no way of reaching an accepting state in the automaton for the derivative expression:



A slightly more interesting case is when $E = \underline{a}$. Now there is a string that starts with $\underline{a}$ in the language, namely $\underline{a}$ itself! The derived language therefore contains only the empty word, and the expression is:

$$\delta_{\underline{a}}(\underline{a}) = 1.$$

Note that the language $\mathcal{L}(1)$ contains ϵ (in fact it *only* contains ϵ), so thought of as an automaton state it accepts only the empty string. But this means that there is only an empty path to traverse in order to reach an accepting state, and hence it must itself be an accepting state:



However, if instead the expression is $E = \underline{\mathtt{b}}$, where $\underline{\mathtt{b}} \neq \underline{\mathtt{a}}$, there is no strings starting with $\underline{\mathtt{a}}$, and the derived expression becomes:

$$\delta_{\underline{\mathtt{a}}}(\underline{\mathtt{b}}) = 0.$$

In a sum expression, the language is just the union of the two languges of the summands. Consequently, the derivative of a sum is just the sum of the derivatives of its summands:

$$\delta_{\underline{\mathtt{a}}}(E_1 + E_2) = \delta_{\underline{\mathtt{a}}}(E_1) + \delta_{\underline{\mathtt{a}}}(E_2).$$

In the case where $E = E_1 E_2$, the result depends on how E_1 behaves. We want to construct an expression that denotes the language that consists only of substrings from $\mathcal{L}(E)$ that come after an $\underline{\mathtt{a}}$. Recall how the language of a concatenation is defined: It is the Cartesian product of the strings in the language of the two subexpressions. Therefore, if $\mathcal{L}(E_1)$ contains the empty string, there can be a string in $\mathcal{L}(E_1 E_2)$ starting with $\underline{\mathtt{a}}$ whose "E_1-part" is just ϵ. The derivative is therefore dependent on the empty word function:

$$\delta_{\underline{\mathtt{a}}}(E_1 E_2) = \delta_{\underline{\mathtt{a}}}(E_1)\, E_2 + \varepsilon(E_1)\delta_{\underline{\mathtt{a}}}(E_2).$$

The algebraic rules of regular expressions contains the two axioms:

$$1E = E1 = E$$
$$0E = E0 = 0,$$

so the above expression is the same as:

$$\begin{aligned}\delta_{\underline{\mathtt{a}}}(E_1)\, E_2 + \varepsilon(E_1)\delta_{\underline{\mathtt{a}}}(E_2) &= \begin{cases}\delta_{\underline{\mathtt{a}}}(E_1)\, E_2 + 0\delta_{\underline{\mathtt{a}}}(E_2) & \text{if } \epsilon \in \mathcal{L}(E_1)\\ \delta_{\underline{\mathtt{a}}}(E_1)\, E_2 + 1\delta_{\underline{\mathtt{a}}}(E_2) & \text{otherwise}\end{cases}\\ &= \begin{cases}\delta_{\underline{\mathtt{a}}}(E_1)\, E_2 & \text{if } \epsilon \in \mathcal{L}(E_1)\\ \delta_{\underline{\mathtt{a}}}(E_1)\, E_2 + \delta_{\underline{\mathtt{a}}}(E_2) & \text{otherwise.}\end{cases}\end{aligned}$$

We easily see that the intuition about the "E_1-part" dependent on whether $\mathcal{L}(E_1)$ contains ϵ is reflected.

If the expression is $E = E_1^\star$, the derivative represents the strings that consists of repetitions of strings from $\mathcal{L}(E_1)$ with $\underline{\mathtt{a}}$ removed. This is the same as the strings from $\mathcal{L}(E_1)$ with $\underline{\mathtt{a}}$ removed followed by repetitions of normal strings from $\mathcal{L}(E_1)$. The above discussion regarding the empty word is relevant here, too. Luckily, the derivative can be formulated in terms of concatenation, ensuring that the case when $\epsilon \in \mathcal{L}(E_1)$ is handled properly:

$$\delta_{\underline{\mathtt{a}}}(E_1^\star) = \delta_{\underline{\mathtt{a}}}(E_1)\, E_1^\star.$$

The derivative defined here is called the *Brzozowski-derivative*, after the author of [3]:

Definition 9 (Brzozowski-derivative)**.** *The* (Brzozowski-)derivative with respect to $\underline{a}$, $\delta_{\underline{a}}(\cdot)$, *is defined as:*

$$\begin{aligned}
\delta_{\underline{a}}(1) &= 0 \\
\delta_{\underline{a}}(0) &= 0 \\
\delta_{\underline{a}}(\underline{a}) &= 1 \\
\delta_{\underline{a}}(\underline{b}) &= 0 \text{ (if } \underline{a} \neq \underline{b}\text{)} \\
\delta_{\underline{a}}(E_1 + E_2) &= \delta_{\underline{a}}(E_1) + \delta_{\underline{a}}(E_2) \\
\delta_{\underline{a}}(E_1 E_2) &= \delta_{\underline{a}}(E_1)\, E_2 + \varepsilon(E_1)\delta_{\underline{a}}(E_2) \\
&= \begin{cases} \delta_{\underline{a}}(E_1)\, E_2 + \delta_{\underline{a}}(E_2) & \text{if } \epsilon \in \mathcal{L}(E_1) \\ \delta_{\underline{a}}(E_1)\, E_2 & \text{otherwise} \end{cases} \\
\delta_{\underline{a}}(E_1^\star) &= \delta_{\underline{a}}(E_1)\, E_1^\star.
\end{aligned}$$

□

10 Disambiguation Strategies

We have already seen one disambiguation strategy, the greedy leftmost one. In this Section, we briefly revisit disambiguation strategies.

10.1 Greedy

For the sake of completeness we repeat here the *greedy* strategy (or *greedy leftmost* strategy). This strategy always "goes left" in the NFA, and backtracks back to the most recent point where it can go right instead when a path fails. This is exactly the strategy that a simple recursive descent parser follows.

Example 14. The language of expression

$$E_{amb} = (\underline{\mathtt{a}} + \underline{\mathtt{ab}})(\underline{\mathtt{b}} + 1)$$

is

$$\mathcal{L}(E_{amb}) = \{\underline{\mathtt{a}}, \underline{\mathtt{ab}}, \underline{\mathtt{abb}}\}.$$

As the expression is ambiguous on the input $\underline{\mathtt{ab}}$, the exact parse tree—either 00 or 11—depends on the disambiguation strategy. The greedy leftmost strategy is the one that causes a parser to greedily take the left option in the first choice, consuming an $\underline{\mathtt{a}}$ from the input string, and thereafter take the left option in the second choice, consuming a $\underline{\mathtt{b}}$. It thus produces the parse tree 00. □

Example 15. If the expression E_{amb} were a bit different, say,

$$E'_{amb} = (\underline{\texttt{a}} + \underline{\texttt{ab}})(\underline{\texttt{c}} + 1),$$

the parse tree 00 would not be valid for the input string $\underline{\texttt{ab}}$. On this input, the parser would first take the left option and consume the $\underline{\texttt{a}}$, as above. The following $\underline{\texttt{b}}$ would cause the left option of the second choice to fail, as it is a $\underline{\texttt{c}}$, which makes the parser try the right branch. But this one also fails, as that branch is 1! Having explored all options at this level, there must be an earlier choice that was wrong. The parser backtracks to the previous choice, picks the right option, and again tries the left option of the second choice. This of course fails again, so the right option is tried, which completes the parsing as only the empty string remains in the input. The greedy parse is 11. □

There is an elegant correspondence between the greedy leftmost strategy and the *lexicographical ordering* on bitcodes. A lexicographical ordering is the intuitive "alphabetical" ordering, where the first letters in two strings that differ determine the ordering between them:

$$\begin{aligned} 0s &< 1s' \\ 0s &< 0s' \text{ if } s < s' \\ 1s &< 1s' \text{ if } s < s' \end{aligned}$$

and the two strings are equal otherise. The greedy leftmost strategy picks the least bit-code amongst the possible parses according to this ordering.

Usage The greedy strategy is most notably used in Perl [19], and therefore also in engines that are "Perl-compatible", like PCRE [8].

10.2 POSIX

The other common disambiguation strategy we refer to as POSIX, after its defining body [10]. This strategy prefers the *longest, leftmost parse*, prioritizing the *length* of the string that a particular subexpression matches. Roughly, it will try to shift as much of the parse tree to the left in the expression.

Example 16. Let us inspect the expression E_{amb} on the input $\underline{\texttt{ab}}$ again. The POSIX strategy will cause parsing to try to consume as much as possible of the input string in the first subexpression $(\underline{\texttt{a}} + \underline{\texttt{ab}})$. Luckily, the whole input can be processed by selecting the right option. In the next subexpression, the right option must therefore also be taken, as no input symbols remain. This makes the POSIX parse tree of $\underline{\texttt{ab}}$ on E_{amb} 11. □

The POSIX strategy is more complicated than the greedy because of the dependence on information about the length of matched strings in the various subexpressions.

References

[1] ABC News. How many ways can you spell 'gadaffi'? http://abcnews.go.com/blogs/headlines/2009/09/how-many-different-ways-can-you-spell-gaddafi/. Sept. 2009 (cited on p. 1).

[2] Ascii subset of unicode. http://www.unicode.org/charts/PDF/U0000.pdf (cited on p. 3).

[3] J. A. Brzozowski. Derivatives of regular expressions. *J. acm*, 11(4):481–494, 1964. ISSN: 0004-5411. DOI: 10.1145/321239.321249 (cited on p. 34).

[4] D. Dubé and M. Feeley. Efficiently building a parse tree from a regular expression. *Acta informatica*, 37(2):121–144, 2000. DOI: 10.1007/s002360000037.

[5] J. Earley. An efficient context-free parsing algorithm. *Communications of the acm*, 13(2):94–102, 1970. ISSN: 0001-0782. DOI: 10.1145/362007.362035 (cited on p. 2).

[6] A. Frisch and L. Cardelli. Greedy regular expression matching. In *Proc. 31st international colloquium on automata, languages and programming (icalp)*. Vol. 3142. In Lecture notes in computer science. Springer, Turku, Finland, July 2004, pp. 618–629. DOI: 10.1007%2F978-3-540-27836-8_53.

[7] 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)*. Vol. 7982, in Lecture Notes in Computer Science (LNCS), pp. 60–71. Springer, July 2013. DOI: 10.1007/978-3-642-39274-0_7 (cited on p. 25).

[8] P. Hazel. PCRE – Perl-compatible regular expressions. http://www.pcre.org/pcre.txt. Jan. 2010 (cited on pp. 2, 35).

[9] F. Henglein and L. Nielsen. Regular expression containment: coinductive axiomatization and computational interpretation. *Sigplan notices, proc. 38th acm sigact-sigplan symposium on principles of programming languages (popl)*, 46(1):385–398, Jan. 2011. DOI: 10.1145/1925844.1926429 (cited on pp. 23, 24).

[10] IEEE Computer Society. *Standard for information technology - portable operating system interface (POSIX), base specifications, issue 7*. IEEE Std 1003.1. IEEE, 2008. DOI: 10.1109/IEEESTD.2008.4694976 (cited on p. 35).

[11] S. C. Kleene. Representation of events in nerve nets and finite automata. *Automata studies*, 1956 (cited on p. 2).

[12] D. Kozen. A completeness theorem for Kleene algebras and the algebra of regular events. *Information and computation*, 110(2):366–390, May 1994. DOI: 10.1006/inco.1994.1037 (cited on pp. 21, 23).

[13] M. Might, D. Darais, and D. Spiewak. Parsing with derivatives: a functional pearl. In *Acm sigplan notices*. Vol. 46. (9). ACM, 2011, pp. 189–195. DOI: 10.1145/2034574.2034801.

[14] L. Nielsen and F. Henglein. Bit-coded regular expression parsing. In *Proc. 5th int'l conf. on language and automata theory and applications (lata)*. In Lecture Notes in Computer Science (LNCS). Springer, May 2011, pp. 402–413. DOI: 10.1007/978-3-642-21254-3_32.

[15] Online regex tester and debugger. http://regex101.com (cited on p. 4).

[16] M. Sulzmann and K. Z. M. Lu. Posix regular expression parsing with derivatives. In *Proc. 12th international symposium on functional and logic programming*. In FLOPS '14. Kanazawa, Japan, June 2014. DOI: 10.1007/978-3-319-07151-0_13.

[17] M. Sulzmann and K. Z. M. Lu. Regular Expression Sub-matching Using Partial Derivatives. In *Proceedings of the 14th symposium on principles and practice of declarative programming*. In PPDP '12. ACM, Leuven, Belgium, 2012, pp. 79–90. ISBN: 978-1-4503-1522-7. DOI: 10.1145/2370776.2370788.

[18] K. Thompson. Programming techniques: regular expression search algorithm. *Commun. acm*, 11(6):419–422, 1968. ISSN: 0001-0782. DOI: 10.1145/363347.363387 (cited on pp. 2, 8).

[19] L. Wall, T. Christiansen, and J. Orwant. *Programming Perl*. O'Reilly, 3rd ed., July 2000. ISBN: 978-0-596-00492-7 (cited on pp. 2, 35).