CHALMERS

A Generator of Incremental Divide-and-Conquer Lexers

A Tool to Generate an Incremental Lexer from a Lexical Specification

Master of Science Thesis [in the Programme MPALG]

JONAS HUGO

KRISTOFER HANSSON

Chalmers University of Technology
Department of Computer Science and Engineering
Göteborg, Sweden, October 2013

A Generator of Incremental Divide-and-Conquer Lexers
A Tool to Generate an Incremental Lexer from a Lexical Specification
JONAS HUGO,
KRISTOFER HANSSON,

© JONAS HUGO, October 2013.
© KRISTOFER HANSSON, October 2013.

Examiner: BENGT NORDSTRÖM

Chalmers University of Technology
University of Gothenburg
Department of Computer Science and Engineering
SE-412 96 Göteborg
Sweden
Telephone + 46 (0)31-772 1000

Cover:
The image is a trol image, and will not be used in the final report.

Department of Computer Science and Engineering
Göteborg, Sweden October 2013

Abstract

A text that is a crash.course of the project. What will the report talk about, what obsticals had to be conquerd. talk talk talk.

Acknowledgements

We would like to take the chance of thanking our supervisor at department of computer science, Jean-Philippe Bernardy. Also thank our parents, and last but not least. We like to thank our self!

Jonas Hugo & Kristofer Hansson, Göteborg October 2013

Contents

1 Introduction

This master-thesis is carried out at Chalmers, on the department of computer science.

1.1 Background

Editors normally have regular-expression based parsers, which are efficient and robust, but lack in precision: they are unable to recognize complex structures. Parsers used in compilers are precise, but typically not robust: they fail to recover after an error. They are also not efficient for editing purposes, because they have to parse files from the beginning, even if the user makes incremental changes to the input. More modern IDEs use compilerstrength parsers, but they give delayed feedback to the user. Building a parser with good characteristics is challenging: no system offers such a combination of properties

1.2 Scope of work

Dan Piponi has writen a blogpost on how to determend in a incremental way if a string furfills a regular expression. This is done by using Monoids, Fingertrees and tabulate functions. [8]

This blogpost is the fundamental idea behind the project. To build one same general idea, but instead build a tool that generates a lexical analyser given a bnf file specification of a language. Where the core algorithm in the tool follows the blogposts idea. The project will use Alex [5] as mutch as possible, that is this algorithm will be used as a wrapper to the Alex lexing tool.

The coal of the project is to create an algorithm that can do a lexical analysis on a update to an already lexed code with a sufficent fast time cost. With a sufficent fast time means that the lexical analysier can be runed in real time.

The report will start by give a more general knowledge about lexical analysis. Then start to give a overviewing image of what is needed of the algorithm to work correctly. Which building blocks needed to create the algorithm. This will lead up to the implementation of the algorithm and specific recvierments on the algorithm for it to be fully correct. The report will also describe how the testing has been done. That is test for correctness, rubustness and efficiency. Also pressent the result for the benchmarking on different computer systemts. The last part of the report will give a more fomal preformance of the algorithm, discussion of the result, some conclussions and futher work.

2

Lexer

A lexer, lexical analyser, is a pattern matcher. Its job is to find sequence of characters in a larger string. The Lexer is a front end of a syntax analyser [10]. This can be done by using regular expressions, regular sets and finite automata. Which are central concepts in formal language theory [1]. All on which will be described in this chapter.

2.1 Lexing vs Parsing

Lexers work as a subprogram to the parser, giving it's result to the syntax analyser. There are several reasons why a compiler should be separated in to a lexical analyser and a parser (syntax analyser). Simplicity of design is the most important reason. When dividing the task in to these two sub tasks, it allows the system to simplify each of these sub-tasks. For example, a parser that has to deal with white-spaces and comments would be more complex than one that can assume these have already been removed by an lexer. Also when the two tasks have been seperated into sub-tasks it can lead to cleaner overall design when designing a new language. The only thing the syntax analyser will see is the output from the lexer, tokens and lexemes which will be described later in this chapter. The lexer also skips comments and white-spaces, since these are not relevant for the syntax analyser. Further overall efficiency of the compiler can be improved. When separating the lexical analyser it allows for use of specialised techniques that can be used only in the lexical task. Last, compiler portability can be enhanced. That is Input-device-specific peculiarities can be restricted to the lexical analysis [2]. So the lexer can detect syntactical errors in tokens, such as ill-formed floating-points literals, and report these errors to the user [10]. Breaking the compilation before running the syntax analyser, saving computing time.

2.2 Token Specification

The job of lexical analyser is to translate a human readable string to an abstract computer-readable list of tokens. How these abstract data-types can be obtained from a string the lexer uses different techniques. This section describe the techniques used when writing rules for the tokens patterns.

2.2.1 Regular Expressions

Example 2.2.1 (Valid C Idents [2]). Say that we want to express the set of valid C identifiers. Using regular expressions makes it very easy. Given a sequence of elements, where the elements are one character. Say we have a element $letter \in \{\text{a}\ldots\text{z}\} \cup \{\text{A}\ldots\text{Z}\} \cup \{\_\}$ and another element $digit \in \{0\ldots 9\}$ Then with help of regular expressions the definition of all valid C identifiers would look like this: $letter(letter|digit)*$.

Definition 2.2.2 (Regular Expressions [1]).

1. The following characters are meta characters $\{'|', '(', ')', '*'\}$.
2. A none meta character a is a regular expression that matches the string a.
3. If r_1 and r_2 are regular expressions then $(r_1|r_2)$ is a regular expression that matches any string that matches r_1 or r_2.
4. If r_1 and r_2 are regular expressions. $(r_1)(r_2)$ is a regular expression that matches the string xy iff x matches r_1 and y matches r_2.
5. If r is a regular expression $r*$ is a regular expression that matches any string of the form $x_1, x_2, \ldots, x_n, n \geq 0$; where r matches x_i for $1 \leq i \leq n$, in particular $(r)*$ matches the empty string, ε.
6. If r is a regular expression, then (r) is a regular expression that matches the same string as r.

Many parentheses can be reduced by adopting the convention that the Kleene closure operator $*$ has the highest precedence, then concat $(r_1)(r_2)$ and then or operator $|$. The two binary operators, concat and or are left-associative.

2.2.2 Languages

An alphabet is an finite set of symbols, for example Unicode. Which includes approximately 100,000 characters. A language is any countable set of some fixed alphabet [2]. The term "formal languages" refers to languages which can be described by a body of systematic rules. There is a subset of languages to formal languages called regular language, these regular languages refers to those languages that can be defined by regular expressions [9].

2.2.3 Regular Definitions

When defining a language it is useful to give the regular expressions names, so they can for example be used in other regular expressions. These names for the regular expressions are them self symbols. If Σ is an alphabet of basic symbols, then a regular definition is a sequence of definitions of the form:

$$\begin{array}{ccc} d_1 & \rightarrow & r_1 \\ d_2 & \rightarrow & r_2 \\ \vdots & \rightarrow & \vdots \\ d_n & \rightarrow & r_n \end{array}$$

where:

1. Each d_i is a new symbol, not in Σ and not the same as any other of the d's.
2. Each r_i is a regular expression over the alphabet $\Sigma \cup \{d_1, d_2 \ldots d_{i-1}\}$

By restricting r_i to Σ and previously defined d's the regular definitions avoid recursive definitions [2].

2.2.4 Longest Match

A lexer should always prefer the longest sequence matching a regular expression. [2]
Example 2.2.3 (Longest Match).
Given the regular expressions: $a * b+$
and the input sequence: ”aaaabbbbbbbbbbb”
valid outputs are:

1	”b”
⋮	⋮
11	”bbbbbbbbbbb”
12	”ab”
⋮	⋮
22	”abbbbbbbbbbb”
⋮	⋮
44	”aaaabbbbbbbbbbb”

Even if there are 44 different valid results from this given input, only the longest sequence should be treated as correct by the lexer. In this example, result number 44: ”aaaabbbbbbbbbbb”.

2.3 Tokens, Patterns and Lexemes

When rules have been defined for a language, the lexer needs structures to represent rules and the result from lexing the code-string. This section describe the structures which the lexical analyser use for representing the abstract data; what these structures are for and what is forwarded to the syntactical analyser.

A lexical analyser uses three different concepts. All which is described below.

Token is a pair consisting of a token name and an optional attribute value. The token name is an abstract symbol corresponding to a lexical unit [2]. For example, a particular keyword, data-type or identifier.

Pattern is a description of what form a lexeme may take [2]. For example, a keyword is just the sequence of characters that forms the keyword, an int is just a sequence consisting of just numbers. Can be described by a regular expression.

Lexemes is a sequence of characters in the code being analysed which matches the pattern for a token and is identified by the lexical analyser as an instance of a token [2].

As mentioned before a token consists of a token name and an optional attribute value. This attribute is used when one lexeme can match more then one pattern. For example the pattern for a digit token matches both 0 and 1, but it is important for the code generator to know which lexeme was found. Therefore the lexer often returns not just the token but also an attribute value that describes the lexeme found in the source program corresponding to this token [2]. A lexer collects chars into logical groups and assigns internal codes to these groups. according to their structure [10]. Where the groups of chars are lexemes and the internal codes are tokens. A example follows how a small piece of code would be divided.
Example 2.3.1 (Logical grouping [10])**.**
This is the code being lexed:

```
result = oldsum - value /100;
```

Given some basic tokens patterns:

letter	::	$\{\text{a}\ldots\text{z}\} \cup \{\text{A}\ldots\text{Z}\} \cup \{\_\}$
digit	::	$\{0\ldots9\}$
IDENT	::	$letter(letter \mid digit)*$
INT_LIT	::	$digit+$
ASSING_OP	::	=
SUB_OP	::	-
DIV_OP	::	/
SEMICOLON	::	;

This is how it will be divided:

Token	Lexeme
IDENT	result
ASSING_OP	=
IDENT	oldsum
SUB_OP	−
IDENT	value
DIV_OP	/
INT_LIT	100
SEMICOLON	;

2.4 Recognition of Tokens

In previous section the topic have been, how to represent a pattern using regular expressions and how these expressions relates to tokens. This section will highlight how to transform a sequence of characters into a sequence of abstract tokens. First giving some basic understanding with transition diagrams.

2.4.1 Transition Diagrams

A state transition diagram, or just transition diagram is a directed graph, where the nodes are labelled with state names. Each node represents a state which could occur during the process of scanning the input, looking for a lexeme that matches one of several patterns [2]. The edges are labelled with the input characters that causes transitions among the states. An edge may also contain actions the lexer must perform when transition is token [10]. Some properties for a transition diagram follows. One state is said to be initial state. The transition diagram always begins at this state, before any input symbols have been read. Some states are said to be accepting (final). They indicate that a lexeme has been found. The found token should then be returned with any additional optional values, mentioned in previous section.[2] Transition diagrams of the formed used in lexers are representations of a class of mathematical machines called finite automata. Finite automata can be designed to recognise members of a class of languages called regular languages, mentioned above [10].

2.4.2 Finite Automata

A finite automaton is essentially a graph, like transitions diagrams, with some differences:

- Finite automata are recognizers; they simply say ”YES” or ”NO” about each possible input string.
- Finite automata comes in two different forms:

  Non-deterministic Finite Automata (NFA) which have no restriction of the edges, several edges can be labelled by the same symbol out from the same state. Further ϵ, the empty string, is a possible label.

  Deterministic Finite Automata (DFA) for each state and for each symbol of its input alphabet exactly one edge with that symbol leaving that state. The empty string ϵ is not a valid label.

Both these forms of finite automata are capable of recognising the same subset of languages, all regular languages [2]. The formal definition of a finite automaton follows:
Definition 2.4.1 (Finite Automata [11])**.** A finite automata is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$, where

1. Q is a finite set called the states,
2. Σ is a finite set called alphabet,
3. $\delta : Q \times \Sigma \to Q$ is a transition function,
4. $q_0 \in Q$ is the start state, and
5. $F \subseteq Q$ is the set of accept states.

Non-deterministic Finite Automata

An NFA accepts the input; x if and only if there is a path in the transition diagram from the start state to one of the accepting states, such that the symbols along the way spells out x [2].

There are two different ways of representing an NFA which this report will describe. One is by transition diagrams, where the regular expression will be represented by a graph structure. Another is by transitions table, where the regular expression will be converted in to a table of states and the transitions for these states given the input. The following examples shows how the transition diagram and transition table representation will look like for a given regular expression.
Example 2.4.2 (RegExp to Transition Diagram [2])**.** Given this regular expression: $(a|b) * abb$
the transition diagram in fig. 2.1 representing this regular expression.
Example 2.4.3 (RegExp to Transition Table [2])**.** Given the regular expression from example 2.4.2 it can be converted into the transition table shown in fig. 2.2

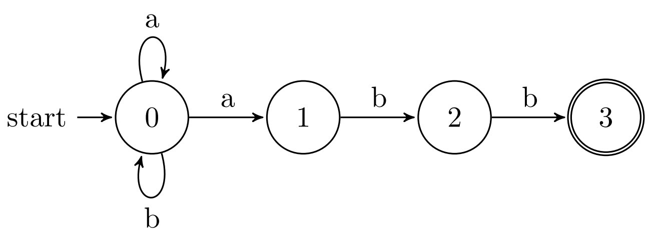


Figure 2.1: Transition Diagram, accepting the pattern $(a|b)*abb$

State	a	b	ϵ
0	$\{0,1\}$	$\{0\}$	$\emptyset$
1	$\emptyset$	$\{2\}$	$\emptyset$
2	$\emptyset$	$\{3\}$	$\emptyset$
3	$\emptyset$	$\emptyset$	$\emptyset$

Figure 2.2: Transition Table representation of regular expression in example 2.4.2

Transition tables have the advantage that they have an quick lookup time. But instead it will take a lot of data space, when the alphabet is large. Most states do not have any move on most of the input symbols [2].

Deterministic Finite Automata

DFA is a special case of an NFA where,

1. there are no moves on input ϵ and
2. for each state s and input symbol a, there is exactly one edge out of s labelled with a.

While NFA is an abstract representation of an algorithm to recognise the string of a language, the DFA is a simple concrete algorithm for recognising strings. Every regular expression can be converted in to a NFA and every NFA can be converted in to a DFA and then converted back to a regular expression [2]. It is the DFA that is implemented and used when building lexical analysers.

Example 2.4.4 (DFA representation of RegExp [2])**.** A DFA representation of same regular expression from example 2.4.2 is shown in fig. 2.3

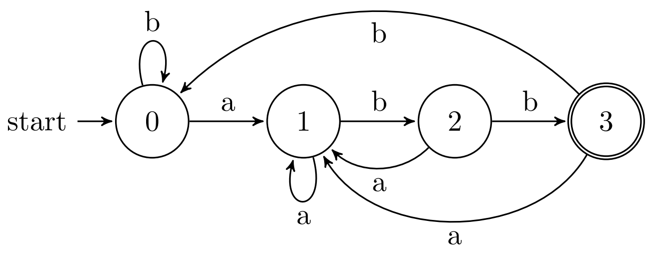

$$\begin{tikzcd}
\text{start} \arrow[r] & 0 \arrow[r, "a"] \arrow[loop above, "b"] & 1 \arrow[r, "b"] \arrow[loop below, "a"] & 2 \arrow[r, "b"] \arrow[l, bend left, "a"] & \boxed{3} \arrow[lll, bend right, "b"'] \arrow[ll, bend left, "a"]
\end{tikzcd}$$

Figure 2.3: DFA, accepting the regular expression: $(a|b) * abb$

3

Divide-and-Conquer Lexer

An incremental divide and conquer lexer works by dividing the sequence, to be lexicaly analysed, into smallest part and analyse them; and then combining them. In the base case the lexical analysis is done on a single character. The conquer step is then to combine the smaller tokens into as large tokens as possible. The end result should be a sequence of token that represent the code. How this is done will be described below.

3.1 Divide and Conquer in General

This capter will try to give a idea of how the Divide and Conquer algorithm works in a general case. So when talking about the incremental lexer you as a reader will have a basic idea of the structure of the algorithm.

3.1.1 The Basics

The general idea of a divide and conquer is to divide a problem into to smaler parts and solv them separatly and then combine the results. A Divide and Conquer algorithm always consist of a pattern with three steps. [6]

Divide: That is if the input size is bigger than the basecase divide the input to sub problems. Otherwise solve the problem using a straightforward method.

Recur: Recursively solve the subproblems associated with the subset.

Conquer: Given the solutions to the subproblem, merge them into to the original problem.

3.1.2 Associative Function

The associative property stated for when an expression containing two or more of the same operator, or function, in a row. The order of which of the operations that are executed first has no impact on the result.

In divide and conquer this is essential. On the conquer step in the divide and conquer algorithm, there is no certain order of which how the subproblems are going to be conquered. So the result of running any part with another part must give the same result as doing it in any other way.

3.1.3 Time Complexity

To calculate the running time of an divide and conquer algorithm the master method can be applied. [4] This method is based on the following theorem.
Theorem 3.1.1 (Master Theorem [4]). *Assume a function*
T_n *constrained by the recurrence*

$$T_n = aT_{\frac{n}{b}} + f(n)$$

This is typically the equation of a divide-and-conquer algorithm, where a is the number of sub-problems at each recursive step, n/b is the size of each sub-problem, and f(n) is the running time of dividing up the problem space into a parts, and combining the sub-results together.
If we let $e = log_b a$, *then*

$$\begin{aligned}
&1.\ Tn &=& \ \Theta(n^e) && \text{if } f(n) = O(n^{e-\epsilon} \text{ and } \epsilon > 0\\
&2.\ Tn &=& \ \Theta(n^e logn) && \text{if } f(n) = \Theta(n^e)\\
&3.\ Tn &=& \ \Theta(f(n)) && \text{if } f(n) = \Omega(n^{e+\epsilon}) \text{ and } \epsilon > 0 \text{ and } a \cdot f(n/b) \leq c \cdot f(n)\\
& & & && \text{where } c < 1 \text{ and all sufficiently large } n
\end{aligned}$$

■

3.1.4 Hand on Example

The divide and conquer pattern can be preformed on different sorts of algorithm that solves different problems. A general problem is sorting or more precies sorting a list of integers. This example will show merge-sort.

1. The algorthim starts by the divide step. Givien the input S the algorithm will check if S consist of a number of elements less or equal to 1.
   - If this is true, return the sequence. A sequence of one or zero elements is always sorted.

  - If this is false, split the sequence into two (as close as possible) equaly big sequences. S_1 and S_2, S_1 is equal S.getRange(1, $n/2$). S_2 is equal S.getRange(($n/2$)+ 1, n). The indexing of the sequence starts at 1.

2. The next step is to sort the subsequences S_1 and S_2. This is done by recursively call it self with the subsequences as input. There will be 2 calls, one with S_1 as input and one with with S_2 as input.

3. The to sequences S_1 and S_2 is has now be recursively sorted and can be merged. Assign S to the result of the merging.

Algortihm 1 shows a more formal definition of merge-sort.

Algorithm 1: MergeSort

```
Data: Sequence of integers $S$ containing $n$ integers
Result: Sorted sequence $S$
1 if $length(S) \leq 1$ then
2     return $S$
3 else
4     $(S_1,S_2) \leftarrow splitAt(S,n/2)$
5     $S_1 \leftarrow MergeSort(S_1)$
6     $S_2 \leftarrow MergeSort(S_2)$
7     $S \leftarrow Merge(S_1, S_2)$
8     return $S$
```

Given the mergesort algorithm, this is how the time complexity is calculated. There are 2 recurrsive calls, the subproblems are 1/2 of the original problem size. To merge the two sorted subproblems the worst case is to check every element in the two list, $2 \cdot n/2 = n$.

$$T(n) = 2T(n/2) + n.$$

Results in:

$$a = 2,\ b = 2 \text{ and } f(n) = n\ \Rightarrow n^{log_b a} = n^{log_2 2} = n$$

Case 2 applies, since

$$f(n) = O(n)$$

So the solution will be:

$$T(n) = \Theta(n^{log_2 2} \cdot logn) = \Theta(n \cdot logn)$$

3.2 Divide and Conquer Lexing in General

This section is for give a more general image of the ideas of the incremental divide and conquer lexer. To give the reader a more overviewing image why this is a good idea before starting to go into deep details about the project.

3.2.1 Treestructure

For a good structuer where the code-lexemes can be related to its tokens, current result can be saved and easy recalculated. A divide and conquer lexer should use a tree structure to save the lexed result in. Since every problem can be divided in to several subproblems, until the basecase is reached. This is cleraly a tree structure of solutions, where the leafs are the tokens for a single character. and the root is a sequence of all tokens in the code.

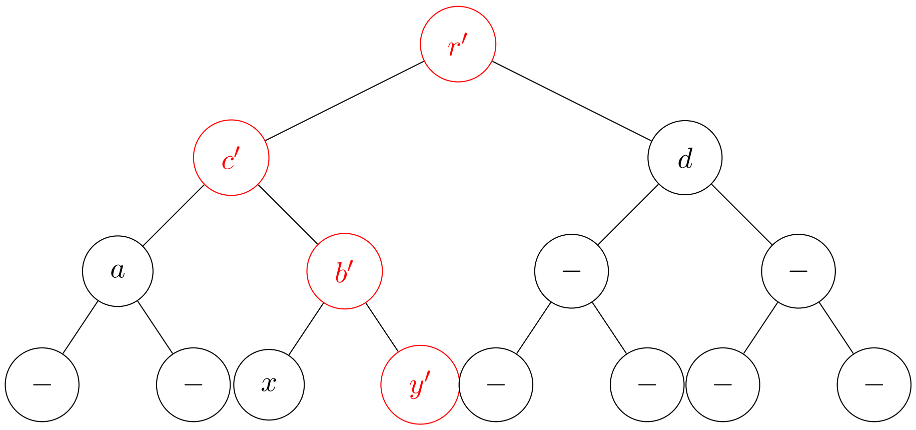


Figure 3.1: Incremental Computing, the updated nodes when a leaf changes

3.2.2 Incremental Computing

To be incremental means that, whenever some part of the data to the algorithm changes the algorithm tries to save time by only recomputing the changed data and the parts that depend on this changed data. [12]

For a divide and conquer lexer this would mean only recompute the changed token and the token to the right of the changed token. This is done recurrsivly until the root of the tree is reached. The expected result of this would be that when a character is added to the code of 1024 tokens, instead of relex all 1024 tokens the lexer will only do 10 recalculations for new tokens. Since, $log_2 1024 = 10$. This can be explained by the theorem 3.1.1.

Since incremental computing stated that only content which depends on the new data will be recalculated. That is, follow the branch of the tree from the new leaf to the root and recalculated every node on this path. As shown by fig. 3.1. Only one subproblem is updated in every level of the tree. Back to the master theorem. Let put this in to numbers, $e = log_b a$ where a is number of recursive calls and n/b is size of the subproblem where n is the size of the original problem. As shown by the fig. 3.1 number of needed update calls is 1, therefor $a = 1$. The constant b is still 2. This will give $e = log_2 1 = 0$. Thus the update function of the incremental algorithm will have a time complexity of $\Theta(n^0 \cdot logn) = \Theta(logn)$

3.2.3 Basecase

The basecase is the token for a single character. Since a single character always can be part of the code, special characters can be in a comment. The basecase should always return a valid result. The result is saved with a transistions map, which is described futher down in this chapter.

3.2.4 Conquer Step

When conquering two sequences tokens in to one sequence of tokens the transitions are checked. To see if there is a way of combining the the right most token in the left sequence and the left most token in the right sequence.

In general the conquer step in a divide and conquer lexer needs to follow the following rules:

- If two tokens can be combined, combine the result with the next token.
- All tokens must start in the starting state (exception are single character tokens).
- If two tokens can't be completly combined, combine the first token with as much as possible of the second token and check that the combination ends in an accepting state.

3.2.5 Expected Time Complexity

Given the information from theorem 3.1.1 and the case of an incremental divide and conquer algorithm (where only one side of the subproblem will be calculated). The number of subproblems will only be 1. The size of the subproblem will be $n/2$, gives $a = 1$, $b = 2$ and $e = log_2 1 = 0$. SOMEHOW THE $f(n)$ FUNCTION WILL GIVE A c THAT IS 0. HELP. Given that $e = c$ the time-complexity for the incremental algorithm will follow the equation $n^c log^{d+1} n$. After inserting the known constants the expected time-complexity for an incremental divide and conquer algorithm should be, $O(log^{d+1}n)$.

3.3 Transition map

To be able to make a divide-and-conquer lexer work you can use a transition map for the subexpression that has currently been lexed. That is you store for each subexpression a list of tuples which are built up of an in state, a list of tokens and an out state. This could in haskell look something like this.

```
type Transition = (State,Tokens,State)
type Transitions = [Transition]
```

The *tokens* type helps to think of as being a sufix, that is the first part of the substring, this has to end in an accepting state; a list of tokens which all are complete tokens, they start in the starting state and end in an accepting state; and a prefix, the last part of the lexed substring, this part has to begin in the starting state; or just a single partial token that can be anywhere in a token.

```
data Tokens = Single String
            | Multiple String [Token] String
```

Now lets see how these data types work in order to create list of tokens for a String.

3.3.1 The Base Case

When the lexer tries to lex one character it will create the above transition table using the DFA for the language. It will for each state in the DFA lookup what out state the character would yield and create a Tokens type of Single. For the character '/' part of a transition map might look like the following.

$$\begin{bmatrix} 10 & Single'/' & 10 \\ 11 & Single'/' & NoState \\ 12 & Single'/' & 10 \end{bmatrix}$$

The reason we keep track of $NoState$ will be evident once we show how the conquer step work.

3.3.2 Conquer Step

The most general case is a naive lexer that takes the first accepting state it can find. When two lists of tokens are combined it will create two tokens if the out state of the first list is accepting then the second list will be appended to the first list. The other case us when the first list of tokens does not end in an accepting state. In this case the lexer will try to find an in state in the second list that is the same as the out state of the first transition.

$$\begin{bmatrix} 0 & Single'/' & 1 \\ 1 & Single'/' & Accepting 5 \end{bmatrix} `combineTokens` \begin{bmatrix} 0 & Single'/' & 1 \\ 1 & Single'/' & Accepting 5 \end{bmatrix} = \begin{bmatrix} 0 & Single'//' & Accepting 5 \\ 1 & Multiple'/'[]'/' & Accepting 1 \end{bmatrix}$$

This won't work as a lexer for most languages since it will lex a variable to variables where the length is a single character, for example "os" will be lexed as two tokens, "o" and "s". To solve this some more work is needed to be done.

3.3.3 Longest Match

Instead of taking the naive aproach where a token is created if you find an accepting state, the rule for creating a new token will instead be when the combination of two transitions yields *NoState* the lists will be appended. That is, when there is an out state from the first transition that corresponds to an in state of the second transition and the out state of the second transition isn't *NoState*, the last token of the first transition and the first token of the second transition will become one token, otherwise append the second list to the first list.

$$\begin{bmatrix} 0 & Single'//' & Accepting 5 \\ 1 & Multiple'/'[]'/' & 1 \end{bmatrix} `combineTokens` \begin{bmatrix} 0 & Single'\backslash n' & Accepting 6 \\ 1 & Single'\backslash n' & 1 \\ 5 & Single'\backslash n' & NoState \end{bmatrix} = \begin{bmatrix} 0 & Multiple'//'[]'\backslash n' & Accepting 6 \\ 1 & Multiple'/'[]'/\backslash n' & 1 \end{bmatrix}$$

The second case is when the out state for the right token list is *NoState*. This means that the two lists of tokens can't be combined. In this case the first token in the second list will be viewed as the start of a token and the last token in the first list will be viewed as the end of a token.

3.4 The First Attempt

The first naive solution whas to b̈ruteforceẗo find the lex. This was showned to be to resource-eating. But it describes the general idea of how the problem could be solved. Why it was a bad solution will be described futher on in the text.

When the code is divided the lexer doesn't know if the string (or character) it lexes is the first, last or is somewhere in the middle of a token. Instead of checking what type of token the string will be (if it were to begin from the starting state) it saves all the possible state transitions for that string.

In the examples that follow below state 0 is considered the starting state and state $1 - 6$ are considered accepting.

Example 3.4.1 (Transition map for a token)**.** A hypothetical transition map for the char 'i'.

$$\begin{array}{cc} \multicolumn{2}{c}{'i'} \\ in & out \\ 0 & 1 \\ 1 & 1 \\ 8 & 7 \end{array}$$

In the base case the lexer will map all the transitions for all individual characters in the code and construct partial tokens of them. The conquer step will then combine two of these at a time by checking which possible outgoing states from the first token can be matched with incoming states from the second token. If there are such pairs of outgoing states with incomming states, then a new partial token is created.

Example 3.4.2 (Combining two tokens)**.** 'if' can be an ident (state 1) or part of 'else if' (state 5).

$$\begin{array}{cc} \multicolumn{2}{c}{'i'} \\ in & out \\ 0 & 1 \\ 1 & 1 \\ 8 & 7 \end{array} \text{`}combineToken\text{`} \begin{array}{cc} \multicolumn{2}{c}{'f'} \\ in & out \\ 0 & 1 \\ 1 & 1 \\ 7 & 5 \end{array} = \begin{array}{cc} \multicolumn{2}{c}{'if'} \\ in & out \\ 0 & 1 \\ 8 & 5 \end{array}$$

If there are no pairs of outgoing states which match the incomming states the lexer will try to combine the first token with as much of the second token as possible. In this case there will be a remainder of the second token, The lexer can now be sure that the begining of the remainder is the begining of a token and that the merged part is the end of the token before. Since the lexer knows the remainder is the begining of a token it strips all transitions but the one that has incomming state as starting state. Since the start token is the end of a Token it strips all but the transitions ending in an accepting state.

Example 3.4.3 (Combining a token a part of the second token)**.** 'ie' ends in the accepting state for ident (1) and '_' starts in the starting state.

$$\begin{array}{c}'e\_'\\ \begin{array}{cc} in & out\\ 10 & 8\end{array}\end{array} = \begin{array}{c}'e'\\ \begin{array}{cc} in & out\\ 0 & 11\\ 1 & 1\\ 6 & 1\\ 10 & 9\end{array}\end{array} \text{`}combineToken\text{`} \begin{array}{c}'\_'\\ \begin{array}{cc} in & out\\ 0 & 2\\ 2 & 2\\ 9 & 8\end{array}\end{array}$$

$$\begin{array}{c}'i'\\ \begin{array}{cc} in & out\\ 0 & 1\\ 1 & 1\\ 8 & 7\end{array}\end{array} \text{`}combineToken\text{`} \begin{array}{c}'e\_'\\ \begin{array}{cc} in & out\\ 10 & 8\end{array}\end{array} = \begin{array}{c}'ie'\\ \begin{array}{cc} in & out\\ 0 & 1\\ 1 & 1\end{array}\end{array} ++ \begin{array}{c}'\_'\\ \begin{array}{cc} in & out\\ 0 & 2\end{array}\end{array}$$

# Perhaps remove this part
However the remainder may not have the start state as a possible incomming state. In this case the lexer tries to find the largest possible token (that has the starting state as incomming state) and tries to construct a token of the rest of the remainder, repeating this procedure until the entire remainder has been split into acceptable tokens. All the tokens accept the one that is on the very end of the sequence will have all but their accepting states stripped. This case does occur quite frequently since most languages has comments and strings which can contain anything.
Example 3.4.4 (Handling the remainder)**.** '_' starts in the starting states and ends in an accepting state and 'e' starts in the starting state, it doesn't have to end in an accepting state.

$$\begin{array}{c}'\_i'\\ \begin{array}{cc} in & out\\ 9 & 7\end{array}\end{array} = \begin{array}{c}'\_'\\ \begin{array}{cc} in & out\\ 0 & 2\\ 2 & 2\\ 9 & 8\end{array}\end{array} \text{`}combineToken\text{`} \begin{array}{c}'i'\\ \begin{array}{cc} in & out\\ 0 & 1\\ 1 & 1\\ 8 & 7\end{array}\end{array}$$

$$checkRemainder\left(\begin{array}{c}'\_i'\\ \begin{array}{cc} in & out\\ 9 & 7\end{array}\end{array}\right) = \begin{array}{c}'\_'\\ \begin{array}{cc} in & out\\ 0 & 2\end{array}\end{array} ++ \begin{array}{c}'i'\\ \begin{array}{cc} in & out\\ 0 & 1\end{array}\end{array}$$

When all partial tokens has been combined in this way the resulting sequence of tokens represents the the code the lexer was run on.

3.4.1 Pitfalls

The above rules will work for very simple languages. When comments are introduced you will get the problem that the whole code can be one long partial comment token. To remedy this you can add two rules:

- Every time you combine two tokens you only do so if the combination has a transition from the starting state.
- If two tokens can be combined completly, check if the next token can be combined aswell.

This ensures that every token starts in the starting state and that each token is as long as it can be.

This also has some problems though. When keywords like "else if" are introduced the lexer will start to lex like in example 3.4.5. To solve this the lexer checks when two tokens are completely uncombinable if the first of these have an accepting state as outgoing state. If the token don't have an accepting out state, the lexer tries to break up the token until it does. The exception to this rule is single characters which are permitted to not have no accepting out states.
Example 3.4.5 (else if lexing)**.** Somewhere in the middle of the code "... 1 else 0 ..."

String	Type
1	*Number*
␣	*Space*
else␣	*Nothing*
0	*Number*

Example 3.4.6 (Devide and Append)**.** The lexer will always try to build as lage tokens as possible. When it realizes that this cant be done it has to backup and try to combine the parts in a different way. This example will show how this is done in theory.

The code segment for this example is:

"else return".

The tree in fig. 3.2 shows the first step of the token combine rutine. Clearly this returns a nonexsisting token. From here when the lexer has found that there are no tokens for this lexeme it will try to split the left child token.

split("else ") => ["else"," "]

Now the lexer has a pair of two lexems that represent valid tokens. The lexer knows that combinding these two lexems in the pair returns in a NoToken result. So The only thing to do is to try to combine the right token in the pair with the right child token and let the token to the left in the pair stand alone. This also return a NoToken. So the

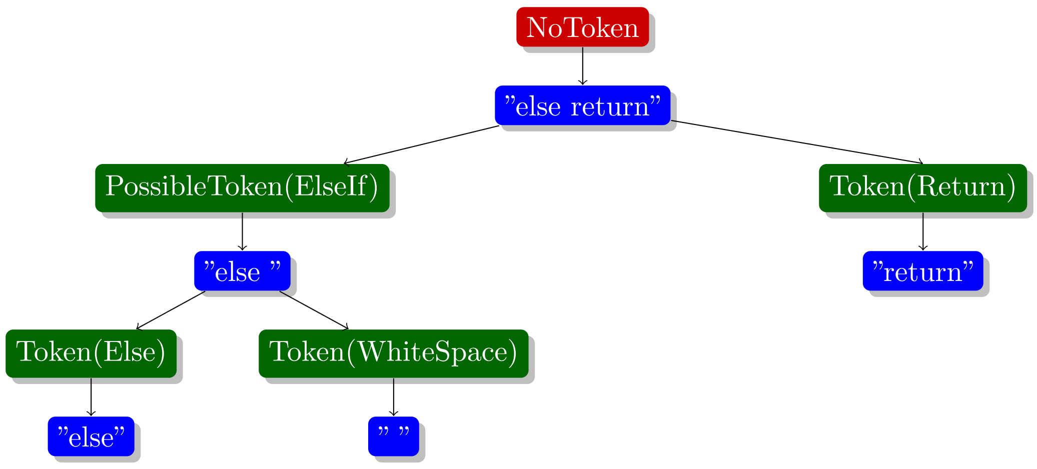


Figure 3.2: Lexer thinks "else " is an "else if" pattern.

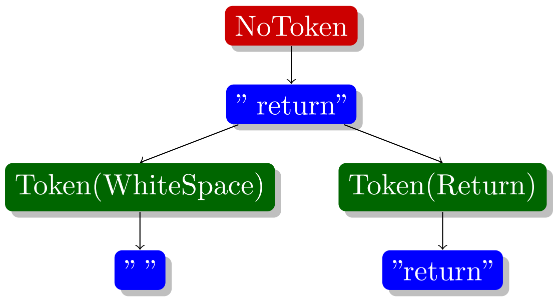


Figure 3.3: Lexer tries to combine an white space with a return statement

same thing will be done again. The lexer tries to split the left child before NoToken was given. In this case the whitespace.

split(" ") => []

But becouse the whitespace is of the lowest form and is not build up by smaller tokens the resulting list from the split function will be empty. Now the lexer knows that this token must be by it self. The "return" is the last lexeme in this example code so the lexer can't combine it futher. Thus the lexer has found the resulting sequence of tokens:

[(Token(Else), "else"), (Token(WhiteSpace)," "), (Token(Return), "return")]

3.5 A Better Solution

A text of the general idea behind this solution, why it should be better then the first attempt. And so on and so on!!

3.5.1 The Data Structure

For the lexer to be able to represent every possible substring of the code in the same datastructure. There need to be a generic representation for tokens and subtokens. A subtoken is a representing of a lexeme for that subtoken and a list of accepting tokens for that lexeme. The final tokens is a sequence of subtokens the output state for this sequence and an suffix, this suffix can be empty or conssist of an alternetiv ending for the sequence of tokens. What this is and how it works will be described more in detail in section 3.5.2. In fig. 3.4 is a code representation of the datastructure

```
type State = Int
type Transition = State -> Tokens -- Transition from in state to Tokens
data Tokens    = Tokens {currentSeq :: Seq PartToken
                        ,outState   :: State
                        ,suffix     :: Suffix}
                 deriving Show
data Suffix    = None
               | End {getToks :: Tokens}
                 deriving Show
data Size      = Size Int
type LexTree   = FingerTree (Table State Tokens,Size) Char
data PartToken = Token { lexeme      :: String
                       , token_id    :: Accepts}
type Accepts   = [AlexAcc (Posn -> String -> Token) ()]
```

Figure 3.4: Datastructure for tokens and lexems.

3.5.2 The Power of Suffix

The suffix is a structure for saving one alternative sequence of acceptable subtokens, for a lexed token that can not end in a accepting state. It is build up by subtokens of the not yet accepting token that the lexer is trying to lex. If the token finally end in a accepting state, after some combination with an other token, the suffix of this accepting token will be set to None. Where None is a data-structure for showing that this token is complete.

This could be shown by example 3.5.1.

Example 3.5.1. Two tokens is trying to be lexed together. A token a which has the lexeme ëlse ;̈ an else with a whitespace following it. A token b which has the lexeme r̈eturn:̈ The structure for theses tokens and there suffixes is shown in fig. 3.5

Now when Token$_a$ and Token$_b$ are combined it will result in an illegal token, that is the

$$\begin{array}{rcl}
\text{Token}_a & : & {}^{\text{start\_state}}\downarrow \underbrace{\text{else }\_}\downarrow^{\text{out\_state} \notin \text{accepting\_states}} \\
\text{Suffix}_a & : & \left[{}^{\text{start\_state}}\downarrow \underbrace{\text{else}}\downarrow^{\text{accepting\_state}},\ {}^{\text{start\_state}}\downarrow \underbrace{\ \_\ }\downarrow^{\text{accepting\_state}}\right] \\
\text{Token}_b & : & {}^{\text{start\_state}}\downarrow \underbrace{\text{return}}\downarrow^{\text{accepting\_state}} \\
\text{Suffix}_b & : & \text{None} \\
 & & \text{where accepting\_states is a list of all accepting states.}
\end{array}$$

Figure 3.5: The tokens and there suffixes.

out_state will not be a valid state. So something has to be done, because Token$_a$ does not end in an accepting state. And since the lexer always want the longest possible tokens, the lexer will try to combine this in an other way. The lexer then replaces Token$_a$ into it's suffixes, saves all the suffixes as finished tokens except for the last suffix. In this case the $\underbrace{\ \_\ }$ suffix. It now tries to combine this suffix with Token$_b$, which will also return a no valid state. And recursively does the same thing again. But this time the $\underbrace{\ \_\ }$ suffix is a single character token and has no suffixes. So the lexer can't do anything further with this, therefor the final result will be three separate tokens. shown in fig. 3.6

$$\begin{array}{rcl}
\text{Token}_{a1} & : & {}^{\text{start\_state}}\downarrow \underbrace{\text{else}}\downarrow^{\text{accepting\_state}} \\
\text{Suffix}_{a1} & : & \text{None} \\
\text{Token}_{a2} & : & {}^{\text{start\_state}}\downarrow \underbrace{\ \_\ }\downarrow^{\text{accepting\_state}} \\
\text{Suffix}_{a2} & : & \text{None} \\
\text{Token}_b & : & {}^{\text{start\_state}}\downarrow \underbrace{\text{return}}\downarrow^{\text{accepting\_state}} \\
\text{Suffix}_b & : & \text{None}
\end{array}$$

Figure 3.6: The resulting tokens.

3.6 Lexical Errors

Since the lexer has to be able to handle any kind of possible not "complete" tokens, error handling can be done in different ways. One approach is to simply return as many tokens as possible from the code and where there might be lexical errors the lexer returns the error in as small parts as possible.

Example 3.6.1 (A lexer that only lexes letters)**.** When the lexer encounters the string "what @ day" it would return:

String	Type
What	*Word*
'_'	*Space*
'@'	*No_Token*
'_'	*Space*
day	*Word*

4

Implementation

Here we should talk about bnfc and alex. What we use from the differnet programs. How this is usefull is it becouse of lazynes or are the existing solutions good??

4.1 Alex

What is this, how have this tool helped us in this project. Some basics how it works and why it therefore fits our project.

4.2 Monoid

What is this, how have this tool helped us in this project. Some basics how it works and why it therefore fits our project.

4.3 Fingertree

What is this, how have this tool helped us in this project. Some basics how it works and why it therefore fits our project.

4.4 Sequences

A sequence in Haskell is a form of sophisticated list. That is a list with better performance than the basic [] list notation. Where a list in Haskell has $\Theta(n)$ for finding, inserting or deleting elements, that is in a list there is only known current element and the rest of the list. Which will result in for example finding the last element of a list, the computer must look at every element until the empty list has been found as the rest of the list. Where in a sequence the last element can be obtained in $\Theta(1)$ time. Adding a element anywhere in the sequence can be done in worst case, $\Theta(logn)$ [7].

Since this project is about creating as real-time lexing tool, performance is very important. String in Haskell are just a list of characters, [Char], and therefor has $\Theta(n)$ in time cost. Lexing is working with strings in a high frequency and there for there is a idea of instead define a string as a sequence of characters instead of a list of character.

5

Result

The incremental lexer has ass mention before three requirements, Robust, Efficient and Precise. To be robustness the lexer don't crash when encounter an error in the syntax. That is, the lexer will find the correct token if the syntactical error is fixed by the user. For it to be efficient the feedback to the user must be instant. Finally to be precise the lexer must give a correct result. This chapter will talk about how this requirements are tested and if they are fulfilled.

5.1 Robustness

5.2 Efficiency

5.3 Preciseness

For an incremental lexer to work, the lexer must be able to do lexical analysis of any substring of a code. That is, come to correct result for that substring. It must also be able to merge any two substrings neighboring in a code and the result of this merge must be the same as lexing the two substrings as one substring.

One would think that haskells quickcheck would be a good way to generate in-data for the lexer. Since quickcheck is build to generate good input for testing a function for any arguments [3]. But the problem is not to test any string representation. It is instead to test valid code segments and any substring of this code segments. Also invalid pieces of code, to see that the lexer informs the user for syntactical errors in these codes. To write a input generator in quickcheck which would generate full code with all of it components and all the different properties would be have to high cost in develop time for the outcome. It would be more time efficient to test the lexer on several different code files. There for the testing of the incremental lexer has not been done with the help of quickcheck.

To test if the lexer is as close as possible to the result done by the compiler the resulting token sequence should be the same sequence given from Alex, a open-source lexer written in Haskell, when lexing the same code-string. To control this the separate tokens in the resulting sequences are compared for equality. The code in fig. 5.1 shows the test for equality:

```
checkCorrectTokens :: IncLex.Tokens -> Alex.Tokens -> Boolean
checkCorrectTokens itoks atoks =
  let tokTupple = zip itoks atoks
  in [] == filter (\(iToken, aToken) -> iToken `notEquals` aToken) tokTupp
```

notEqual function is a function which patter-match on the to different tokens and returns true if they are not of the same type.

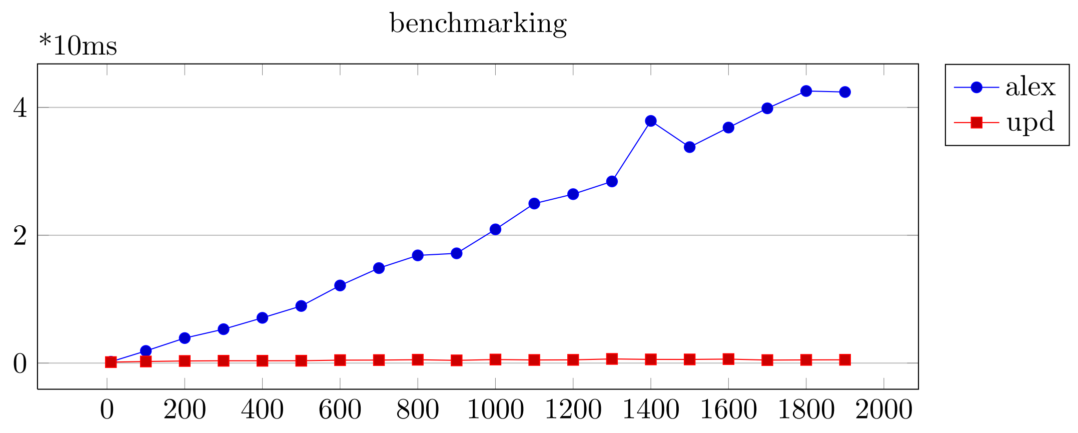
Figure 5.1: Code for testing tokens from IncLex is equal to tokens from Alex.

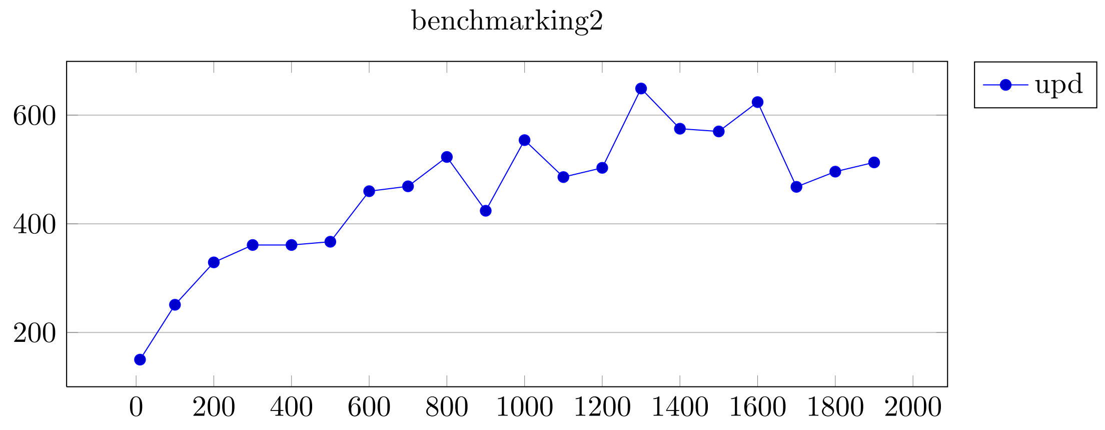
5.3.1 Check Splits

5.3.2 Check Merges

5.3.3 Performance

upd splits a piece of code into two parts then evaluates the out state in order to make sure that the entire structure is evaluated.





# Just some bench data:

Hugos Home-Computer: AMD X2 3.1 GHz, 4 Gb Ram, 32 bit OS Windows 8 Func: Alex - 12.38 ms IncLex - 55.02 ms Update - 35.29 ms Array: Alex - 14.22 ms IncLex - 331.75 ms Update - 9.92 ms

Hugos Laptop: Intel Celeron M 1.6 GHz, 1 Gb Ram, 32 bit OS Unix

Func: Alex - 24.12 IncLex - 182.47 ms Update - 113.22 ms Array: Alex - 26.54 ms IncLex - 658.53 ms Update - 11.40 ms

6

Performance Analysis

#How fast is the lexer. How have we come to this conclusion?

7

Discussion

#Discuss discuss!!

8

Conclusion and Futher Work

#what will our Minions do????

Bibliography

[1] Alfred V. Aho. *Handbook of theoretical computer science (vol. A)*, chapter Algorithms for finding patterns in strings, pages 255–300. MIT Press, Cambridge, MA, USA, 1990.

[2] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2006.

[3] Koen Claessen and John Hughes. Quickcheck: a lightweight tool for random testing of haskell programs. *SIGPLAN Not.*, 35(9):268–279, September 2000.

[4] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd edition, 2009.

[5] Chris Dornan, Isaac Jones, and Simon Marlow. *Alex User Guide*. Haskell. www.haskell.org/alex/doc/html/index.html.

[6] Michael T Goodrich and Roberto Tamassia. *Data Structures and Algorithms in Java, 4th Edition*. John Wiley & Sons, 2005.

[7] RALF HINZE and ROSS PATERSON. Finger trees: a simple general-purpose data structure. *Journal of Functional Programming*, 16:197–217, 3 2006.

[8] Dan Piponi. Fast incremental regular expression matching with monoids. blog.sigfpe.com/2009/01/fast-incremental-regular-expression.html, 2009.

[9] Aarne Ranta and Markus Forsberg. *Implementing Programming Languages*, chapter Lexing and Parsing, pages 38–47. College Publications, London, 2012.

[10] R.W. Sebesta. *Concepts of Programming Languages [With Access Code]*. Always learning. Pearson Education, Limited, 2012.

[11] M. Sipser. *Introduction To The Theory Of Computation*. Advanced Topics Series. Thomson Course Technology, 2006.

[12] R. S. Sundaresh and Paul Hudak. A theory of incremental computation and its application. In *Proceedings of the 18th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '91, pages 1–13, New York, NY, USA, 1991. ACM.