Contents

List of Figures

List of Tables

Chapter 1

Introduction

Given a string and a regular expression, regular expression matching is the problem of finding whether the string is matched by the regular expression or not. Regular expression sub-matching is the problem of finding which part of regular expression (sub-expression) matches which part of string (sub-string). Parsing of a regular expression produces a clear picture of the sub-expressions which when constituted all together form the given regular expression. When a regular expression is parsed, it produces a parse tree. This parse tree can be a solution to both regular expression matching and sub-matching problems.

Regular expressions found its applications in many areas of computer science like validating, filtering, and classifying input provided. In practical, a number of implementations of regular expressions are moderately very slow even for a small matching problem. Regular expression matchers which use back-tracking technique in their matching algorithms exhibit an exponential behavior for certain kind of inputs which incorporates repetition for complex sub-expression of a regular expression [4].

Authors of this[3] journal paper devised a new methodology to compute POSIX parse trees for a regular expression based on Brozozowski's regular expression derivatives [5]. This report elucidates the implementation of the algorithm coined by Martin Sulzmann and Lu in the above mentioned paper[3]. As per the authors of the above paper, they are the first to develop an efficient algorithm for the construction of POSIX parse trees.

1.1 Motivation and Objectives

Authors of this paper[3] Sulzmann and Lu devised an algorithm to check whether a string word is matched by a regular expression or not. They employed an old concept coined by Brozozowski namely Brozozowski's Derivative [5]. Apart from devising a new algorithm, authors Martin Sulzmann and Lu affirmed that they are the first to propose an algorithm for the construction of POSIX parse trees. The aims of this project are to understand the concept of Brozozowski's derivatives, POSIX policies, understand the paper *POSIX Regular Expression Parsing with Derivatives*[3] and then implement it. This can be achieved first by reading the relevant literature about regular expressions and Brozozowski's derivative.Gain better understanding of the primary paper $POSIXRegularExpressionParsingwithDerivatives$ and implement the POSIX regular expression parsing using derivatives algorithm. After implementing the algorithm, compare the results with the already existing algorithms.

1.2 Overview

The organisation of the paper is as follows: Chapter 2 explains some basic definitions of regular expressions. It also provides an introduction to disambiguation techniques, explains various disambiguation techniques followed by a brief introduction to various regular expression matching algorithms. It also describes the concept of backtracking algorithm.

Chapter 3 describe the specification of the programming language used and concentrates on explaining the the platform and programming language choice used while implementing the algorithm.

Chapter 4 starts with providing an overview of the POSIX parsing of regular expression with derivatives algorithm. It mainly deals with the algorithmic details and makes use of an example to make explanation easier.

Chapter 5 evaluates the implemenatation of the algorithm and also mentions the problems faced.

Chapter 6 provides future scope for further work and also conclusion.

Chapter 2

Background

2.1 Overview

This chapter provides the history of regular expressions. it also provides a basic introduction to various definitions of regular expressions. It then provides an introduction to ambiguity when parsing a regular expression and explains various disambiguation techniques to match a string against a regular pattern. This chapter concludes by providing an explanation to backtracking and catastrophic backtracking techniques.

2.2 Regular expressions and their definitions

The concept of regular expression was first explained by Stephen Cole Kleene[6]. The general notion of regular expressions is as patterns signifying arrangements of a set of strings. Under this understanding, we can plan the issue of figuring out if a given string is contained in the set determined by some regular expression. This sums up the issue of figuring out if a particular string happens as a substring in a bigger content of strings. The inspiration for utilizing regular expressions for simple string search is that they are computationally sufficient to give assurance of effective running time and memory use, which is not the situation for more expressive formalisms, for example, context free grammar [7].

Regular expressions are being used very commonly among software engineers and power users, and we have their uses in numerous applications, for example, lexical analysis in compilers and scripting. Well known implementations are incorporating regular expressions in UNIX devices (for example, *sed*, *grep* and *awk*), word processors (for example, *emacs*)[8], and some programming languages even incorporate them as top of the inline developments [9] being a remarkable case. What these implementations have in similar is that they all execute the conventional regular expression of Kleene, albeit some include augmentations, yielding a great dealo f more expressiveness (and in a way computationally more perplexing) formalism. A striking sample is again Perl which includes "back-references" [10].

Definition 2.1. An expression T is said to be regular when one of the following is true[8]:

- *T is an expression which is atomic i.e. comprising of only one alphabet b from a set of alphabets $\sum$, or a special symbol 1 and 1 does not appear as an alphabet in $\sum$.*
- *If T_1 and T_2 are regular expressions, then $T_1 + T_2$, T_1T_2, or T_1^* are termed as compound regular expressions. [1] The set of alphabets represented by $\sum$ is assumed to be a finite and depending upon the application contents of the set varies.*

If we are given an RE T, now $L(T)$ represents the set of strings that are represented by T. $L(.)$ can be considered as a medium or a function explaining a regular expression as an arrangement of a set of strings. For the atomic regular expressions , the language interpretation is straightforward:

$$L(b) = b$$

$$L(1) = \varepsilon$$

Thus, the regular expression b denotes a set which comprises of only one single-letter string b. when a string contains no characters or letters in it, then we term that string as an empty string and we denote an empty string using a special symbol ε. Therefore, regular expression 1 contains only one element (empty string) and hence it is a singleton set. An empty string is different from an empty set.

In the above, compound expression $T1 + T2$ directly corresponds to the union of the two regular expressions languages T_1 and T_2. Thus,

$$L(T_1 + T_2) = L(T_1) \cup L(T_2)$$

The compound expression E_1E_2 can be formulated as follows:

$$L(T_1T_2) = \{p_1p_2 | p_1 \in L(T_1), p_2 \in L(T_2)\}$$

In the above expression, we use p_1 and p_2 which constitutes the set of strings in the languages T_1 and T_2 respectively. Now, $L(T_1T_2)$ informs us of pre-pending strings from

[1]These expressions sometimes are written as $T|S$, TS, $T * S$ where T and S are regular expressions.

the language T_1 with all the strings that are in T_2. The compound expression T^* can be depicted as follows:

$$T^0 = 1$$

$$T^n = T\ T\ldots\ldots\ldots\ldots T\ (n\ times)$$

Thus, T^n is the concatenation of T repeatedly for n times. T^0 specifies of concatenating a string in T zero times which in turn equals to an empty string. This can be formulated as follows:

$$L(T^n) = \{\varepsilon\} \bigcup_{n=1}^{\infty} (T)^n$$

Definition 2.2. The structure of regular expressions is defined by the rules[8]

$$T\ ::=\ a \mid 1 \mid T_1 + T_2 \mid T_1T_2 \mid T_1^*$$

Where it is supposed that a finite set of letters $\sum$ is given and $b \in \sum$.

Definition 2.3. The language interpretation for $L(.)$ is defined as follows[8]

$$L(a) = a$$

$$L(1) = \varepsilon$$

$$L(T_1 + T_2) = L(T_1) \cup L(T_2)$$

$$L(T_1T_2) = p_1p_2\ |p_1 \in L(T_1), p2 \in L(T_2)$$

$$L(Tn) = \bigcup_{n=0}^{\infty} (T)^n$$

Definition 2.4. Type Interpretation[8]

We can say that a regular expression contains a structure and this structure corresponds to a type. Various ways of constructing a regular expression are:

- *Concatenation*

  When we say concatenation in the scope of regular expression, it corresponds to a sequence i.e. a particular order should be followed on what comes after what. For example, if the RE is '*abcd*', RE is following concatenation and a particular order should be followed in the form of '*b*' after '*a*', '*c*' after '*b*', '*d*' after '*c*'. We can also call this concatenation in regular expressions as *Product Type*.

- *Alternation*

  When we say alternation in the scope of regular expressions, it regards to choosing between the left hand side and the right hand side of the symbol '+'. For example, if the RE is '$a + b$', RE is following alternation i.e. we say a match occurred with a string if the corresponding is string is either 'a' or 'b'. We can also call this alternation in regular expressions as *Sum Type*.

- *Kleen Star*

  In regular expression, this kleen star relates to any finite number of repetitions of something.

A *type* is something that portrays the *structure* of items in a set. we can finalize that these regular expressions can be considered as types. We compose the types similarly as the ordinary regular expressions. The way how a regular expression signifies an arrangement of strings, a regular expression translated as a type depicts a set of structured values. These are denoted using the notation of $L(.)$. $L(.)$ maps a regular expression to the corresponding set of structured values.

$$T(0) = \Phi$$

$$T(1) = \{(\ )\}$$

$$T(r) = r$$

$$T(A_1\ A_2) = T(A_1) \times T(A_2)$$

$$T(A_1 + A_2) = T(A_1) + T(A_2)$$

$$T(A_1^*) = \{[b_0, ..., b_n] \mid b_i \in T(A_1)\}$$

The structured values defined above can be depicted as parse trees. When we are talking about trees then there should be nodes and leaves. When forming a parse tree for a given regular expression RE, we can say anything that doesn't accept any arguments can be termed as a leaf i.e. in an RE, alphabets can be termed as leaves and anything that does take arguments can be termed as a node i.e. structured values mentioned above can be considered as a node.

2.3 Disambiguation Strategies

When given a string and a regular expression pattern, and if we consider parsing problem of regular expression, parsing produces a parse tree which gives a detailed picture of which part of regular expression matches which part of the string (that has to be matched against the given regular expression). Sometimes result of the parsing can be ambiguous as it may produce more than one parse tree for the regular expression pattern. In other words, when we say that patterns or regular expressions are ambiguous, it means that there are several ways of matching a string input with a given regular pattern.

In general, there are three disambiguation strategies for regular expression patterns.

1. POSIX Policy [11]
2. First and Longest Match Policy[12]
3. First Match and Greedy Match Policy [10]

2.3.1 POSIX[1]

POSIX or "Portable Operating System Interface" is a collection of benchmarks that characterize a portion of functionalities that should be supported by an operating system (UNIX). Unix was chosen as the premise for a standard framework interface mostly on the grounds that it was *independent of the manufacturer*. However, a several noteworthy adaptations of Unix existed so there was a need to build up a system that is common to all the operating systems. POSIX for regular expressions defines two kinds of regular expressions. Commands which include regular expressions such as `grep` and `egrep`, execute these two kinds on POSIX-compatible UNIX systems. A few database systems also utilize POSIX regular expressions.

The two major kinds of regular expressions defined by POSIX are **Basic Regular Expressions**[1] and **Extended Regular Expressions** [1] [13]. The Basic Regular Expressions or BRE flavor standardize a flavor like the one utilized by the customary UNIX `grep` order. One thing that separates BRE is that most metacharacters require a backslash to give the metacharacter its true meaning. For example, BRE $ab\{1,2\}$ matches $ab\{1,2\}$ literally, while $ab\backslash\{1,2\backslash\}$ matches ab or abb. BRE doesn't support any other features it includes alternation too.

The Extended Regular Expressions or ERE standardizes a flavor like the one utilized by the UNIX `egrep` command. *Extended* is with respect to the UNIX grep, which just had bracket expressions, ., $*$, \$. In ERE, the quantifiers $?, *, \{x\},\ \{x,y\}\ and\ \{x,\ \}$ repeat the preceding token zero or once, once or more, x times, between x and y times, x or more times respectively. For example, ERE $(pqr)\{2\}$ matches $pqrpqr$.

A glossary of the POSIX rules:[14]

- Regular expressions always consider the match beginning furthest left, and the longest match starts from there.

- Sub-expression that appear early in the regular expression possess leftmost-longest priority over those sub-expressions that appear later.
- In POSIX regular expressions associativity for concatenation is right and this associativity can be modified using parenthesis.
- Sub-expressions in regular expression that are Parenthesized return the match from their last usage.
- Contents of higher-level sub-expressions should contain the content of component sub-expressions.
- Text of component sub-expressions must be contained in the text of higher-level sub-expressions.
- $x|y$ and $y|x$ are equivalent if x and y can never match the same text, up to trivial renumbering of captured sub-expressions.
- Extra repetitions of x in x^* will not match an empty string, if x in x^* is used to match non-empty string.

2.3.2 First and Longest Match Policy

The second matching policy which guarantees a disambiguous matching is First and Longest match policy. This policy contains two rules that should be followed:

- First match policy:
  As per first match policy, whenever a pattern of the form $P + Q$ given, priority is given to the left side sub-expression i.e. P. Moreover, alternation distributes over concatenation, it means that when matching an input string r against $(P + Q).R$, R should be first matched against the pattern $P.R$. Priority should be given to $Q.R$ only when matching against $P.R$ fails.

- Longest match policy:
  The longest rule deals with the confusion raised when the Kleene closure pattern comes into the picture. If the pattern is of the form $P^*.Q$, then it tries to match the input as much as possible against P^*, yet permitting the remaining pattern to match.

2.3.3 First Match and Greedy Match Policy

Like first and longest match policy, even this policy consists of two guidelines to be followed:

- First match rule

  The first match rule in *First Match and Greedy Match Policy* is same as the first match rule in *First and Longest Match Policy.* Therefore, first match rule disambiguates whenever alteration pattern (of the form $P + Q$) by giving priority to the left side subexpression.

- Greedy match rule

  Greedy match rule tries to remove the ambiguities in case of Kleen closure. Whenever a pattern under Kleen closure appears say P^*, greedy match rule disambiguates this kleen closure pattern by recursively modifying P^* into $(P.P^* + \varepsilon)$. Unlike "POSIX" and "First and Longest Match policy", "First Match and Greedy Match Policy" doesn't implement longest match policy. Therefore, if

  $$T1 = (((P\ .\ Q)\ +\ P)^*\ .\ Q)$$

  $and\ T2 = (P\ +\ (P\ .\ Q)) * .Q)$ and the word is PQ, when matching PQ against $T1, ((P.Q) + P)^*$ matches PQ. When matching PQ against $T2, (P + (P.Q))^*$ matches only P.

2.4 Catastrophic Backtracking

There are mainly three different algorithms using which one can stipulate whether a regular expression matches a string or not [15].

First and speediest depends on an outcome in formal language and automata theory that permits each nondeterministic finite automaton (NFA) to be changed into a deterministic finite machine (DFA). The DFA can be built expressly and afterward keep running on the subsequent information of the string one alphabet at once. Developing the DFA for a regular expression of size 'k' has sufficient time and memory expense of $O(2k)$, however it can be keep running on a string of size 'n' in time $O(n)$.

Second option is to model NFA computationally, choosing and constructing DFA only when required and then discarding it at the successor step. This keeps the DFA suggested in required scenarios and maintains a strategic distance from the exponential development cost, however running cost ascends to $O(kn)$ [2].

The third algorithm employs a technique named backtracking to match a string against a regular expression pattern. When used this algorithm, sometimes the time complexity is of exponential order for inputs of type $[a^n a^n]$ [3] and input of the form $[a^n]$ [4].

Backtracking is an algorithm for discovering all answers for some computational problem which produces probable solutions to the problem in question [16]. This backtracking works in such a way that, it deserts every other solution K when it verifies that K cannot in any way be a finished and valid solution to the problem.

When using a regular expression matcher which employs backtracking to address the matching problem of a regular expression, in some instances, for some specific inputs, backtracking causes the matcher to extend the time taken to match the input string. This concept is referred to as Catastrophic Backtracking. It can be utilized to launch regular expression denial of service attacks, called as ReDos attacks.[17]

Catastrophic backtracking often arises (although not mandatorily or solely) when the given regular expression (or a sub-expression of the regular pattern) is of the form R^* (R being the regular pattern or a part of it), where R could match a non-empty string r in many ways. Thus an input string r^k can be matched exponentially by R in many ways (without using the concept of subset construction).

2.4.0.1 Under what circumstances regex engine with backtracking feature exhibits exponential time behavior?

In general, if a regular expression matcher tries to match input as much as possible, when it encounters $*$ (zero or more repetitions) or $+$ (one or more repetitions), that behavior is termed as *Greedy matching*[18]. If a regular expression matcher tries to match input as least as possible, then that behavior is called *Lazy matching*[18]

Backtracking is a very sophisticated and useful methodology being employed in modern regular expression matching engines. If an expression (or a sub-expression) fails to match a string, the matcher backtracks to a certain position from where it could have taken an alternate way.

[2] k being the size of the regular expression and n being the size of the input string
[3] 'n' being relatively large, say 28
[4] when we say a^n, it means that a is repeated and concatenated n-times

In case of no matching, a regex matcher employing a greedy matching methodology will let go off one letter from its last matching whereas a regex matcher with lazy matching methodology may expand its match to one more character. If the regular expression matching fails repeatedly, then the matcher investigates all the available ways to match the input string against the regular expression.

The number of ways of matching an input string against a regular expression increases when the input string can be matched in more than one way. For example, if the regular expression is $(a+a+)+b$ (contains four sub-expressions $a+$, $a+$, $(a+a+)+$ and b) and the string to be matched is $aaaaaaaa$.[19]

Now, the matching of the string $aaaaaaaa$ against the pattern $r = (a + a+) + b$ works as follows : First the whole string will be matched by first $a+$ and the second $a+$ fails since there are no more letters in the string. Therefore the first $a+$ backtracks and let go off one a. Now, first $a+$ matches seven a's and second $a+$ matches one a and since b is missing, the matching fails. Regex matcher backtracks and the first $a+$ let go off another a and hence matches six a's and second $a+$ matches aa. Again b fails and backtracks. This time, second $a+$ has a provision to backtrack and hence let go off one a thus matching only one a. Now matching occurs at first $a+$ but now fails at second $a+$ and also at b. Regex matcher backtracks. This time first $a+$ let go off another a thus matching five a's and second $a+$ matching aaa. In this way, the regex matcher explores all the possible ways to match the given input string against the regular pattern. If the input string contains considerably large number of a's one can practically explode the regular expression matcher.

In this way, if quantifier token like $*$ or $+$ or m, n is used improperly like in the above example, an input string can be matched in many ways thus providing a provision for regular expression matcher to explore many paths in obtaining a solution and this in-turn increases the time taken by the matcher to match the input string against the regular expression.

Chapter 3

Specifications and Design

3.1 Overview

This section introduces the details about programming language used for implementing the algorithm and also the technical specifications of the platform used.

3.2 Programming Language - C#

C# is the programming language used in the implementation of the algorithm proposed by Martin Sulzmann and Lu for POSIX regular expression parsing using derivatives. C# is an object oriented programming language. Other object oriented programming languages like Java, $C++$ and functional languages like *Scala* can also be used for the implementation. The principle reason in choosing C# is not only because it is one of the most widely used languages for developing web applications, mobile applications but it also has very good performance in algorithmic configuration and implementation.

There are many advantages in using C# as a programming tool in implementing algorithms. While implementing an algorithm, data structures play a major role in deciding the time complexity of the algorithm. In C#, there are in-built libraries for stacks, queues, linked lists, etc. Therefore, all those data structures can be used like any other data types like *int*, *char* or *boolean*. While implementing the algorithm, the regular expression has to be converted to a tree like structure. Therefore, in implementing the algorithm using pre-defined data structures like stacks and queues saves a lot of time. Since C# is an object oriented code it can be utilized in imbibing user-defined behavior to the program using a specific programming concept called Classes. These classes are a set of instructions to define a specific type of object and can be re-usable.

3.3 Platform

Environment used for the implementation of the algorithm is MonoDevelop. It is an open source integrated development environment for various operating systems like Linux, OS X and Windows. Its principle focus is in developing projects that utilize .Net frameworks[20]. Even though the main programming environment for C# is Microsoft .Net framework, MonoDevelop can also be used for C# on mac OS X as it combines all the features together that are present in .Net framework of which garbage Collector, a graphical user interface, automated code is to name a few. When we use MonoDevelop for programming, programs are executed not in hardware environment but in software environment. MonoDevelop contains functionalities similar to that of .Net framework. Therefore, MonoDevelop is an incredible integrated development environment for C# programming.

Chapter 4

POSIX Regular Expression Parsing with Derivatives algorithm

This chapter introduces some relevant and important concepts to have a better understanding of the *POSIX parsing of regular expressions with derivatives* algorithm. Most of the techniques mentioned in this chapter will be used in the implementation of the algorithm. To have a clear picture, algorithm is explained with the help of an example.

4.1 Overview

Brozozowski's derivatives of a regular expression is an old, yet rich, system for transforming regular patterns to deterministic finite automata. It effortlessly incorporates extending the regular expression operators with boolean operations, for example, intersection and complement. Tragically, this method has been lost in the sands of time and few computer researchers know about it.[2]

In [5], Janusz Brzozowski introduced a sophisticated algorithm for straightforwardly building a matcher from an RE taking into account derivatives of regular expressions (Brzozowski, 1964). His methodology is sophisticated and effortlessly bolsters regular expression which are extended with the aid of operations which are boolean. [2]

Martin Sulzmann and Lu utilized the concept explained by Brozozowski, the derivatives of regular expression in order to tackle the problem of POSIX matching. Sulzmann and Lu extended the concept of derivatives into the POSIX matching of a regular expression.

Sulzmann and Lu calculated a value which explains how a string is being matched by a regular expression and then proposed a function on those values which reverts back the process of building of derivatives on a regular expression. The detailed explanation of all the steps in the algorithm proposed by Sulzmann and Lu is described in the following sections.

4.1.1 Brozozowski's derivative of regular expressions:

To calculate derivative of a regular expression as another regular expression, first we need a mediator function, F, from a regular expression to another regular expression[2]. In other words, derivative of a regular expression can be calculated by using two recursive functions.

First function takes a regular expression and produces a boolean value, second function takes a regular expression and a value and produces another regular expression. First function is to determine the nullability of a regular expression whereas as second function is to calculate derivative of a regular expression.

4.1.1.1 Nullability of a regular expression:[2][3]

A regular expression RE is said to be nullable if the language identified by the given regular expression contains the empty string. The function Nullable can be defined as follows provided 'r' is the regular expression:

$$Nullable(r) = \begin{cases} \varepsilon \; if \; r \; is \; nullable \\ \Phi \; otherwise \end{cases}$$

and is defined as follows:

$$Nullable(\varepsilon) = true$$

$$Nullable(\Phi) = false$$

$$Nullable(x) = false$$

$$Nullable(x \cdot y) = Nullable(x) \wedge Nullable(y)$$

$$Nullable(x + y) = Nullable(x) \vee Nullable(y)$$

$$Nullable(x^*) = true$$

4.1.1.2 Derivative of a regular expression[2][3]

The concept of derivatives can be applied to any language. If L ($L \subseteq \Sigma*$) is a language, then the derivative of L with respect to a symbol $x \in \Sigma$ is the language that contains only those substrings of the given set of strings which are obtained by removing the prefix x from each string (only if the string starts with x).

If 'R' is the regular expression, then the set of strings represented by 'R' is termed a s regular set of strings. If derivative is applied on the regular set of strings, then the result is also a regular set.[2]

The following are the rules proposed by Brozozowski to calculate the derivative[1] of a regular expression:[5]

1. $\Phi \ \backslash d = \Phi$
2. $\varepsilon \ \backslash d = \Phi$
3. $c \backslash d = \begin{cases} \varepsilon if \ c = d \\ \Phi otherwise \end{cases}$
4. $(p_1 \ + \ p_2) \backslash d = (p_1 \backslash d) \ + \ (p_2 \backslash d)$
5. $(p_1 \ . \ p_2) \backslash d \ = \begin{cases} (p_1 \backslash d) \ . \ p_2 + (p_2 \backslash d) \ if \ Nullable(p_1) \\ [(p_1 \ \backslash d) \ . \ p_2] \ Otherwise \end{cases}$
6. $(p^*) \backslash d = (p \backslash d) \ . \ (p^*)$

4.1.2 mkeps function

When a regular expression RE and a string that is to be matched against RE is given, first derivative of the RE is calculated with respect every character in the string word and checked whether the RE was nullable or not. If the given RE is nullable then the algorithm calls mkeps function, which produces a value that elucidates how the derivative result of the RE matches the empty string. Therefore, mkeps function is called only when the given regular expression is nullable. In case, if the given regular expression is not nullable, then the input string cannot be matched against the given regular pattern and hence program exits reporting an error. The mkeps function is defined by following rules:[3]

[1] $a \backslash c$ = derivative of a regular expression a with respect to a symbo c

1. $mkeps(\varepsilon) = ()$
2. $mkeps(p_1\ .\ p_2) = Seq\ (mkeps\ p_1)\ (mkeps\ p_2)$
3. $mkeps(p_1 + p_2) = \begin{cases} Left\ (mkeps\ p_1) if\ Nullable(p_1) \\ Right\ (mkeps\ p_2) Otherwise \end{cases}$
4. $mkeps(p^* = Stars\ []$

The function mkeps adheres to POSIX policies. POSIX policy always favors leftmost match. When there is a regular expression which contains alternation say, $p_1 + p_2$, if the given pattern matches an empty string and in addition, if p_1 matches the empty string, then mkeps returns left-value p_1. mkeps retuns right-value only when left value cannot match the empty string.[21]

4.1.3 Inject function

- The input to the inject function is a character value, a regular expression and a value. One of the inputs to inject function, a character value (each character in the input string word that is to be matched against the input regular expression) is supplied in the reverse order. For example, if the string that is to be matched against regular pattern is “abc”, then the order of character value inputs to the inject function are in the order ‘c’, ‘b’, ‘a’ in successive iterations.
- The inject function is characterized by recursion on regular expressions by examining the shape of the values regarding the derivative of the regular expressions. The inject function is defined by the following rules[3]:
    1. $inject\ d\ c\ ()\ =\ Char d$
    2. $inject\ (p_1\ +\ p_2)\ c\ (Left\ r_1) = Left\ (inject\ p_1\ c\ r_1)$
    3. $inject\ (p_1\ +\ p_2)\ c\ (Right\ r_2) = Right\ (inject\ p_2\ c\ r_2)$
    4. $inject\ (p_1\ .\ p_2)\ c\ (Seq\ r_1\ r_2) = Seq\ (inject\ p_1\ c\ r_1)\ r_2$
    5. $inject\ (p_1\ .\ p_2)\ c\ (Left\ (Seq\ r_1\ r_2)) = Seq\ (inject\ p_1\ c\ r_1)\ r_2$
    6. $inject\ (p_1\ .\ p_2)\ c\ (Right\ r_2) = Seq\ (mkeps\ p_1)\ (inject\ p_2\ c\ r_2)$
    7. $inject\ (p^*)\ c\ (Seq\ r\ (Starsvs)) = Stars\ (inject\ p\ c\ r :: vs)$
- We shall understand the definition rules of the inject function by having a close look at them.
    - In rule-2 we have to build an *injected value* for $p_1 + p_2$. This must be of the structure either $Left(\_)$ or $Right(\_)$ because

    * $(p1 + p2)\backslash a = (p_1\backslash a) + (p_2\backslash a)$

    The derivative consists alternation, therefore we have to consider either $Left-value(r_1\backslash a)$ or $Right-value(r_2\backslash a)$. If it is $Left-value$, then the value should be of the form $Left(r_1)$ which matches with the value on the left hand side of rule-2. If it is $Right-value$, then the value should be of the form $Right(r_2\backslash a)$ which matches with the value on the left hand side of the rule-3.

  - In rules 4, 5 and 6, we want to build an *inject value* for $p_1 \cdot p_2$. This value should be of the form *Seq* (_) (_).

    * $(p_1\ .\ p_2)\backslash a = \begin{cases} (p_1\backslash a)\ .\ p_2 + (p_2\backslash a) \text{ if } Nullable(p_1) \\ [(p_1\backslash a)\ .\ p_2]\ Otherwise \end{cases}$

    Consider the *else* branch of the derivative above, $(p_1\backslash a)\ .\ p_2$. Therefore, the value should be of the form *Seq*(_)(_) which matches with the value on left hand side of the rule-4. The $if - branch$ $(p_1\backslash a)\ .\ p_2 + p_2\backslash a$ consists alternation *i.e.* we have to choose either $Left - value$ or $Right - value$. If it is $Left - value$, then the value should be of the form $Left$ (*Seq* (_) (_)) which matches with the value on left hand side of rule-5. If it is $Right - value$, then the value should be of the form *Right*(_) which matches with the value on left had side of rule-6

  - In rule-7, we want to build an $injectedvalue$ for p^*. This must be of the form $Start$(_).

    * $p^*\backslash a = (p\backslash a)\ .\ p^*$.

    Since the derivative of a kleen star is $p^*\backslash a = (p\backslash a)\ .\ p^*$, the value should be of the form *Seq* (_) ($Stars$ (_)) which matches with the value on left hand side of rule -7. In the value of *Seq* (_) ($Stars$ (_)), the second part is $Stars$ (_) because $mkeps$ $(p^*) = Stars$ []

- Let us suppose the regular expression is r and the input string is *abc*. Now derivative results on r according to the string *abc* are as follows:

$$r\backslash a \Longrightarrow r_1$$
$$r_1\backslash b \Longrightarrow r_2$$
$$r_2\backslash c \Longrightarrow r_3$$

  Derivative result of the regular expression with respect to each character in the input string serves as one of the inputs to the inject function *i.e.* the regular expression input to the inject function corresponds to the derivative results of the regular expression served in reverse order.

  - If c, b, a are character value inputs to the inject function, then corresponding regular expression inputs to the inject function are r_2, r_1 and r respectively

- Output of the *mkeps* function serves as one of the inputs during first iteration of the *inject* function. In other words, *mkeps* calculates and produces an output value (v_4) which specifies how the derivative result of the regular expression with respect to the input string matches an empty string. This output value v_4 by *mkeps* serves as input to the *inject* function. During first iteration of the inject function, the input values are c, r_2, v_4.

- The flow of the *mkeps* and *inject* functions is as shown in the Figure 4.1[21]:

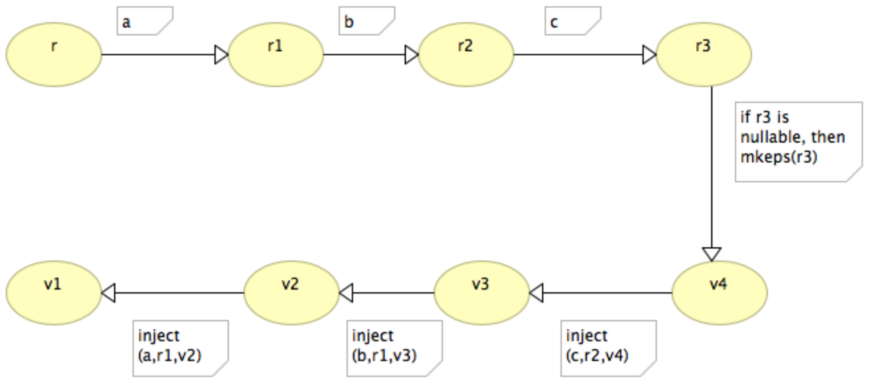


FIGURE 4.1: steps r to r_3 constructing derivatives successively. r_3 to v_4, if *nullable* (r_3) then calculate *mkeps* (r_3) yielding v_4. steps v_4 to v_1 is reverting the construction of derivatives for a regular expression by calling inject function

$$inject\ c\ r_2\ v_4 \Longrightarrow v_3$$
$$inject\ b\ r_1\ v_3 \Longrightarrow v_2$$
$$inject\ a\ r\ v_2 \Longrightarrow v_1$$

- First calculate the *derivative* of the regular expression with respect to every character in the input string. In the second step employ *nullability* test on the final derivative result of the regular expression to find out whether the give regular pattern matches an empty string or not. If yes *i.e.* regular pattern matches an empty string, then call the *mkeps* function which produces a value that describes how the final derivative result of the regular expression matches an empty string and this value is supplied as one of the input values to the *inject* function during the first iteration. By supplying all the inputs i.e. character values from the input string word, regular expression and a value, *inject* function calculates and produces a parse tree which shows how the regular expression matches the input string considering POSIX policies in case if there are several ways).

4.2 Converting regular expression into an expression tree

While parsing a regular expression, we compare and match the string word with all the sub-expressions of the regular expression. Therefore, it is very much essential to convert a regular expression into a structure which facilitates the checking of every sub-expression in the given regular expression. One of the many structures which is highly helpful in fulfilling the above specification is tree structure.

4.2.1 Data Structures

Data structures play an important role in building applications which are time and memory bound. Many data structures are required to support an algorithm. One of the data structures used in the implementation of the algorithm is tree. The tree structure selected is not of any specific type (like binary tree or AVL tree) but a basic tree in which a node can have any number of children.

The main integral part of the tree structure used in the program is formed by the user-defined data type known as Node. Every node in the tree consists of four fields:

- Data field
- First field
- Next field
- Parent field

In a tree, a node will have a parent (except root node) and can have children. If more than one node has same parent, then all those nodes are said to be siblings. Contrary to the general tree structure, the structure of the tree considered in implementing this algorithm differs slightly.

Data field of a node provides information about the data associated with a node. First field gives information about the first child of a node. Next field contains details about the sibling of a node. Parent field provides information about the parent of a node.

For example, if the regular expression is $a + (b + c)$ then the structure of the tree corresponding to the given regular pattern is shown in figure 4.2.

In the tree diagram above, there are in total three levels. Level-1 contains a node with data "+". In level-1

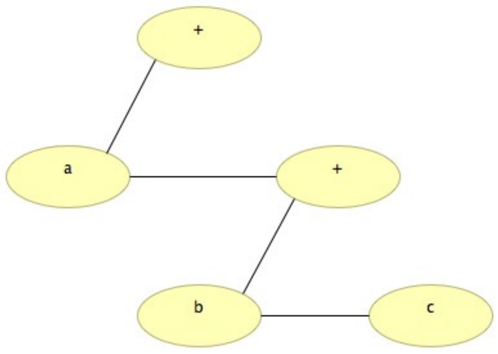


FIGURE 4.2: Tree Structure for a+(b+c)

- Node +'s data field = "+"
- Node +'s first field = node with data "a"
- Node +'s next field = *null*
- Node +'s parent field = *null*

In level-2 tree has two nodes, "a" and "+".

- Node a's data field = "a"
- Node a's first field = *null*
- Node a's next field = node with data "+" in level-2
- Node a's parent field = node with data "+" in level-1

The main reason in making the tree structure as above is, since we are working with regular expressions, sometimes more than two sub-expressions may be concatenated with each other. For example, if the regular expression is of the form "$(a + b)$ $(c + d)$ $(e + f)$ $(g + h)$", then the main root will have four children since we have four main sub-expressions in the given regular expression.

Now if we use traditional binary tree or an AVL tree, a node in those trees can only have maximum of two children. Contrary to that, in our program a tree node should have the capability to incorporate any number of children. The tree structure of above regular expression "$(a + b)$ $(c + d)$ $(e + f)$ $(g + h)$" is as shown in figure 4.3

Tree structure in the above figure has three levels. The level-1 contains root node of the tree. As per our regular pattern, four sub-expressions are concatenated. Therefore, our root node " " should have four children. All the nodes in the level-1 have same parent

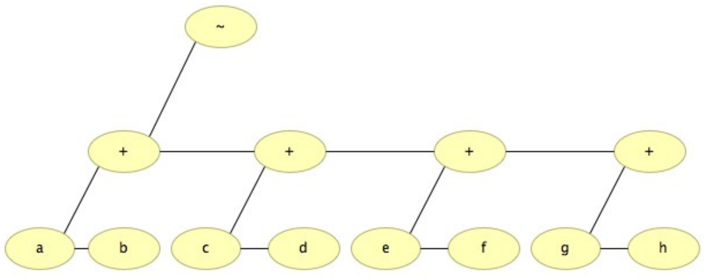


FIGURE 4.3: Tree Structure for $(a+b)(c+d)(e+f)(g+h)$

i.e. root node. In the tree structure above, all the children are not associated with the root node, instead, only first child "$(a+b)$" is linked with the root node.

Root node $\sim$ has direct access only to the first child. If from root node, one wants to traverse other children, then they can achieve this requirement with the aid of next field of first child of root node. For example, if one wants to access last child of the root node, it can be achieved by following the following link:

Root node = w

w's first field = "+" (first + node in level-1 with children a and b) = w_1

w_1's next field = + (second + node in level-1 with children c and d) = w_2

w_2's next field = + (third + node in level-1 with children e and f) = w_3

w_3's next field = + (fourth + node in level-1 with children g and h)

In this way, one can access all the nodes (children, siblings and parent) in the tree above. Next field in a tree node helps in associating a node with any number of children.

The other data structure used in the implementation of the algorithm is stack. While converting an expression into a tree, four stacks play a major role. The four stacks are

- operator_st
- operand_st
- node_st
- bracket_st.

The contents of operator_st, operand_st, node_st and bracket_st are of char, string, node and string data type respectively. The main use of stacks can be understood in the next section.

4.2.2 Conversion to a tree

The Program implemented accepts a regular expression as a string data type and later converts it into a tree considering precedence of operators present in the regular expression. The operator's precedence in regular expression is as follows [22]:

Priority	Operator	Description
Highest	()	For grouping operation
	$*$	Star operator for repetition
	‘.’	Concatenation operator
	$+$	Alternation operator

Table 4.1: order of precedence of the operators

The following are the assumptions made while processing the given regular expression RE:

- ‘!’ is treated as empty string.
- ‘@’ is treated as null.
- The operator $\sim$ is considered to be equivalent to concatenation operator ‘.’. Therefore, in the tree structure $\sim$ represents concatenation.
- When more than one character is encountered without any operators between them, say if the RE is abc, then it is assumed that the structure of the given regular expression is $a \sim b \sim c$.

The algorithm for converting a given regular expression to a tree works as follows:

- Given a regular expression RE, it is accepted as input string by the program.
- The program processes the given RE in reverse order of the string. For example, if the input RE is “a+b” then the program processes in the order of b, $+$, a.
- The program processes each character in the RE one by one. Depending upon the character now currently being processed, it pushes that character into the corresponding stack. If it is an operator, the program pushes the character onto the operator_st. If the character is an operand, it got pushed onto the operand_st. If the character is either $'('$ or $')'$, it is pushed onto the bracket_st.

- The program groups all the characters between '(' and ')' into one. Since the given RE is being read in the reverse order, the program first encounters ')'. When '(' encountered the program first checks whether there is corresponding ')' in the bracket_st stack or not by popping up the contents of the bracket_st stack. If the character read after performing popping operation on bracket_st is ')' then the program continues with its execution, if not, it exits the program and reports an error.

- Whenever the program reads an operator, before pushing this operator into operator_st, it checks the number of operands in the operand_st. If only one operand is there, then it creates a node with this one operand and then pushes onto node_st. If more than one operand is present in the operand_st, first it concatenates all the operands together into one node structure and then pushes this newly formed node into the node_st stack.

- Let us consider an example:
  - Consider the given RE is $(p + q).(s + t)$. At the start of the program, the count of the contents in all the stacks is empty.

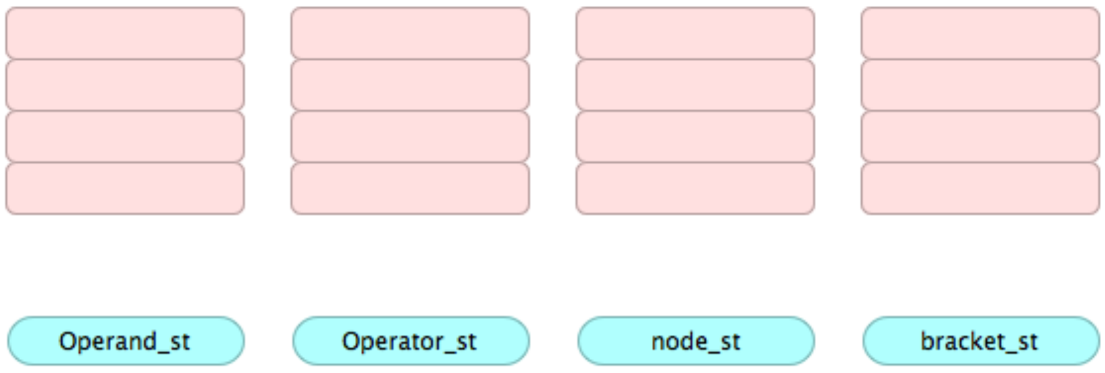


Figure 4.4: Snapshot of stacks at the beginning of the program execution

  - Since the input RE $(p + q).(s + t)$ is being processed in reverse order, first character ')' is read and pushed onto bracket_st. After that it reads t. Since t is an operand it got pushed onto operand_st. After t got pushed onto operand_st, the snapshot of stacks is as shown in Figure 4.5
  - Next character is $+$. Since it is an operator, first the program checks whether there are any contents in operand stack. In this case, yes, therefore converts t into a node and then adds to the node_st. After this, $+$ got pushed onto operator_st. After $+$ got pushed onto operator_st, the snapshot of stacks is as shown in Figure 4.6

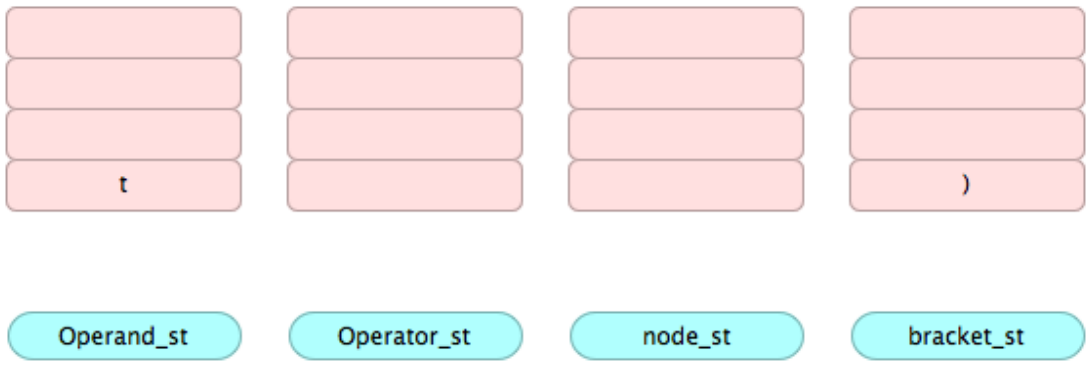


FIGURE 4.5: Snapshot of stacks after processing $')'$ and t

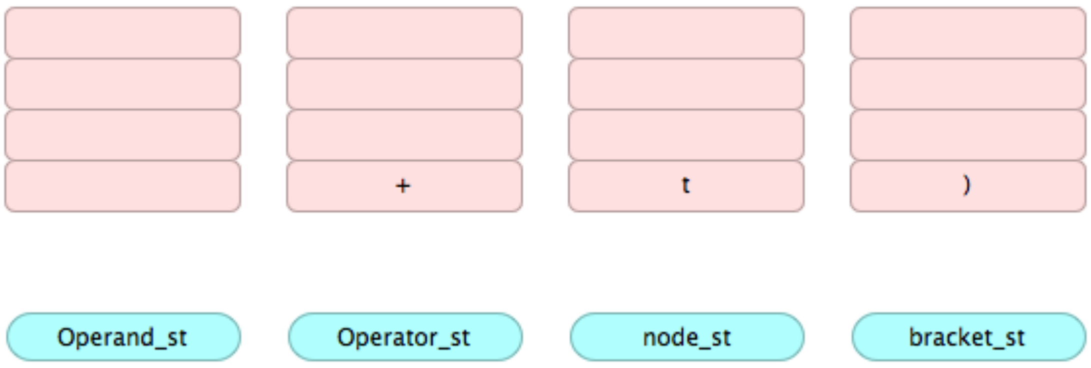


FIGURE 4.6: Snapshot of stacks after processing $')'$, t and $+$

– Next character is s. Therefore, it will get pushed onto operand_st. After s getting pushed onto operand_st, the snapshot of stacks is as shown in Figure 4.7

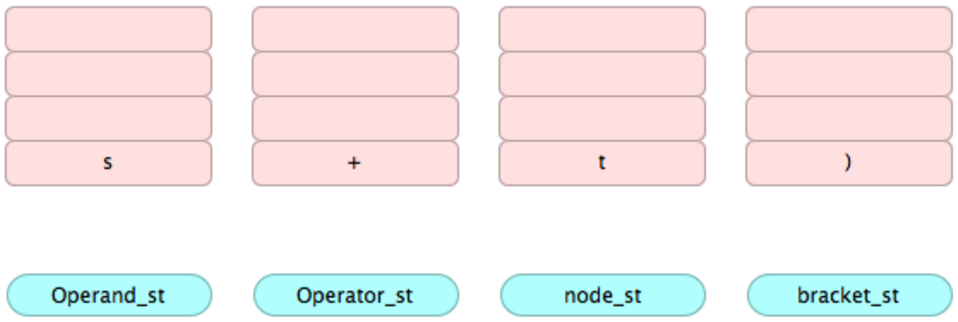


FIGURE 4.7: Snapshot of stacks after processing $"s + t)"$

– The next character is '('. Therefore, program first pops out the contents of bracket_st. if the popped out character is $')'$, then program continues its execution. If not, then it is evident that parenthesis is not matched and hence the program exits reporting an error.

In this case, when pop() operation is performed on bracket_st stack, ‘)′ is popped out. Therefore, program continues its execution by grouping all the characters present inside ‘(‘ and ‘)′ into one group.

At this juncture, when program is checking whether operand_st contains any operands or not, three cases come into picture:

∗ Case-1:
  <u>Operand_st is empty:</u>
  In this case, since there are no operands in operand_st, if the operator is ∗ then node_st contains only one node and that one node is associated with ∗. If the operator is + then top two nodes of the node_st are popped out and associated with +. Now the new node formed with + or ∗ as root is of node type and hence got pushed into node_st

∗ Case-2:
  <u>Operand_st contains only one element:</u>
  In this case, since there is only one operand in operand_st, the operand is popped out and get converted to a node and then pushed onto node_st. Now, the program executes as mentioned in Case-1.

∗ Case-3:
  <u>Operand_st contains more than one element:</u>
  In this case, first all the operands in operand_st are concatenated and then pushed onto node_st. After this step, program works as mentioned in Case-1.

Here Case-2 applies *i.e.* there is only one operand in operand_st. Therefore, it gets converted to a node and then pushed onto node_st. Now since the operator is + top two elements of the node_st are popped out and get associated with +. After this step the snapshot of all the stacks is as as shown in Figure 4.7:

All the stacks are empty except node_st which contains Element-1 and the structure of Element-1 is represented as shown in Figure 4.8:

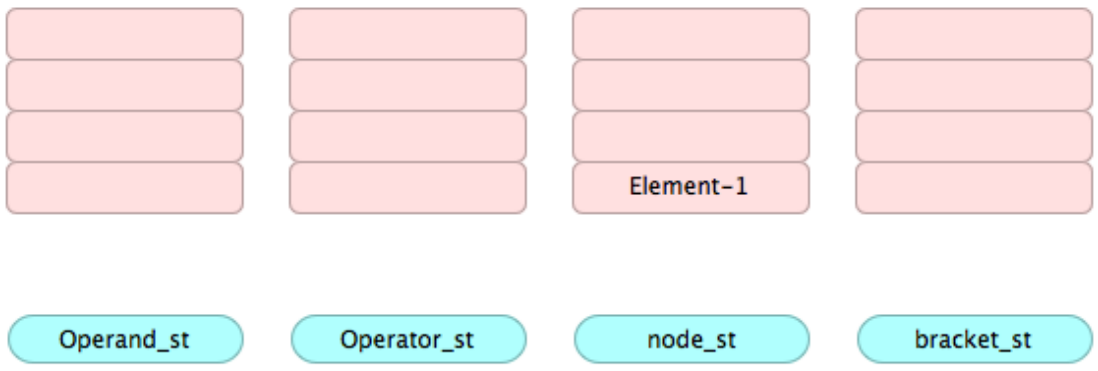


FIGURE 4.8: Snapshot of stacks after processing $(s + t)$

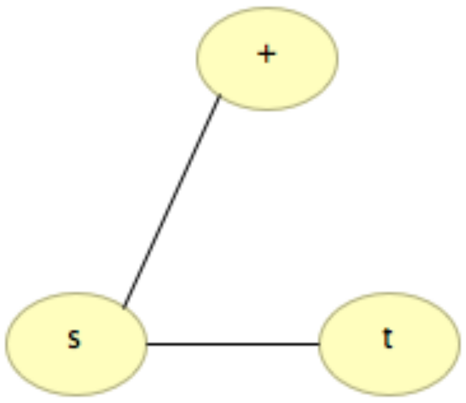


FIGURE 4.9: Structure of sub-string node $(s + t)$

– Now, the sub-string $(p + q)$ will also be processed in the same way the sub-string "(s + t)" was processed. After processing the whole string, the snapshot of all the stacks is as shown in Figure 4.9:

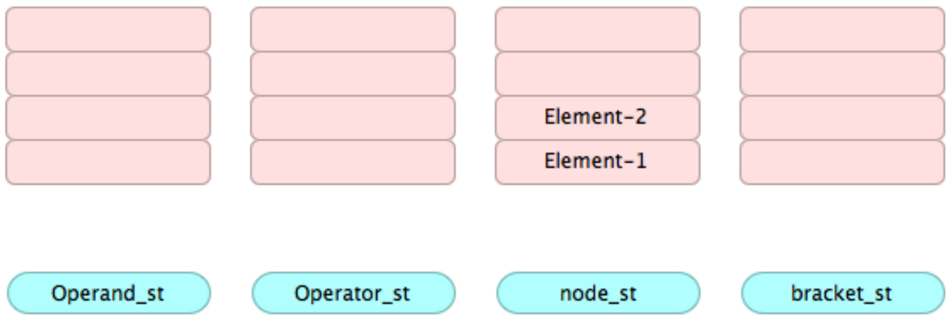


FIGURE 4.10: Snapshot of stacks after processing $(p + q).(s + t)$

Except node_st, all the remaining stacks are empty. Since, bracket_st is empty, parentheses are balanced (except in those cases where parenthesis may not be present). The structure of Element-1 is represented above and the structure of Element-2 is as follows:

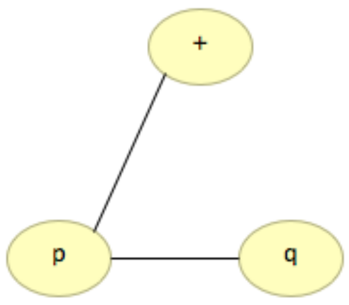


FIGURE 4.11: Structure of the sub-string node $(p + q)$

– At this step, whole string is processed. The program now behaves according to the number of elements in each of the stack. In this case, since all the stacks except node_st are empty, it concatenates all the elements in node_st and pushes the newly formed node after concatenation onto node_st. Finally, node_st contains only one node and the structure of that one node is as shown in the Figure 4.12

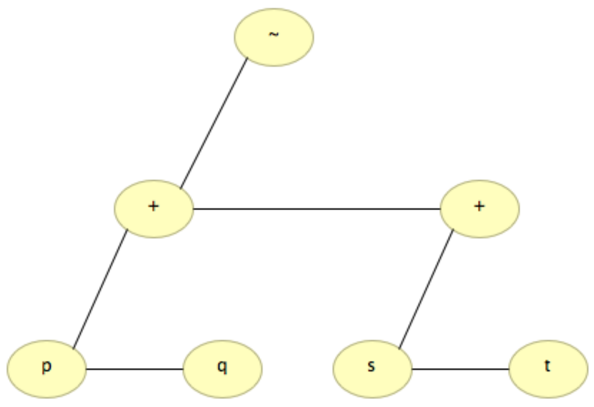


FIGURE 4.12: expression tree structure of the regular expression $(p + q(.(s + t)$

Therefore, the expression tree for the given regular expression $(p+q)(s+t)$ is represented as above

4.3 Derivative of a Regular Expression

We have already seen the rules defined by Brozozowski in Section 4.1.1.2 on how to construct derivative of a regular expression. We shall understand the computation of derivative for a regular expression with an example.

Let us consider a string *ps* that has to be matched against the regular expression $(p+q)(s+t)$. Since the input string contains two characters, derivative has to be calculated with respect to one character at a time. Therefore, derivative of the regular expression is first calculated according to p and then with respect to s.

$(p+q)(s+t) \sim\ ps \Longrightarrow\ [(p+q)(s+t)]\backslash p\ \sim\ s$
$\Longrightarrow\ [((p+q)\backslash p).(s+t)] \sim\ s$
$\Longrightarrow\ [(\varepsilon+\Phi).(s+t)]\ \sim\ s$
$\Longrightarrow\ [(\varepsilon+\Phi).(s+t)]\backslash s$
$\Longrightarrow\ [((\varepsilon+\Phi)\backslash s).(s+t)] + (s+t)\backslash s$
$\Longrightarrow\ [(\Phi+\Phi).(s+t)] + (\varepsilon+\Phi)$

Therefore, the derivative of the regular expression $(p+q)(s+t)$ with respect to the string word *ps* is $[(\Phi+\Phi).(s+t)]+(\varepsilon+\Phi)$.

4.4 Nullability test

We have already discussed in the earlier section 4.1.1., a regular expression r is said to be nullable if the regular language represented by a regular expression contains an empty string. In order to determine the nullability of a regular expression it is sufficient to calculate the nullability of the final derivative result of the regular expression. The rules for nullability test were listed in the section 4.1.1.1.

From rules of Nullability it is clear that, if the given RE contains alternation, say $p + q$, then for $Nullable(p+q)$ to be true either $Nullable(p)$ or $Nullable(q)$ or both should be true.
If the given RE is concatenation of sub-expressions, say $p.q$, then for $Nullable(p\ .\ q)$ to be true, both $Nullable(p)$ and $Nullable(q)$ should be true.

We shall perfrom nullability test on the regular expression $(p+q)(s+t)$" for which derivative was calculated in the previous section 4.3 and the derivative result is $[(\Phi + \Phi).(s+t)] + (\varepsilon + \Phi)$ with respect to the string ps.

$$Nullable[(\Phi+\Phi).(s+t)] + (\varepsilon+\Phi)] \Longrightarrow \underline{Nullable((\Phi+\Phi).(s+t))} \vee \underline{Nullable(\varepsilon+\Phi)}$$

$$\Longrightarrow [\underline{Nullable(\Phi+\Phi)} \wedge \underline{Nullable(s+t)}] \vee [\underline{Nullale(\varepsilon) \vee Nullable(\phi)}]$$

$$\Longrightarrow [\underline{Nullable(\Phi)} \vee \underline{Nullable(\Phi)}] \wedge [\underline{Nullable(s)} \vee \underline{Nullable(t)}] \vee [\underline{true \vee false}]$$

$$\Longrightarrow [\underline{(false \vee false)} \wedge \underline{(false \vee false)}] \vee [true]$$

$$\Longrightarrow [false \wedge false] \vee [true]$$

$$\Longrightarrow true$$

since the $Nullable[[(\Phi+\Phi).(s+t)] + (\varepsilon+\Phi)] = true$, we can conclude that regular expression $(p+q)(s+t)$ is also nullable.

4.5 mkeps function

We conducted nullability test on regular expression $(p+q)(s+t)$ in the previous Section 4.4 and we concluded that it is nullable. Therefore, now the algorithm calls mkeps function by passing derivative result (result obtained in Section 4.3) as an argument. $mkeps([[(\Phi + \Phi).(s + t)] + (\varepsilon + \Phi)]$ is calculated as follows:

$mkeps([[(\Phi + \Phi).(s + t)] + (\varepsilon + \Phi)]$
$\Longrightarrow\ Right\ (mkeps\ (\varepsilon + \Phi))$
$\Longrightarrow\ Right\ (Left\ (mkeps\ \varepsilon))$
$\Longrightarrow\ Right\ (Left\ ())$

The result of applying *mkeps* function on $[[(\Phi+\Phi).(s+t)]+(\varepsilon+\Phi)]$ (which is derivative result of $(p + q)(s + t)$) is $Right\ (Left\ ())$. This result serves as one of the input to *inject* function which is the next step in the algorithm.

4.6 Inject function

We already discussed in Section 4.1.3 that inject function induces back the characters in the input string into the partial values thus reverts back the construction of derivatives for a regular expression. From previous Section 4.5 we obtained a value which tells us how the derivative value of the regular expression $(p+q)(s+t)$ matches an empty string. The final value v_1 that tells how the string is being matched by the given regular expression is calculated as follows when the regular expression is $(p+q)(s+t)$ and the string ps:

$r = (p+q)(s+t)$
$r\backslash p = r_1 = [(\varepsilon + \Phi).(s+t)]$
$r_1\backslash s = r_2 = [(\Phi + \Phi).(s+t)] + (\varepsilon + \Phi)$
$mkeps\ (r_2) = Right\ (Left\ ()) = v_3$

- <u>1st iteration of inject function:</u>

  As we have already discussed in Section 4.1.3, during the first iteration of the inject function, the inputs to the function are r_1, s, v_3. Therefore,

  $inject\ \underline{[(\varepsilon + \Phi).(s+t)]}\ s\ Right\ (Left\ ())$

  $\Longrightarrow Seq\ \underline{(mkeps\ (\varepsilon + \Phi))}\ \underline{(inject\ (s+t)\ s\ Left\ ())}$

  $\Longrightarrow Seq\ \underline{(Left\ (mkeps\ \varepsilon))}\ \underline{Left\ (inject\ s\ s\ ())}$

  $\Longrightarrow Seq\ \underline{(Left\ (\ ()))}\ \underline{Left\ (Char\ s)}$

  $v_2 = Seq\ \underline{(Left\ (\ ()))}\ \underline{Left\ (Char\ s)}$

- <u>2nd iteration of inject function:</u>

  During second iteration, output of the 1^st iteration is one of the inputs *i.e.* v_2 along with character value p and r

  $inject\ \underline{[(p+q).(s+t)]}\ p\ \underline{Seq\ (Left\ (\ ()))}\ \underline{Left\ (Char\ s)}$

  $\Longrightarrow Seq\ \underline{(inject\ (p+q)\ p\ Left\ (\ ()\ ))}\ \underline{(Left\ (Char\ s))}$

  $\Longrightarrow Seq\ \underline{(Left\ (inject\ p\ p\ ()\ ))}\ \underline{(Left\ (Char\ s))}$

  $\Longrightarrow Seq\ \underline{(Left\ (Char\ p))}\ \underline{(Left\ (Char\ s))}$

$v_1 = Seq\ \underline{(Left\ (Char\ p))}\ \underline{(Left\ (Char\ s))}$

- Now, all the derivative results and characters in the input string word are consumed and there is nothing left to supply as input to inject function. Therefore, the output of inject function after completion of all the iterations is $Seq\ (Left\ (Char\ p))\ (Left\ (Char\ s))$. This final result of inject function stipulates how the given regular expression "$(p + q)(s + t)$" matches the input string "ps".

Chapter 5

Evaluation

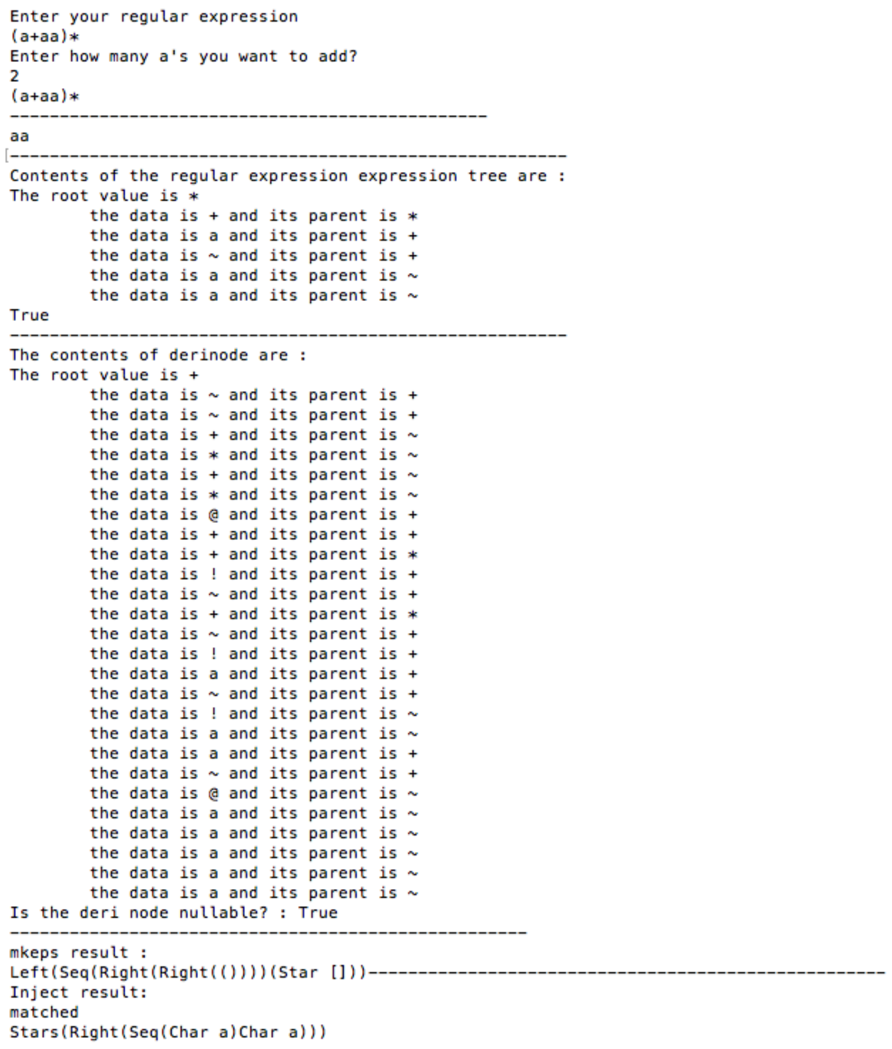
In the previous sections, we introduced the algorithmic details coined by Martin Sulzmann and Lu and also various functions which plays a pivotal role in the POSIX parsing of a regular expression. This section discusses about the critical evaluation of the code in terms of the time taken by the program to process various input regular expressions.

5.1 Results

When supplied a regular expression and a string to the program, it displays the output in the following way:

- First program asks the user to enter a regular expression.
- Next, depending upon the regular expression entered, it either forms its own string or asks the user to enter one.
- As discussed in section 4.2, the input regular expression is converted from string to its expression tree form. Program first prints out the contents of the tree following level-order traversal
- After tree contents, it says whether the input regular expression is nullable or not and then print the contents of the derivative result of the regular expression.
- At this point, it prints mkeps function result (if the input expression is nullable) follwed by inject function result displaying how the regular expression matches the input string.

The algorithm implemented in this project follows adapted POSIX policies to match a string against a regular expression. The program implemented indeed reporting a



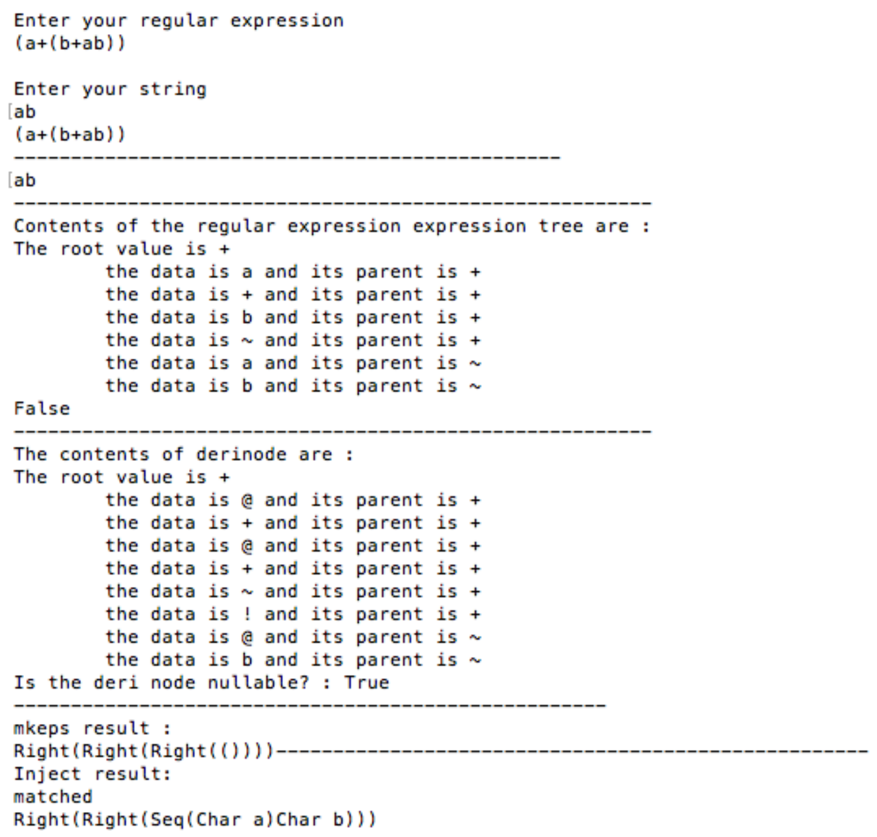
```
Enter your regular expression
(a+aa)*
Enter how many a's you want to add?
2
(a+aa)*
------------------------------------------------
aa
--------------------------------------------------------
Contents of the regular expression expression tree are :
The root value is *
        the data is + and its parent is *
        the data is a and its parent is +
        the data is ~ and its parent is +
        the data is a and its parent is ~
        the data is a and its parent is ~
True
--------------------------------------------------------
The contents of derinode are :
The root value is +
        the data is ~ and its parent is +
        the data is ~ and its parent is +
        the data is + and its parent is ~
        the data is * and its parent is ~
        the data is + and its parent is ~
        the data is * and its parent is ~
        the data is @ and its parent is +
        the data is + and its parent is +
        the data is + and its parent is *
        the data is ! and its parent is +
        the data is ~ and its parent is +
        the data is + and its parent is *
        the data is ~ and its parent is +
        the data is ! and its parent is +
        the data is a and its parent is +
        the data is ~ and its parent is +
        the data is ! and its parent is ~
        the data is a and its parent is ~
        the data is a and its parent is +
        the data is ~ and its parent is +
        the data is @ and its parent is ~
        the data is a and its parent is ~
        the data is a and its parent is ~
        the data is a and its parent is ~
        the data is a and its parent is ~
        the data is a and its parent is ~
Is the deri node nullable? : True
----------------------------------------------------
mkeps result :
Left(Seq(Right(Right(())))(Star []))----------------------------------------------------
Inject result:
matched
Stars(Right(Seq(Char a)Char a)))
```

FIGURE 5.1: Output displayed when expression is $(a + aa)^*$ and the string is aa

matching following POSIX policies. For example, if the input regular expression is $(a + aa)^*$ and the input string is aa, the following figure 5.1 is the output displayed by the program: another example output is given in the figure 5.2

The main concerns in the implementation of the algorithm is time taken by the program to process a regular expression. Experimental results show that the implementation works well exception of a few issues. The most important problem is encountered when input regular expression can match the input string in more than one way and the regular expression contains any quantified tokens like "*". Examples of regular expressions for which the implementation doesn't work as expected are $(a^*)^*$, $a?(n)a(n)$, $(a^*a^*)^*$, $(a^*a^*) + b$ and $(a + aa)^*$.

The following tables and graphs gives a detailed information about the time taken by the program to process the respective regular expression:



```
Enter your regular expression
(a+(b+ab))

Enter your string
[ab
(a+(b+ab))
------------------------------------------------
[ab
--------------------------------------------------------
Contents of the regular expression expression tree are :
The root value is +
        the data is a and its parent is +
        the data is + and its parent is +
        the data is b and its parent is +
        the data is ~ and its parent is +
        the data is a and its parent is ~
        the data is b and its parent is ~
False
--------------------------------------------------------
The contents of derinode are :
The root value is +
        the data is @ and its parent is +
        the data is + and its parent is +
        the data is @ and its parent is +
        the data is + and its parent is +
        the data is ~ and its parent is +
        the data is ! and its parent is +
        the data is @ and its parent is ~
        the data is b and its parent is ~
Is the deri node nullable? : True
----------------------------------------------------
mkeps result :
Right(Right(Right(())))------------------------------------------------------
Inject result:
matched
Right(Right(Seq(Char a)Char b)))
```

FIGURE 5.2: Output displayed when expression is $(a + (b + ab))$ and the string is ab

- $a?(n)a(n)$

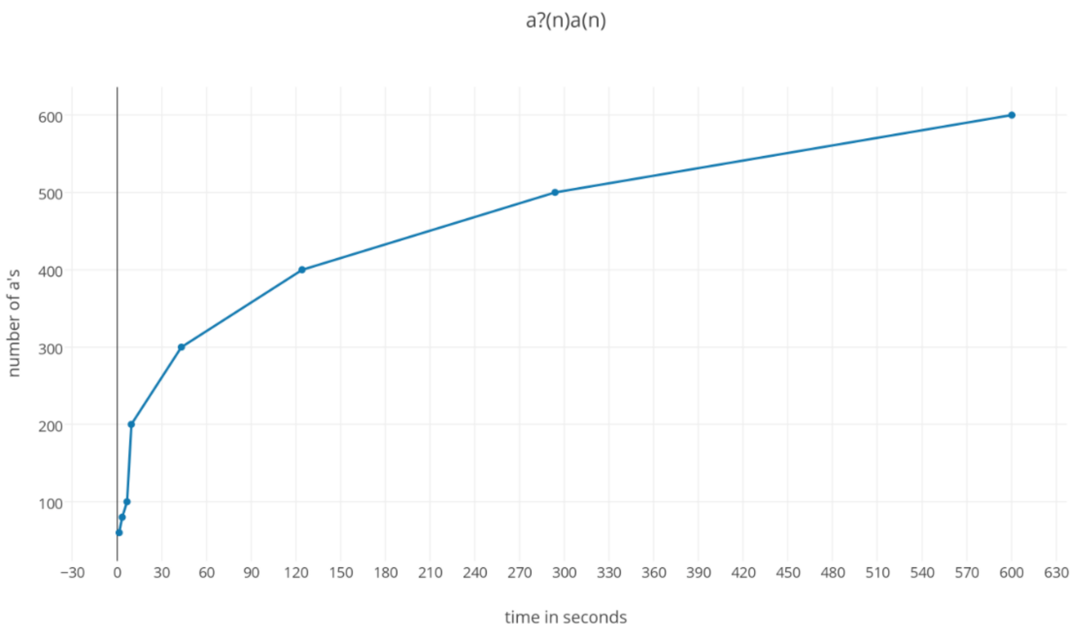


FIGURE 5.3: time plot of the program for the expression $a?(n)a(n)$

- $(a^*a^*) + b$

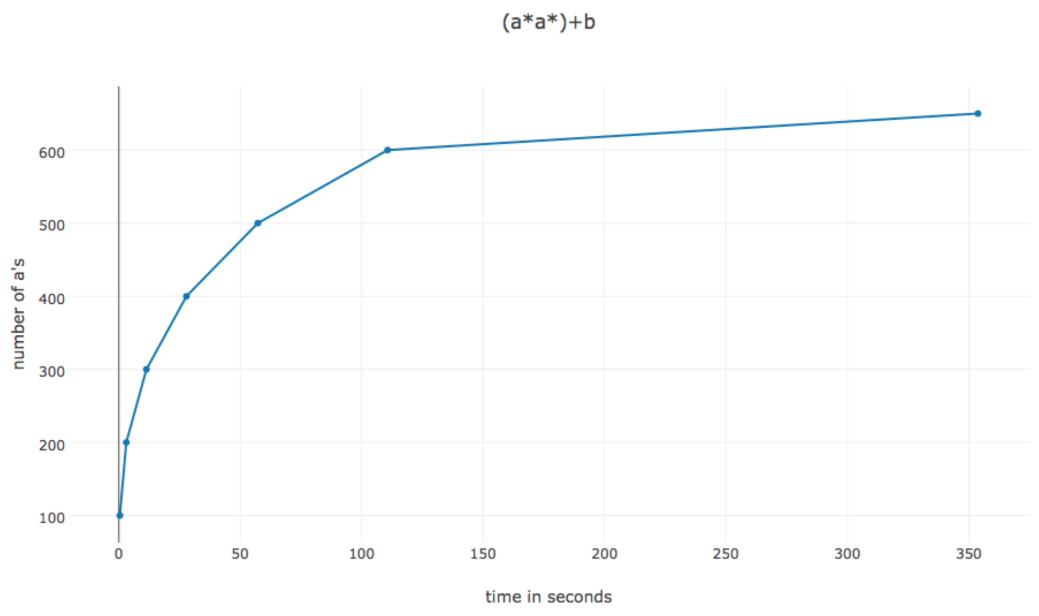


FIGURE 5.4: time plot of the program for the expression $(a^*a^*) + b$

- $(a^*)^*$

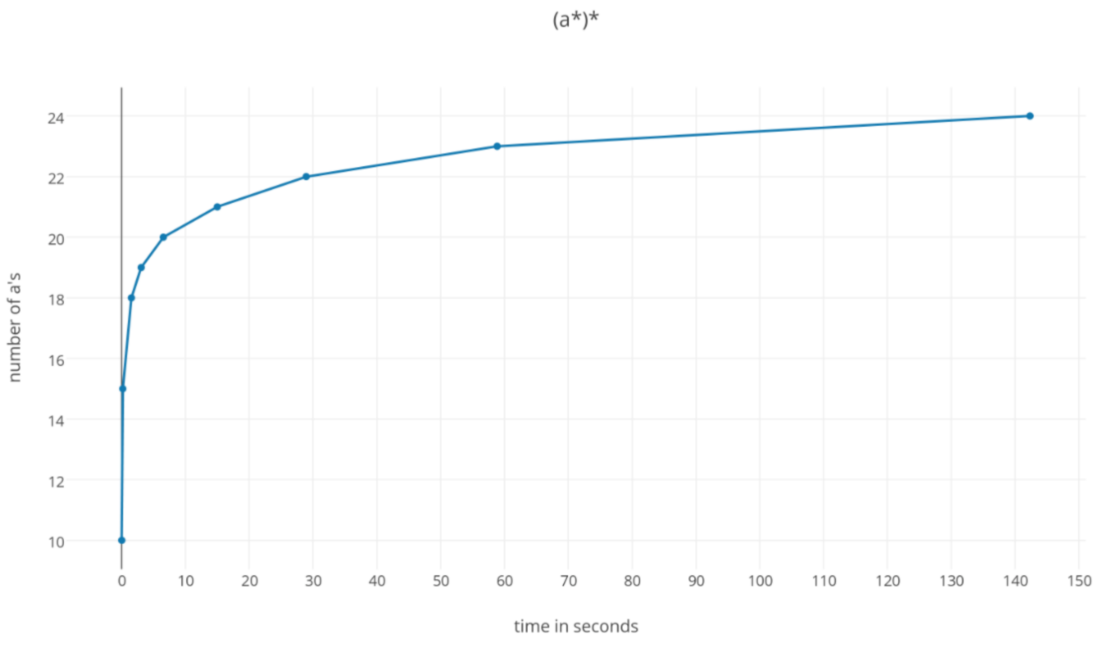


FIGURE 5.5: time plot of the program for the expression $(a^*)^*$

- $(a^*a^*)^*$

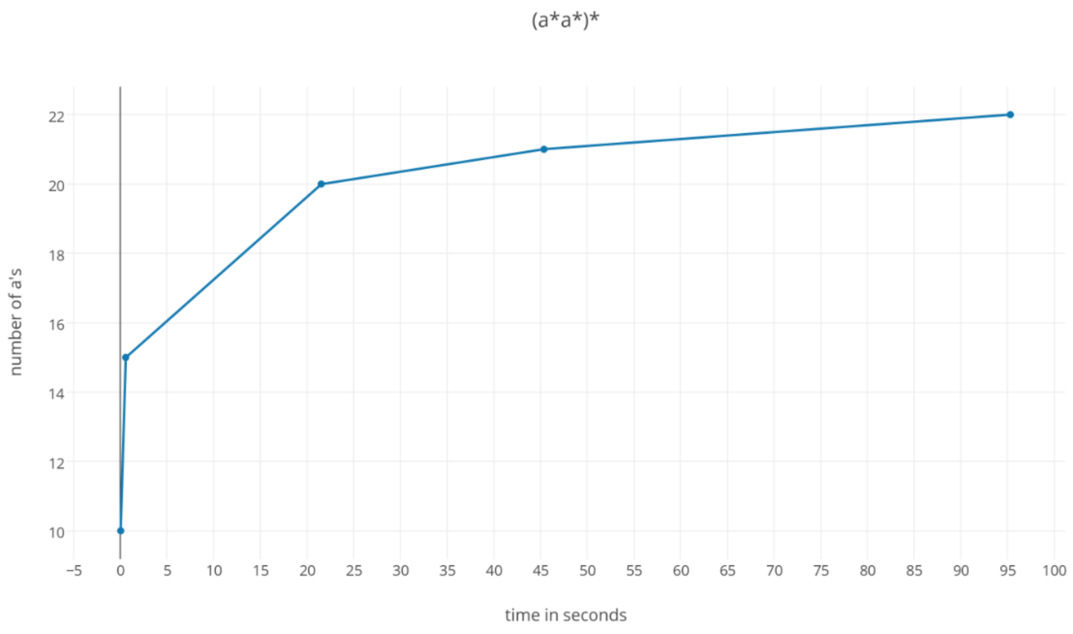


FIGURE 5.6: time plot of the program for the expression $(a^*a^*)^*$

- $(a + aa)^*$

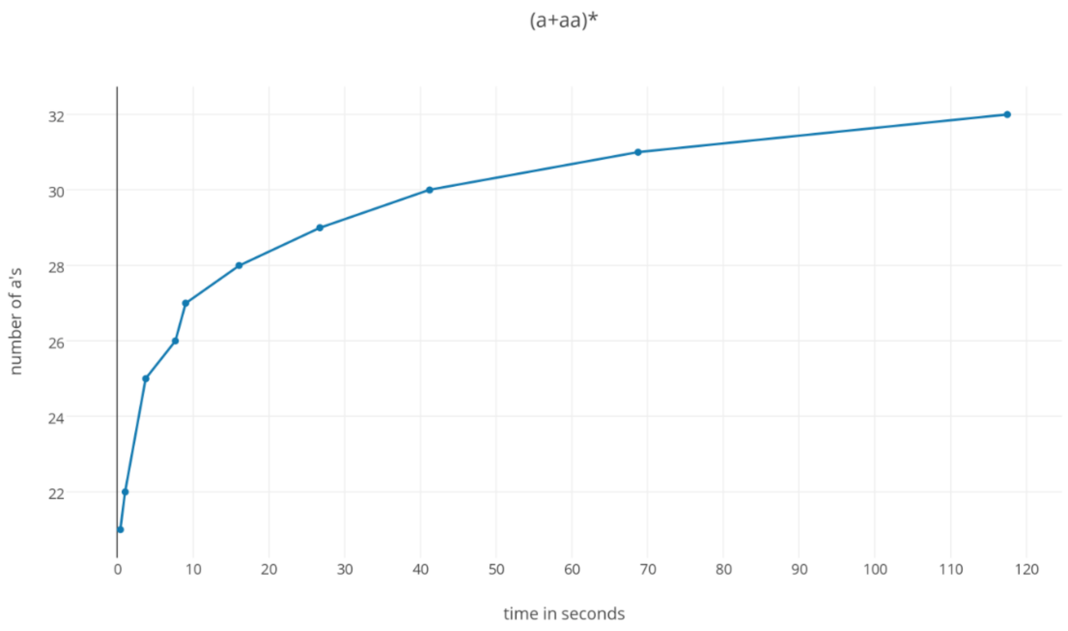


FIGURE 5.7: time plot of the program for the expression $(a + aa)^*$

In the above table each column corresponds to a regular expression and each row corresponds to the number of a's in a particular input string. In the table, S after a value in any row suggests the time value is in Seconds and M suggests that time value is in minutes.
From the graphs, we can observe that, as we increase the number of a's in the input string to be matched, the time taken by the program to decide whether the input regular expression matches the input string also increases. For every benchmark regular expression $(a^*)^*$, $a?(n)a(n)$, $(a^*\ a^*)^*$, $(a^*a^*) + b$ and $(a + aa)^*$ used in testing, there is some threshold value with respect to the length of the input string . If the number of a's in the input string is more than this threshold value, execution of the program is halted after considerable amount of time and "Garbage collector could not allocate 16384 bytes of memory for major heap section" error is displayed as shown in below figure 5.8.

.The threshold value of an input string with respect to a regular expression is detailed in the following table:

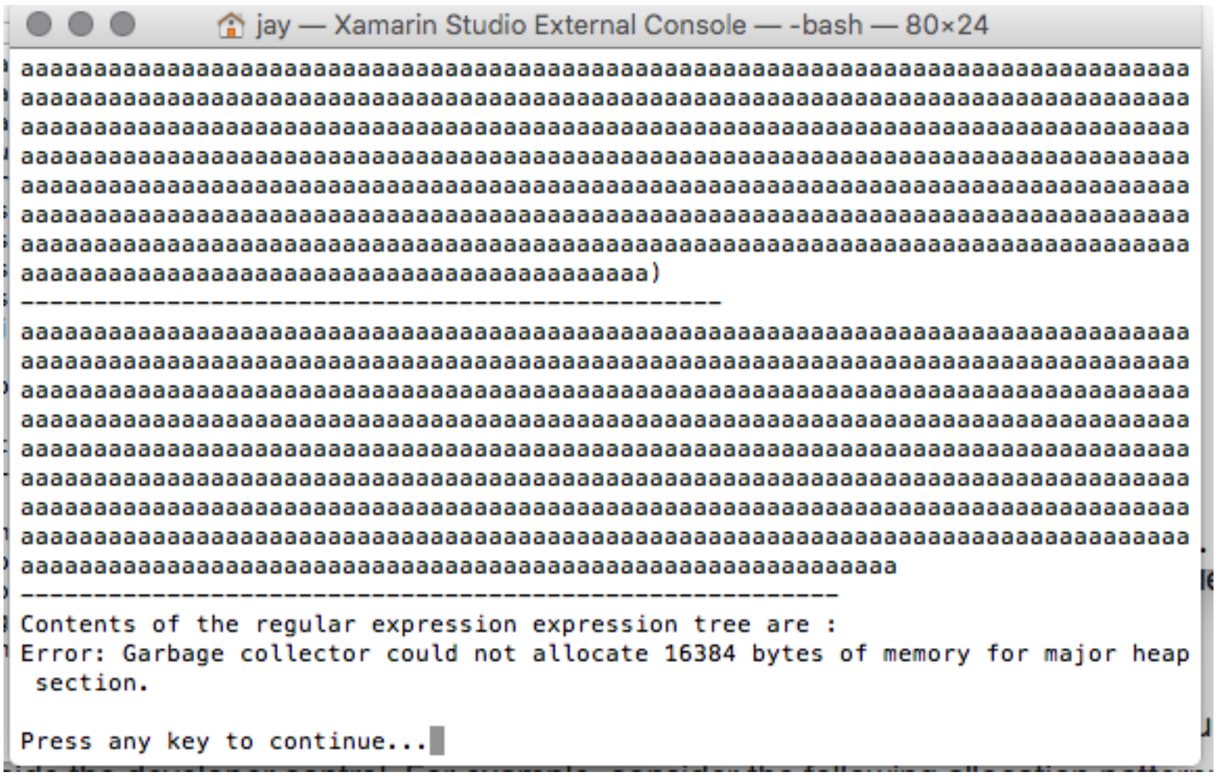
In conclusion, except in some extreme cases, algorithm produces POSIX based result and exhibits a better performance when compared to those algorithms mentioned in Motivation and Objectives section.

	a?(n)a(n)	(a*a*)+b	(a*)*	(a*a*)*	(a+aa)*
10			0.0416 S	0.0528 S	
15			0.1962 S	0.5805 S	
18			1.5464 S	4.5578 S	
19			3.0878 S	9.5178 S	
20			6.551 S	21.5207 S	
21			15.0062 S	45.3724 S	0.4124 S
22			28.9397 S	95.3162 S	1.0461 S
23			58.8584 S		1.0610 S
24			142.318 S		1.8509 S
25					3.7645 S
26					7.6645 S
27					9.0222 S
28					16.0755 S
29					26.7402 S
30					41.2295 S
31					68.7483 S
32					117.4921 S
60	1.5405 S				
80	3.5683 S				
100	6.7899 S	0.4128 S			
200	9.6568 S	3.0614 S			
300	43.2858 S	11.3551 S			
400	124.1882 S	27.8306 S			
500	4.898 M	57.2212 M			
600	10.0046 M	1.8443 M			
650	5.8975 M				

Table 5.1: Time taken by the program to process an input pair of a regular expression and an input string

Regular Expression	Threshold value
$a?(n)a(n)$	600
$(a^*a^*)+b$	650
$(a^*)^*$	24
$(a^*a^*)^*$	22
$(a+aa)^*$	32

Table 5.2: Threshold value table



```
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa)
------------------------------------------------
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
--------------------------------------------------------
Contents of the regular expression expression tree are :
Error: Garbage collector could not allocate 16384 bytes of memory for major heap
 section.

Press any key to continue...
```

Figure 5.8: Garbage collector error when number of a's in the input string is more than the threshold value

Chapter 6

Future Work and Conclusion

6.1 Future Work

Clearly there is a large scope for further work. When calculating derivatives using rules defined by Brozozowski [5], the resulting derivative regular expression are complex in nature which makes derivative-based regular expression matching a rather slow process [21]. Also, in some cases there will be a tremendous increase not only in the size of the derivatives but also the number of derivatives. For example, if $r = (a^*a^*)$ and the string $s = aa$, the following are the derivative steps

$$(a^*a^*)na \implies (a^*\ na).(a^*) + (a^*\ na) \implies (\varepsilon.a^*).a^* + (\varepsilon.a^*) = r_1$$

$(r_1na \implies [(\varepsilon.a^*).a^* + (\varepsilon\ .a^*)] \implies [[(\varepsilon.a^*).a^*]na + (\varepsilon\ .a^*)na]$
$\implies [[\Phi.a^* + \varepsilon.a^*].a^* + \varepsilon.a^*] + [\Phi.a^* + \varepsilon.a^*)$

If we observe the above example, with the advent of each a, the size and number of derivative is increasing and with sufficiently large number of a's in the input string, derivative calculation can explode. Therefore, there should be a simplification mechanism for the derivatives and it is calculated and explained in the paper [3]. These simplifications increase the speed with which algorithm computes and this can also be a solution to the problem arising when the length of the input string is more than a threshold value (explained in the results - Chapter 5.1).

6.2 Conclusion

To sum up, the report highlighted the aims, specifications and significance of the project. The report clearly explained all the essential concepts like Nullability of a regular expression, derivative of a regular expression, roles of functions like *mkeps* function and *inject* function to make the POSIX parsing of regular expression algorithm more understandable.

Report also contains an example explaining all the steps of the algorithm in detail. Finally, an evaluation the program implemented is done and is given in the report.

At the end of this report, it can be concluded that the algorithm for POSIX parsing of regular expression using derivatives is indeed a sophisticated approach which adheres to the POSIX policies and also the time complexity of the algorithm in some cases is very much promising. However, algorithm still has some difficulty in processing input strings of length more than some threshold value (explained in Results Section 5.1). we can conclude that one of the solutions to improve the algorithmic complexity is by simplifying derivative function. Therefore, algorithm is implemented only partially since there are some issues which can be dealt with accordingly and those issues can be fixed.

Bibliography

[1] Regex-buddy. Posix basic and extended regular expressions, 2016. URL http://www.regular-expressions.info/posix.html. [Online; accessed 25-August-2016].

[2] Scott Owens, John Reppy, and Aaron Turon. Regular-expression derivatives re-examined. *J. Funct. Program.*, 19(2):173–190, March 2009. ISSN 0956-7968. doi: 10.1017/S0956796808007090. URL http://dx.doi.org/10.1017/S0956796808007090.

[3] Martin Sulzmann and Kenny Zhuo Ming Lu. POSIX regular expression parsing with derivatives. In *International Symposium on Functional and Logic Programming*, pages 203–220. Springer, 2014.

[4] Wikipedia. Redos malicious regexes, 2010. URL https://en.wikipedia.org/wiki/ReDoS. [Accessed: 20-August-2016].

[5] Janusz A Brzozowski. Derivatives of regular expressions. *Journal of the ACM (JACM)*, 11(4):481–494, 1964.

[6] C Steven. Kleene. representation of events in nerve nets. *Automata Studies*, pages 3–40, 1956.

[7] Jay Earley. An efficient context-free parsing algorithm. *Communications of the ACM*, 13(2):94–102, 1970.

[8] Niels Bjørn Bugge Grathwohl, Fritz Henglein, and Ulrik Terp Rasmussen. A crash-course in regular expression parsing and regular expressions as types.

[9] Larry Wall, Tom Christiansen, and Jon Orwant. *Programming perl.* " O'Reilly Media, Inc.", 2000.

[10] Philip Hazel. Pcre–perl compatible regular expressions. *Online http://www. pcre. org*, 2005.

[11] Institute of Electrical and Electronics Engineers (IEEE): Standard for Information technology. Portable operating system interface (posix) – part 2 (shell and

utilities. *Section 2.8 (Regular Expression notation), New York, IEEE Standard 1003.2 (1992)*, 2005.

[12] Stijn Vansummeren. Type inference for unique pattern matching. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 28(3):389–428, 2006.

[13] Regular-Expressions.info. Portable operating system interface (posix). *http://www.regular-expressions.info/posix.html*, 2016.

[14] Regex Posix. Posix policies. *https://wiki.haskell.org/Regexposix*, 2016.

[15] Wikipedia. Patterns for non-regular languages, 2010. URL https://en.wikipedia.org/wiki/Regular_expression#Patterns_for_non-regular_languages\OT1\textendashunderImplementationsandrunningtimes. [Accessed: 20-August-2016].

[16] Wikipedia. Backtracking — wikipedia, the free encyclopedia, 2016. URL https://en.wikipedia.org/w/index.php?title=Backtracking&oldid=719767143. [Online; accessed 25-August-2016].

[17] Nicolaas Weideman, Brink van der Merwe, Martin Berglund, and Bruce Watson. Analyzing matching time behavior of backtracking regular expression matchers by using ambiguity of nfa. In *International Conference on Implementation and Application of Automata*, pages 322–334. Springer, 2016.

[18] Marius Schulz. blog. *https://blog.mariusschulz.com/2014/06/03/why-using-in-regular-expressions-is-almost-never-what-you-actually-want*, 2016.

[19] Regex-buddy. Runaway regular expressions: Catastrophic backtracking, 2016. URL http://www.regular-expressions.info/catastrophic.html. [Online; accessed 25-August-2016].

[20] Wikipedia. Mono develop. *https://en.wikipedia.org/wiki/MonoDevelop*, 2016.

[21] Fahad Ausaf, Roy Dyckhoff, and Christian Urban. *POSIX Lexing with Derivatives of Regular Expressions (Proof Pearl).* 8 2016.

[22] Code Idol. Operator precedence. *http://codeidol.com/community/perl/know-the-precedence-of-regular-expression-operator/14215/*, 2016.