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 Program MPALG]

JONAS HUGO

KRISTOFER HANSSON

Chalmers University of Technology
Department of Computer Science and Engineering
Göteborg, Sweden, May 2015

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

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

Department of Computer Science and Engineering
Göteborg, Sweden May 2015

Abstract

This report aims to present a way to do lexical analysis incrementally instead of the present norm: sequential analysis. In a text editor, where updates are common, an incremental lexer together with an incremental parser could be used to give users real time parsing feedback. Previous work has proven that regular expressions can be implemented incrementally [11], we make use of these findings in order to show that it can be expanded to a lexical analyzer. The results in this report shows that an incremental lexer is efficient, it can do an update in $\Theta \log n$ time which makes it suitable when updates are common. In order for an incremental lexer to be applicable it has to be precise, only correctly lexed tokens are relevant. It is required that an incremental lexer is robust, a lexical error for a partial result must be handled gracefully since it may not propagate to the final result. To achieve incrementality a divide and conquer tree structure, fingertrees, is used that stores the intermediate lexical results of all the partial trees. When an update to the tree is made only the effected node and its parents are updated. The state machine in the implementation is generated by Alex since it is efficient and enables support for lexical analysis of different languages. The report finishes with giving suggestions for improvements to the drawbacks found during the work, The suggestions given are mainly for improving space complexity. This report shows that an implementation of an incremental lexer can be precise, efficient and robust.

Acknowledgments

We would like to take the chance of thanking our supervisor at department of computer science, Jean-Philippe Bernardy, with which we have had a lot of constructive discussions. We would also like to thank our examinator at the department of computer science Bengt Nordström. Lastly we would like to thank everyone that has helped proof read this report.

Jonas Hugo & Kristofer Hansson, Göteborg May 2015

Contents

Acronyms

BNFC Backus-Naur-Form Converter.

DFA Deterministic Finite Automata.

IDE Integrated Development Environment.

NFA Non-deterministic Finite Automata.

Glossary

Alex Is a lexical analyzer generating tool written in Haskell.

Finger Fingers are the first and last elements of a tree and they can be accessed in $O(1)$ time.

Fingertree Is a tree data structure with fast access time to elements.

Incremental (computation) Save running time by only update directly depending data.

Lazy Evaluation Evaluate an expression first when the value is needed.

Lexeme Is the string representation of a token.

Lexer Given a string, find abstract tokens represented by the string.

Monoid Is a mathematical structure for a binary operator and an identity element.

Normal Form Is the form of an expression when no more computations can be made.

Regular Expression Is a pattern for which a sequence of symbols can follow.

Regular Language Is a formal language where all lexems in the language can be expressed using regular expressions.

Sequence Is a data structure which represents a finite ordered collection of elements.

Spine Is the subtree of a fingertree.

Token Is an abstract class for a lexeme matching a pattern.

Transition Is a path in a Finite Automata from an in-state to an out-state given a string.

Transition Map Is a collection of all possible transitions for a string in each state.

1

Introduction

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 compiler strength 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.

In order to implement a parser with the characteristics described above; robust, precise and efficient; a lexer that has the same properties is needed. This project aims to implement such a lexer.

1.1 Scope of work

Existing lexical analyzers are sequential. When the text is updated the lexer must start the lexical analysis from the beginning. The goal of this project is to create an algorithm that, after an update to the text, only needs to recalculate the update and the part of the result effected by the update. The recalculation should have time complexity $\Theta log(n)$ in order to be run in real time, for example in a text editor with immediate update.

Chapter 2 gives a general understanding of how lexical analyzers work. Chapter 3 presents tools needed for a divide and conquer implementation to work. Chapter 3 also gives an overview of the ideas behind a divide and conquer lexer in order to give enough understanding for the algorithm this report proposes. A robust implementation of the algorithm is presented in chapter 4 with explanations on how different cases are handled.

Tests for preciseness, time performance and space performance are explained and their corresponding result are presented in chapter 5. A discussion on where a divide and conquer lexer is useful is presented in chapter 6.

1.2 Related Work

This project revolves around the idea of using incremental regular expressions. Piponi wrote a blog post about how to implement incremental regular expressions using finger trees [11]. The solution to matching regular expressions incrementally in the blog post gives a good starting point to this project, however a lexer does not match a string against one expression. A lexer matches a string against a set of regular expressions and returns which expressions where matched and in what order rather then answering if the string matched the expressions. The "longest match" rule for lexers further complicates the issue [11].

Bernardy and Claessen [3] wrote a paper titled "Efficient Divide-and-Conquer Parsing of Practical Context-Free Languages" that describes an efficient parallel parser built on Valiants algorithm [16]. The paper proves that for a defined set of input the complexity for the parser will be $\Theta log^3(n)$. Since the implementation of the parser is done in BNFC [6] it uses a sequential lexer generated by alex [5]. The lexical analyzer this project purposes is a divide and conquer solution which could be integrated to the parser proposed by Bernardy and Claessen to get a divide and conquer solution from the programming code to the result of the parser.

2

Lexer

This chapter describes the concepts of a lexer, lexical analyzer, in detail. A lexer can be seen as a pattern matcher and is the first part of a syntactical analyzer. A lexer has the responsibility to translate a text into abstract tokens. Abstract tokens typically tell what a part of the text belongs to, for example classes of words, numbers or symbols. The resulting tokens of the lexer is given to the syntactical analyzer which is responsible for identifying expressions and statements which the tokens represent [13]. To implement a lexer several fundamental concepts of formal language theory can be used. Concepts like regular expressions, regular sets and finite automata [1].

2.1 Benefits of Disjointed Parsing

It is common that a parser has a pre-step where the plain text is transformed in to some more computer readable form. This step is performed by a lexical analyzer. The computer-friendly output is then then given to the syntactical analyzer of the parser. Splitting up a parser into a number of different tasks have several benefits [13]. Here follows some benefits from breaking out the lexical analyzer from the syntactical analyzer.

If the lexical analysis is stripped out of the parser, the parsing step can be designed in a cleaner way. The lexical analyzer can also be designed in a cleaner and smarter way. A lexer can for instance ignore to pass along unneeded (for the parser) data, like white spaces and comments. This opens up for a cleaner design when defining a new programming language. The syntactical part of the parser will only receive the output from the lexer as input. The output of the lexer is described in more detail further on in this chapter [2].

Splitting a big problem into smaller specific sub-problems enables methods that are refined on specific tasks to be used. This can result in a more efficient parser [2]. The lexical part of a parser make up for a big portion of the overall compilation time of a compiler, therefore breaking out this step enables the possibility to explicit optimize the lexer [13].

Breaking out the lexical analyzer from the parser makes it possible for the syntactical analyzer to solve its problem in a generic way. When changing language to parse, the only part that needs to be changed in the parser is the lexical analyzer. This opens up for portability [2][13].

If there is an illegal character sequences inside the code it will be detected by the lexer and feedback will be given to the user [13]. Because the lexer can find and report these errors to the user, there is no need to go into the syntactical analysis of the parser. Hence saving running time for giving the user feedback.

2.2 Fundamentals of Lexical Analysis

As mentioned earlier in this chapter, the responsibility of the lexical analyzer is to transform a human readable text to an abstract computer readable list of tokens. There are different techniques a lexer can use when finding the abstract tokens representing a text. This section describes the techniques used when writing rules for the tokens patterns.

2.2.1 Regular Expressions

Regular expressions are used to verify if a sequence of symbols matches a pattern. Due to the definition of regular expressions they cannot describe all possible patterns. However, they are in most cases good enough for lexers.

Example 2.2.1 (Valid C Idents [2])**.** A valid C identifier must start with a letter character and then have zero or more characters or digits. To describe this an element $letter \in \{\mathrm{a} \ldots \mathrm{z}\} \cup \{\mathrm{A} \ldots \mathrm{Z}\} \cup \{\_\}$ is introduced and another element $digit \in \{0 \ldots 9\}$.

By using these elements the regular expression for describing all legal C identifiers can be expressed in the following way: $letter(letter|digit)*$.

Definition 2.2.2 (Regular Expressions [1])**.**

1. The following characters are meta characters: $meta = \{'|', \ '(', \ ')', \ '*'\}$.
2. A character $a \notin meta$ 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 X_i matches r 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.

■

Consider the definition of regular expressions seen in definition 2.2.2, by introducing a priority level and associativity to the different operators parentheses can be eliminated. The operator with the highest priority is the $*$ operator. The second highest is the *concat* operator $(r_1)(r_2)$ and the operator with the lowest level is the *or* operator $|$. The $*$ operator can not have a associativity since it only takes one argument. The two binary operators *concat* and *or* are left-associative [1].

2.2.2 Languages

A language is built up by an alphabet which is represented by a finite collection of characters. These symbols can build up strings and a language is a countable set of these different strings [2]. For example the alphabet Unicode used by computers to represent text includes over 100,000 different symbols [2]. This means that a language can be enormous.

There are different types of languages. Formal languages are described by a set of systematic rules and they are a subset of all the languages. The lexer can however not work with all formal languages, only with the languages which can be described by regular expressions, these are called regular languages [12].

2.2.3 Regular Definitions

To be able to reuse already written expression, an identifier d can be assigned to an expression, d can then be used in expressions. However this introduces the problem of recursive definitions. To counteract this the properties for the identifiers and expressions are defined as follows.

A set of regular definitions for an alphabet Σ is given, which can be seen in fig. 2.1

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

Figure 2.1: List of definitions and their regular expressions

The following property apply to the definition identifiers $d_1...d_n$: a definition identifier d_i is a new symbol not already present in the alphabet Σ and not equal to any other definition identifier $d_x \in d_1...d_n$ where $i \neq x$. A new regular expression r_i can work on the alphabet and all the previously defined identifiers, $\Sigma \cup d_1...d_{i-1}$ [2].

2.3 Tokens, Atoms of a Language

When rules have been defined for a language, the lexer needs structures to represent the rules and the result from lexing the text. This section describes the structures which the lexical analyzer uses for representing the abstract data; what these structures are used for and what is forwarded to the syntactical analyzer.

The following three different structural concepts are vital to the lexical analyzer:

- A **token** is an abstract for representing an atomic code segment. The token is represented by a name and an optional attribute for holding the value of the token [2].
- A **pattern** is the regular expression for describing the text format on which a token can be represented [2]. For example a string in most languages is represented by first a " character and zero or more characters and finally ends with a " character.
- A **lexeme** is the text that matches the pattern bound to the token. Therefore a lexeme can be viewed as an instance of an abstract token [2].

As mentioned before, a token can carry an optional attribute. When a token can be represented by several different instances, lexemes, this attribute is used for giving the specific value. For example a string token can be represented by the different strings "", "a", "b" and so on. However different parts of the compiler may need to know which token instance that was found. Therefore the lexer need to pass the information further on [2].

To summarize, a lexer reads characters from a code and finds the largest continues sequences which builds up valid tokens [13]. As mentioned before it is not always relevant to return an attribute to the token. These cases can be when finding keywords of the language, like in if, for and $while$. There are cases when it make no sense to return the found token. Example of such cases could be tokens for comments and white-spaces, which in most languages has no relevance to the compiled code. In these cases the lexer just drops the token and continues the lexical routine [2]. example 2.3.1 shows how a small piece of code is divided in to abstract tokens using the rules in fig. 2.2. The complete language specification can be found in appendix A.

$\langle letter \rangle \in$ {'a' - 'z'} ∪ {'A' - 'Z'} ∪ {'_'}

$\langle digit \rangle \in$ {0 - 9}

$\langle identifier \rangle$ ::= $\langle letter \rangle$ ($\langle letter \rangle$ | $\langle digit \rangle$)*

$\langle string \rangle$ ::= '"' [∧ '"']* '"'

$\langle multi\text{-}line\ comment \rangle$ ::= '/*' ([∧ '*'] | '*' [∧ '/'])* '*/'

$\langle reserved\text{-}words \rangle$ ::= '(' | ')' | '{' | '}' | ';' | '=' | '++' | '<' | '+' | '−' | '*'

Figure 2.2: Grammar rules for example 2.3.1 & example 2.4.1

Example 2.3.1 (Logical grouping [13])**.**
Consider the following text; to be lexed:

```
fileName = filePath + ".png";
```

Given the regular language defined in appendix A, the lexical analyzer would use the rules defined in fig. 2.2 and produce the resulting tokens shown in fig. 2.3.

Token	Lexeme
Identifier	fileName
Reserved	=
Identifier	filePath
Reserved	+
String	".png"
Reserved	;

Figure 2.3: Result of lexing the code in example 2.3.1

2.4 From a Text to Tokens

The previous section show how Regular expressions can be used to express patterns for tokens. In this section the different techniques on how to transform a sequence of characters into abstract tokens using these patterns is described.

2.4.1 Transition Diagrams

To recognize tokens from a pattern transitions diagrams can be used, these are directed graphs consisting of nodes and edges [13]. The nodes correspond to the discrete states in the transformation process. In the transition diagram there are three kinds of states. The first is the the starting state, there can only be one starting state in the graph. It is from this state the process start when a new token should be found. There are at least one accepting state, these states represents that a valid token has been found. Then there can be zero or more none-accepting states. These states represents that a token has not yet been found [2].

The edges in the graph are represented by the input character which must be found to be able to traverse between the two states which the edge connects. If there is no valid edge out of an accepting state, the found token is said to be the longest match (see section 2.4.2) then that token is returned and the lexer starts reading the next character from the starting state [2].

2.4.2 Longest Match

If there are multiple feasible solutions when performing the lexical analysis, the lexer will return the token that is the longest. To manage this the lexer will continue in the transition diagram if there are any legal edges leading out of the current state, even if it is an accepting state [2].

The above model introduces a new problem. If the lexer ends up in a state that is not accepting and do not have any legal edge out of that state, the lexer can not return a token. To solve this the lexer has to keep track of what the latest accepting state was. When the lexer reaches a state with no legal edge out of it, the lexer returns the token corresponding to the last accepting state. The tail of the string, the part that was not in the returned token, is then lexed from the initial state as part of a new token [2].

Example 2.4.1 (Longest Match)**.** Consider the following text; to be lexed.

```
/*fileName = filePath + ".png";
```

Although this piece of C code is not syntactically correct, there are no lexical errors in it. Since the text starts with a multi line comment sign the lexer will try to lex it

Token	Lexeme
Reserved	/
Reserved	*
Identifier	fileName
Reserved	=
Identifier	filePath
Reserved	+
String	".png"
Reserved	;

Figure 2.4: Result of lexing the code in example 2.4.1

as a comment. When the lexer encounters the end of the text it will return the token corresponding to the last accepting state and begin lexing the rest from the initial state. The rules relevant to this example are defined in fig. 2.2 the rest of the rules can be found in appendix A.
The result can be found in fig. 2.4.

2.4.3 Finite Automata

To recognize members of regular languages, which are languages that lexers can be used with, a mathematical machine called finite automata can be used [13]. Finite automata are like regular expressions, purely recognizers. A finite automate only say if an input sequence is valid or not [2].

There are two different forms of finite automata, which both are capable of working on regular languages [2]:

Non-deterministic Finite Automata (NFA): As the name suggest there are no requirements of a deterministic path for an input sequence in this type of automata. A state may have multiple edges for the same symbol. Also edges my take no symbol, the empty string ϵ.

Deterministic Finite Automata (DFA): In this form there can only be one path for an input. That is, a state must have exactly one edge per input symbol leaving the state and edges are not allowed to have the empty string ϵ as symbol.

There are two common ways of representing a finite automata, transition diagram and transition table. A transition diagram is a directed graph where the nodes are the states in the automata and the edges represent the symbol needed for the next state. A transition table is a table where the rows represents the current state, the columns the next symbol and the cell is the next state. Examples of how a NFA can be described by both a transition diagram and a transition table can be seen in example 2.4.3 [2].

Non-deterministic Finite Automata

A string is recognized as an accepted string by a NFA if there exist at least one path from the starting state to one of the accepting states where the edges symbols along this path builds up the string [2].

The formal definition of a non-deterministic finite automaton follows:
Definition 2.4.2 (Non-deterministic Finite Automata [14])**.** 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. $\delta : Q \times \Sigma \to P(Q)$ is a transition function,
4. $q_0 \in Q$ is the start state, and
5. $F \subseteq Q$ is the accepting states.

Example 2.4.3 shows how the transition diagram and transition table representation will look like for a given regular expression. Since a state can have several edges with the same symbol, the transition function does not map to a single state.

Example 2.4.3 (Regular Expression to Transition Diagram & Transition Table [2])**.** Given the regular expression $(a|b|c)(a|c) * c$ a transition diagram can be created that represents the expression, see fig. 2.5. The transition table in fig. 2.6 represents the same graph.

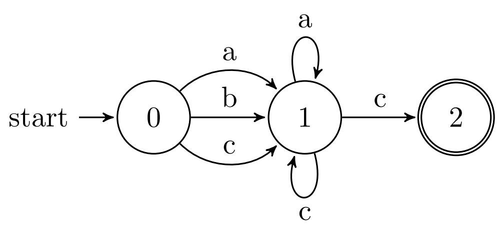

Figure 2.5: Transition Diagram, accepting the pattern $(a|b|c)(a|c) * c$

Transition tables store all possible transitions which gives it a quick lookup time. However there is often a majority of states which does not have any transitions for some input symbols. And since the table stores all states it will need a lot of data space, especially for situations when the alphabet for the language is large [2].

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

Figure 2.6: Transition Table, accepting the pattern $(a|b|c)(a|c)*c$

Deterministic Finite Automata

DFA is a NFA with stricter rules. These rules are that edges can not be labeled with the empty input ϵ and there is exactly one edge for each symbol in the alphabet out of every state [2].

Finite Automata can be generated from regular expressions. That is, a NFA can be generated from regular expressions and a DFA can be generated from a NFA. This goes the other way as well, and DFA can be converted into a regular expression. A lexer uses a DFA as the algorithm to match a lexeme to a specific token [2].

The formal definition of a deterministic finite automaton follows:
Definition 2.4.4 (Deterministic Finite Automata [14])**.** 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 accepting states.

Example 2.4.5 (DFA representation of Regular Expression [2])**.** A DFA representation of the regular expression from example 2.4.3 is shown in fig. 2.7

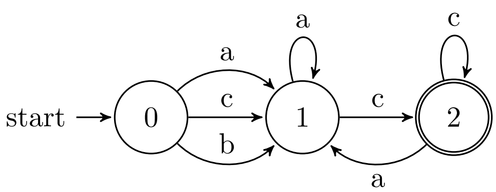

Figure 2.7: DFA, accepting the regular expression: $(a|b|c)(a|c)*c$

3

Divide-and-Conquer Lexer

An incremental divide and conquer lexer works by dividing the sequence to be lexically analyzed, into small parts; analyzes them and then combines them. In the base case the lexical analysis is done on a single character. The conquer step then combines the smaller tokens into as large tokens as possible. The end result is a sequence of tokens that represent the code. How this is done is described in this chapter.

3.1 Divide and Conquer in General

This section gives an idea of how the Divide and Conquer algorithm works in general, before addressing in detail how to apply it to lexing. It describes the power of divide and conquer in terms of executing time and how laziness can be applied to these algorithms.

3.1.1 The Three Steps

The general idea of a divide and conquer algorithm is to divide a problem into smaller parts, solve them independently and then combine the results. A Divide and Conquer algorithm always consists of a pattern with the steps described in fig. 3.1 [7].

3.1.2 Associative Function

An associative function, or operator, is a function that does not care in what order it is applied. An example of such a function is addition $(+)$ of numbers, which is associative since it has the property in example 3.1.1, that is, $a + (b + c) = (a + b) + c$.

Divide: If the input size is bigger than the base case then divide the input into subproblems. Otherwise solve the problem using a straightforward method.
Recur: Solve the subproblems by recursively calling itself with each sub-problem as argument.
Conquer: Given the solutions to the subproblems, combine the results to solve the original problem.

Figure 3.1: The three steps of a Divide-and-Conquer algorithm

In divide and conquer algorithms working on sequences this is essential. In the divide step of the divide and conquer algorithm there is no certain order of how the subproblems are going to be divided. This means that the order the subproblems are being conquered can not have an impact on the algorithm, hence the conquer step must be associative.

Example 3.1.1 (Associativity of the conquer step)**.** Let $f(x,y)$ be the conquer function, where x and y are of the same type as the result of f, then:

$$f(x,f(y,z)) = f(f(x,y),z)$$

Otherwise the algorithm can give different results for different division of the input, but for the same data.

3.1.3 Time Complexity

To calculate the running time of any divide and conquer algorithm the master method can be applied [4]. This method is based on the following theorem.

Theorem 3.1.2 (Master Theorem [4], as described by [3])**.**
Assume a function T_n constrained by the recurrence

$$T_n = \alpha T_{\frac{n}{\beta}} + f(n)$$

(This is typically the equation for the running time of a divide and conquer algorithm, where α is the number of subproblems at each recursive step, n/β is the size of each subproblem, and $f(n)$ is the running time of dividing up the problem space into α parts, and combining the results of the subproblems together.)
If we let $e = \log_\beta \alpha$, then

1. $T_n = O(n^e)$ *if* $f(n) = O(n^{e-\epsilon})$ *and* $\epsilon > 0$
2. $T_n = \Theta(n^e \log n)$ *if* $f(n) = \Theta(n^e)$
3. $T_n = \Omega(f(n))$ *if* $f(n) = \Omega(n^{e+\epsilon})$ *and* $\epsilon > 0$ *and* $\alpha \cdot f(n/\beta) \leq c \cdot f(n)$ *where* $c < 1$ *and all sufficiently large* n

■

3.1.4 Hands on Example

The divide and conquer pattern can be performed on algorithms that solves different problems. A general problem is sorting, or more precisely sorting a sequence of integers. This example shows merge-sort [7].

Divide: The algorithm starts with the divide step. Given the input S the algorithm will check if the length of S is less then or equal to 1.

- If this is true, the sequence is returned. A sequence of one or zero elements is always sorted.
- If this is false, the sequence is split into two equally big sequences, S_1 and S_2. S_1 will be the first half of S while S_2 will be the second half.

Recur: The next step is to sort the subsequences S_1 and S_2. The sorting function sorts the subsequences by recursively calling itself twice with S_1 and S_2 as arguments respectively.

Conquer: Since S_1 and S_2 are sorted combining them into one sorted sequence is trivial. This process is what is referred to as merge in merge-sort. The resulting sequence of the merge is returned.

Algorithm 1 shows a more formal definition of merge-sort. The merging step is associative which ensures that the algorithm always returns the same result for the same input.

Algorithm 1: MergeSort

Data: Sequence of integers S containing n integers
Result: Sorted sequence S
if *length*$(S) \leq 1$ **then**
    return S
else
    $(S_1,S_2) \leftarrow splitAt(S,n/2)$
    $S_1 \leftarrow MergeSort(S_1)$
    $S_2 \leftarrow MergeSort(S_2)$
    $S \leftarrow Merge(S_1, S_2)$
    return S

Given the merge-sort algorithm, time complexity can be calculated as follows using the master method. There are 2 recursive calls and the subproblems are 1/2 of the original problem size, so $\alpha = 2$ and $\beta = 2$. To merge the two sorted subproblems the worst case is to check every element in the two list, $f(n) = 2 \cdot n/2 = n$.

$$T(n) = 2T(n/2) + n$$
$$e = \log_\beta \alpha = \log_2 2 = 1$$

Case 2 of the theorem 3.1.2 applies, since

$$f(n) = \Theta(n)$$

So the solution will be:

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

3.1.5 Incremental Computing

For an algorithm to be incremental means that when a point in the data source is updated, the algorithm only needs to update the directly effected path in the data source [15]. fig. 3.2 illustrate the updated nodes in a tree structured data source.

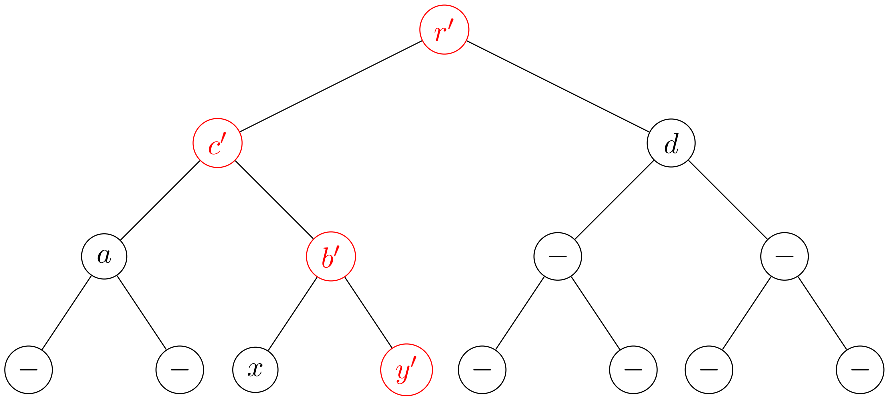

Figure 3.2: When the node y changed recomputed nodes are marked with a $'$.

For a divide and conquer lexer this means to only recompute the changed token and the token to the right of the changed token. This is done recursively 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 recalculating all the 1024 tokens the lexer only needs to do 10 recalculations, since $log_2 1024 = 10$. This can be explained by the theorem 3.1.2.

Only one branch in the tree will be followed at every level and the problem is already divided. Therefore the parameters will be set to:

$$\alpha = 1,\ \beta = 2 \text{ and } f(n) = 1.\ e = log_\beta \alpha = \log_2 1 = 0$$

Case 2 of the theorem 3.1.2 applies, since

$$f(n) = \Theta(n^e)$$

The complexity is therefore:

$$T(n) = \Theta(n^e \cdot \log n) = \Theta(\log n)$$

3.2 Fingertree

A fingertree is a tree structure that is built to make access to the beginning and end of a collection easy. In the following section fingertrees are explained. The code examples are simplified for demonstrative purposes, for instance in real implementations lists are not used since they do not give good performance for what fingertrees are designed to do.

3.2.1 Structure of Fingertrees

To achieve fast access to the beginning and end of the tree the leaves for the 1-4 first elements and 1-4 last elements are placed in the root of the tree, these are called fingers. The rest of the elements constitutes the spine which is another fingertree, with one difference, instead of having the first 1-4 elements the second level will instead have two 2-3 tree of depth 2 at the beginning and end of the tree. The third level will have 2-3 trees of depth 3 and for level n the 2-3 trees in the beginning and end will have depth n. An illustration of how a fingertree can look can be seen in fig. 3.3 [8].

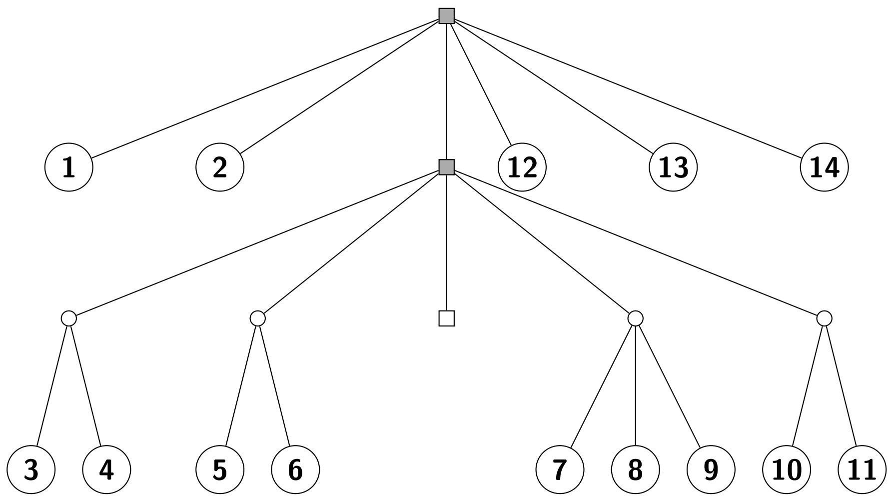

Figure 3.3: Illustration of Fingertree

As can be seen in fig. 3.4 there are three constructors for a fingertree, there are the trivial cases for an empty tree and for a tree with one element. The last constructor calls itself with $Node\ a$ instead of just a. This is what determines the depth of the 2-3 tree on each side of the spine. The reasons for the first level of the tree having fingers of size 1-4 is because of insertion and deletion which is covered in section 3.2.2 [8].

Accessing an element at place d in the tree will take $O(\log(min(d,n-d)))$ time. This is because the closer to the end of the fingertree the element is the closer the surface it is.

This in turn gives the time complexity of accessing an element, which in worst case is $O(\log n)$ and for the first and last element is $O(1)$ [8].

```haskell
data FingerTree a = Empty
                  | Single a
                  | Deep (Digit a) (FingerTree (Node a)) (Digit a)

type Digit a = [a]

data Node a = Node2 a a | Node3 a a a
```

Figure 3.4: Definition of the Fingertree data type [8]

3.2.2 Insertion and Deletion

The fingertree described so far can only handle insertion and deletion to the beginning and end of the tree.The fingers in a tree have two different states called dangerous and safe. The safe states is when there are 2 to 3 elements in the finger and when the finger is safe an insertion or deletion from the tree will not be anything more then an insertion or deletion in that finger [8].

When a finger has 1 or 4 elements the finger is called dangerous. In this case there might be implications down the spine. The first case is when there is an insertion into a finger that has 4 elements, in this case there are 5 elements that is assigned to the same level. If the insertion was done to the end of the tree the last 2 elements will be the new finger which is then a safe finger. The first 3 elements are used to create a new $Node3$ which is passed down the spine as a single element to be inserted at the next level. Inserting an element to the end of a tree can be seen in fig. 3.5, conversely adding an element to the beginning of a tree is the mirror to this function [8].

```haskell
(|>)                                :: FingerTree a -> a -> FingerTree a
Empty                (|>) a = Single a
Single b             (|>) a = Deep [b] Empty [a]
Deep pr m [e,d,c,b] (|>) a = Deep pr (m (|>) Node3 e d c) [b,a]
Deep pr m sf         (|>) a = Deep pr m (sf ++ [a])
```

Figure 3.5: Adding an element to the end of the sequence [8]

The other dangerous operation is when a deletion from a finger of size 1 is done. In this case, if the deletion is made to the end of the tree, the last element of the end finger in the spine is deleted and used as the finger for the level where the the finger would have been empty. Since the element in the finger below will either be a $Node2$ or a $Node3$ the new finger will be safe. Deletion of an element at the end or beginning of a tree is implemented similarly to *head* for lists, the functions *viewL* which returns the first element and the rest of the tree can be seen in fig. 3.6 [8].

```
viewL :: FingerTree a -> ViewL a
viewL Empty            = EmptyL
viewL (Single x)       = x :< Empty
viewL (Deep pr m sf) = head pr :< deepL (tail pr) m sf

deepL :: [a] -> FingerTree (Node a) -> Digit a -> FingerTree a
deepL [] m sf = case viewL m of
  EmptyL  -> toTree sf
  a :< m' -> Deep (nodeToDigit a) m' sf
deepL pr m sf = Deep pr m sf

data ViewL a = EmptyL
             | a :< FingerTree a
```

Figure 3.6: Adding an element to the end of the sequence [8]

Since the dangerous states propagate actions down the spine insertion and deletion will not take $O(1)$ time in the worst case. Since each new level will take at most $O(1)$ time the insertion or deletion of an element will in the worst case take $O(\log n)$ time. However since 3 out of 4 operations are safe for insertion and deletion respectively the expected time consumption for an insertion or deletion at the beginning or end of the tree will be $O(1)$. This is because each operation on a dangerous finger will render it safe for the next time it is accessed [8].

3.2.3 Concatenation of Fingertrees

When 2 fingertrees are concatenated there are a number of different cases which can occur. To begin with, when a concatenation of two trees is done a function called *app*3 is called with the two trees and an empty list of "between" elements. As can be seen in fig. 3.7 there are 4 trivial cases, the first two is when either tree is empty in which case the "between" elements is added to the nonempty tree. The other two are when there is exactly one element in one of the trees in which case that element is added to the other tree after the "between" elements [8].

In the last case two trees of more then one elements are concatenated. In this case, a new tree is created which has the first finger set as the first finger from the first tree and the last finger set as the last finger from the second tree. The spine will be created by calling *app*3 recursively with the spine from the first tree as the first tree, the last finger of the first tree plus "between" elements plus the first finger of the second tree as the new "between" elements and the spine from the second tree as the second tree [8].

```
(><) ::FingerTree a -> FingerTree a -> FingerTree a
xs (><) ys = app3 xs [] ys

app3 :: FingerTree a -> [a] -> FingerTree a -> FingerTree a
app3 Empty ts xs       = ts <|' xs
app3 xs ts Empty       = xs |>' ts
app3 (Single x) ts xs = x <| (ts <|' xs)
app3 xs ts (Single x) = (xs |>' ts) |> x
app3 (Deep pr1 m1 sf1) ts (Deep pr2 m2 sf2)
  = Deep pr1 (app3 m1 (nodes (sf1 ++ ts ++ pr2)) m2) sf2

(<|') :: [a] -> FingerTree a -> FingerTree a
(<|') = flip (foldr (<|))

(|>') :: FingerTree a -> [a] -> FingerTree a
(|>') = foldl (|>)
```

Figure 3.7: Concatenation function for Fingertree [8]

```
nodes :: [a] -> [Node a]
nodes [a, b]            = [Node2 a b]
nodes [a, b, c]         = [Node3 a b c]
nodes [a, b, c, d]      = [Node2 a b, Node2 c d]
nodes (a : b : c : xs) =  Node3 a b c : nodes xs
```

Figure 3.8: Help function for transforming a list of element into a list of Nodes [8]

The time complexity for concatenation can be reasoned as follows. As can be seen in fig. 3.7 the only operation that is run recursively is *nodes*, fig. 3.8. *nodes* will run in $O(1)$ time since the most amount of arguments passed to it will be 12 in which case 4 $Node3$ elements will be returned. Since each recursive step takes at most $O(1)$ time and the function terminates when the bottom of the shallower tree has been reached the total time to concatenate two trees is $\theta(\log(min(n,m)))$ where n and m is the size of the trees being concatenated [8].

3.2.4 Measurements

To make fingertrees useful for a divide and conquer lexer a measure of the tree needs to be added. A measure of a tree may for example be how many elements is in the tree or as will be shown later in the report, the lexed tokens of the text in a tree. To implement measures time efficiently the data type that is chosen should be a monoid. A monoid is in abstract algebra a set S and an operator (<>) which satisfies the rules in fig. 3.9 [8].

Closure $\forall a,b \in S : a <> b \in S$
Associativity $\forall a,b,c \in S : (a <> b) <> c = a <> (b <> c)$
Identity element $\exists e \in S : \forall a \in S : e <> a = a <> e = a$

Figure 3.9: Monoid rules over a set S with operator <>

An example of a monoid is the natural numbers, which under addition form a monoid where the identity element is 0. Using Haskells class system measures are defined as in fig. 3.10. *mempty* will henceforth be used as the identity element in examples and definitions.

```haskell
class (Monoid v) => Measured a v where
  measure :: a -> v
```

Figure 3.10: Definition of the Measure class [8]

Since the measure is a monoid, when two trees are concatenated the measure of the new tree is simply *measure* $tr1$ <> *measure* $tr2$. The *Digit* data type in the trees are always of constant size, however the elements in *Digit* are of type *Node a* which grows with the depth of the tree. Because of this the *Node a* data type is also measured as can be seen in fig. 3.11 [8].

```haskell
instance (Measured a v) => Measured (Digit a) v where
  measure xs = foldl (\v x -> v <> measure x) mempty xs

data Node v a = Node2 v a a | Node3 v a a a

node2 :: (Measured a v) => a -> a -> Node v a
node2 a b = Node2 (measure a <> measure b) a b

node3 :: (Measured a v) => a -> a -> a -> Node v a
node3 a b c = Node3 (measure a <> measure b <> measure c) a b c

instance (Monoid v) => Measured (Node v a) v where
  measure (Node2 v _ _) = v
  measure (Node3 v _ _ _) = v
```

Figure 3.11: Measure of the data type Node [8]

Figure 3.12 shows the fingertree implementation with measures which is similar to the implementation without measures, fig. 3.4. The new data type, *deep*, is used as the constructor for trees with more then one element. Because of how fingertrees are implemented the type of the elements will change, in the beginning it is a at the second level it is *node v a*. However the measure will always be of type v [8].

```
data FingerTree v a = Empty
| Single a
| Deep v (Digit a) (FingerTree v (Node v a)) (Digit a)

deep :: (Measured a v) =>
  Digit a -> FingerTree v (Node v a) -> Digit a -> FingerTree v a
deep pr m sf = Deep (measure pr <> measure m <> measure sf) pr m
    sf

instance (Measured a v) => Measured (FingerTree v a) v where
  measure Empty       = mempty
  measure (Single x) = measure x
  measure (Deep v)    = v
```

Figure 3.12: Fingertrees Measure function [8]

Fingertrees offers a data structure where the time complexity for operations on the tree scales logarithmically with the size of the tree. Because of this and the fact that operations on the measure in the fingertree has constant time it makes the data structure suitable for a divide and conquer lexer [8].

3.3 Divide and Conquer Lexing in General

In section 3.1 the general divide and conquer algorithm was covered. This section covers the general data structures and algorithms for an incremental divide and conquer lexer.

3.3.1 Tree structure

The incremental divide and conquer lexer should use a structure where the code-lexemes can be related to its tokens, current result can be saved and easily recalculated. A divide and conquer lexer should therefore use a tree structure to save the lexed result in. Since every problem can be divided into several subproblems, until the base case is reached. This is clearly a tree structure of solutions, where a leaf is a token for a single character, and the root is a sequence of all tokens in the code.

3.3.2 Transition map

When storing a result of a lexed string it is a good idea to store more then just the tokens. In particular the in and out states are needed when combining the lexed string with another string. The information needed can be bound to a type synonym like in fig. 3.13. This report will henceforth refer to this as a *transition*.

Since the lexer does not know if the current string is a prefix of the entire code or not it can not make any assumptions on the in state. Because of this the lexer needs to store a transition for every possible in state, fig. 3.13. the report will henceforth refer to this as a *transition map*.

```
type Transition = (State ,[Token] ,State)
type transitionMap = [Transition]
```

Figure 3.13: Type synonyms for the transition map

The Base Case

When the lexer tries to lex one character it will create a transition map using the DFA for the language. It will, for each state, create a transition that has the state as in state, a list containing the character as the only token and by using the DFA, lookup what out state the transition should have. For the character 'o' part of a transition map might look like fig. 3.14.

In fig. 3.14, fig. 3.16 and fig. 3.17 the first number refers to the in state, the middle part is the sequence of tokens and the second number is the out state, that can be accepting.

$$\begin{bmatrix} 0 & ['o'] & Accepting5 \\ 1 & ['o'] & 1 \\ 10 & & NoState \end{bmatrix}$$

Figure 3.14: The Base Case for divide and conquer lexing

$NoState$ transition is used to tell the lexer that using that particular transition will result in a lexical error. For reasons being covered later in this section, they can not be discarded.

Conquer Step

The conquer step of the algorithm is to combine two transition maps into one transition map. This is done by, for every transition in the left transition map, combining the transition with the transition in the right transition map that has the same in state as the left transitions out state. This can be described by the function in fig. 3.15 where $map1$ and $map2$ refers to the first and second transition map.

```
merge :: transitionMap -> transitionMap -> transitionMap
merge map1 map2 = [(i,t1><t2,o) | (i,t1,o1) <- map1, (i2,t2,o) <-
    map2, o1==i2]
```

Figure 3.15: Function for merging two transition maps into one transition map

The most general case is a naive lexer that takes the first accepting state it can find. When two transitions are combined there are two different outcomes:

Append: If the out state of the first transition is accepting, the sequence in the transition that starts in the starting state of the second transition map will be appended to the first.

```
appendTokens :: Tokens -> Tokens -> Tokens
appendTokens tokens1 tokens2 = tokens1 >< tokens2
```

Merge: If the out state of the first transition is not accepting, the transition in the second transition map with the same in state as the out state of the first transition will be used. The last token of the sequence from the first transition will be merged with the first token in the second transition into one token and put between the two sequences.

```
mergeTokens :: Tokens -> Tokens -> Tokens
mergeTokens tokens1 tokens2 = prefix1 |> newToken >< suffix2
  where prefix1 |> token1 = tokens1
        token2 <| suffix2 = tokens2
        newToken          = token1 `combinedWith` token2
```

For both the cases the in state of the first transition will be the new in state and the out state of the second transition will be the new out state. An example of both cases is shown in fig. 3.16.

$$\begin{bmatrix} 0 & ['o'] & Accepting5 \\ 1 & ['o'] & 1 \end{bmatrix} \text{`}combineTokens\text{`} \begin{bmatrix} 0 & ['\ '] & Accepting2 \\ 1 & ['\ '] & 1 \end{bmatrix} = \begin{bmatrix} 0 & ['o','\ '] & Accepting2 \\ 1 & ['o\ '] & 1 \end{bmatrix}$$

Figure 3.16: The Conquer step for Divide and Conquer lexing

This will not work as a lexer for most languages since the longest match rule is not implemented. For example, 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.

Longest Match

To ensure that only the longest token is returned some stricter rules for combinations are needed. Firstly, if two transitions can be combined without having the outgoing state $NoState$ then *merge* those transition. When two transitions are merged the last token of the left transition is merged with the first token of the right transition into one token. Secondly, If the combination of two transitions would yield $NoState$, the transitions are *appended* instead. When two transitions are appended the right transition starting from the starting state is appended to the left transition. As can be seen in fig. 3.17

$$\begin{bmatrix} 0 & ['o',' \ '] & Accepting2 \\ 1 & ['o \ '] & 1 \end{bmatrix} `combineTokens` \begin{bmatrix} 0 & ['/','*'] & Accepting4 \\ 1 & ['*/'] & Accepting3 \\ 2 & & NoState \end{bmatrix} = \begin{bmatrix} 0 & ['o',' \ ','*','/'] & Accepting4 \\ 1 & ['o \ */'] & Accepting3 \end{bmatrix}$$

Figure 3.17: The Conquer step when the longest match rule is applied

When two transitions are appended another rule needs to be accounted for. If the last token of the first transition does not end in an accepting state a lexical error is found. How lexical errors are handled and stored is explained in section 3.3.3.

3.3.3 Lexical Errors

Even though lexical errors can not halt the lexer it is still useful to keep them since they tell the user what is wrong. In an incremental lexer there are different ways this can be achieved. The simplest way is to store the lexical error, instead of the tokens and outgoing state, when an error is encountered. To use this method the transition need to be modified to store the error, see fig. 3.18. The advantage of this is that when a lexical error is encountered nothing more will be computed for that transition, however all other transitions in the transition map will be computed as normal. If this style is used in a text editor and a lexical error is encountered, the user will only get feedback from that error.

```
type Transition = (State, Either ([Token], State) Error)
```

Figure 3.18: Transition that can either contain tokens or a lexical error

Another way is to keep as much of the correct tokens as possible and only store errors for the lexeme that does not match anything else. With this approach the lexer would store all tokens up until a lexical error is encountered. When an error is encountered, the error is stored and the lexer tries to lex the rest of the text starting from the starting state. For this to work the sequence that stores the tokens needs to store the lexical errors as well, see fig. 3.19. With this approach the lexer will continue combining tokens after a lexical error is found, the drawback with this is that extra token computations needs to be made that may not be useful in the final lexical analysis. If this approach is used in a text editor the user will see the minimal combination of characters that construct a lexical error. After that error, tokens that are lexed from the starting state is returned.

```
type Transition = (State,[Either Token Error],State)
```

Figure 3.19: Transition contains a sequence of tokens and errors

Example 3.3.1 (A Java lette light lexer, see appendix A)**.** Lexical analysis is done on the string “Hello /*World”. When global error handling, the transition contains one error or a sequence of tokens, is used the result of the lexical analysis will be as in fig. 3.20(a). When local error handling, the transition contains a sequence of tokens and errors, is used the result of the lexical analysis will be as in fig. 3.20(b).

String	Type
'/'	*Error*

(a) Global Error

String	Type
'Hello'	*Ident*
' '	*Space*
'/'	*Error*
'*'	*Reserved*
'World'	*Ident*

(b) local Error

Figure 3.20: Difference in error handling

If local error handling is used and the comment in the string would later on be closed, the tokens after '/' would be thrown away since another transition would be used which constructs a multi line comment. If global error handling is used the user will get little to no use of the lexical analysis until the lexical error is corrected, however run time is saved since nothing is computed after the lexical error is found for that transition.

3.3.4 Expected Time Complexity

Incremental computing states 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 recalculate every node on this path. As shown by fig. 3.2. Only one subproblem is updated in every level of the tree. Using the master method to calculate the expected time complexity gives: $e = \log_b a$ 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 fig. 3.2, the number of needed update calls at every level of the tree is 1, therefore $a = 1$. The constant b is still 2. This will give $e = \log_2 1 = 0$. Thus the update function of the incremental algorithm will have an expected time complexity of $\Theta(n^0 \cdot \log n) = \Theta(\log n)$

Since the fingertree is lazy, when an element is added to the root level of the tree, root elements might be pushed down in the tree. The measure of the lower levels does not need to be immediately recalculated. Instead they are recalculated when they are used. Paying for this expensive operation like described in the section about bankers method [8].

4

Implementation

In this chapter the tools, data structure and implementation of the incremental divide and conquer lexer is explained. The implementation of the incremental divide and conquer lexer uses fingertrees for storing the intermediate tokens and the lexed text. It has an internal representation of the tokens to keep track of the data needed when two fingertrees are combined. The lexical routines for combining the internal token data type take advantage of functional composition in order to get lazy updating of the tokens when two fingertrees are combined. The complete implementation can be found in appendix B.

4.1 The DFA Design

The DFA used in the incremental lexer was created using Alex. Alex is a Haskell tool for generating lexical analyzers given a description of the language in the form of regular expressions, it is similar to lex and flex in C and C++. The resulting lexer is Haskell 98 compatible and can easily be used with the parser Happy, a parser generator for Haskell [5]. Alex is notably used in BNFC which is a program to generate among other things a lexer, parser and abstract syntax from Backus-Naur Form [6].

The reason for using Alex to generate the DFA is that it optimizes the number of elements in the transition table. Instead of having an array for every possible character and state combination, 5 arrays are generated that takes advantage of the fact that for most characters the same state will be used the majority of time. This saves a lot of elements that would otherwise be the same in the array.

The trade off for using the Alex generated DFA is that some minor arithmetic operations are used and some extra lookups are needed. These operations are far less time consuming then the rest of the lexical operations.

4.2 Token data structure

To keep all the information that might be needed when combining two texts, a data structure for the tokens was created. This data type contains more information about the last token than what a sequential lexer would save, exactly what is explained in section 4.2.2.

Since this project is about creating a real-time lexing tool, performance is important. Therefore there are advantages of using sequences instead of lists, since they have better time complexity. The most notable place where this is used is in the measure of the fingertree, where the tokens are stored in a sequence rather then a list. Sequences are also used elsewhere in the project but the measure is the most notable place since it is frequently updated.

4.2.1 Tokens

The internal structure used to store lexed tokens is called *Tokens*. There are three constructors in the *Tokens* data type, see fig. 4.1.

```
data Tokens     = NoTokens
                | InvalidTokens (Seq Char)
                | Tokens { currentSeq :: (Seq Token)
                         , lastToken  :: Suffix
                         , outState   :: State}
```

Figure 4.1: Tokens Data Type

NoTokens is a representation of when an empty string has been lexed. *InvalidTokens* represents a lexical error somewhere in the text that was lexed, the sequence of characters is the lexical error or last token lexed. The *Tokens* constructor is the case when legal tokens have been found. *currentSeq* are all the currently lexed tokens save for the last, *lastToken* are all the possible ways that the last token can be lexed, in this implementation this is referred to as the suffix and what it is and why it is needed will be explained next.

4.2.2 Suffix

When a text is lexed it is uncertain that the last token is the actual end of the file since it may be combined with something else. To ensure that all possible outcomes will be handled the last token can take one of three different forms. The part of the text lexed can end in:

- a state that is not accepting,
- an accepting state,
- a state that is not accepting, but the text can also be a sequence of multiple tokens.

To keep track of these cases a data structure that captures them was implemented, see fig. 4.2.

```
data Suffix    = Str (Seq Char)
               | One Token
               | Multi Tokens
```

Figure 4.2: Suffix Data Type

The *Str* constructor is used to keep track of partially complete tokens, an example of this is when a string is started but the end quotation character have not yet been found.

The *One* constructor is used when exactly one token has been found, it may or may not be the token that is used in the final result of the lexing. Since this constructor is a special case of the *Multi* constructor it can be omitted. However the *One* constructor makes certain cases redundant since the lexer makes assumptions that can not be made for the *Multi* constructor.

The *Multi* constructor is used when at least one token has been found but the lexeme for the suffix does not match exactly one token. The entire suffix still needs to have an out state. This type of suffix can typically be found when the beginning of a comment is lexed. for example the text */*hello world* would be lexed to a sequence of complete tokens, "/", "*", "hello" and "world", but the lexer still needs to keep track of the fact that it may be in the middle of a multi-line comment. Note that in this case the *Tokens* data structure would have one out state, the state for the middle of a comment, and the suffix would have another, the end of an ident.

4.3 Transition Map

The transition map is a function from an in state to *Tokens*. As shown in fig. 4.1 the *Tokens* data type contains the out state.

```haskell
type State = Int

type Transition = State -> Tokens

getTokens :: Transition -> State -> Tokens
getTokens trans state = trans state
```

Figure 4.3: Transition Data Type

This data type is used in the lexical routines. The reason for using transition maps is that the lexer does not know what the in state for a lexed text is, hence the tokens for all possible in states must be stored. The transition map can be implemented in two ways, a table format and a function composition format.

The table format uses an array to store the currently lexed tokens where the index of the array represents the in state for that sequence of tokens. This is useful when the tokens need to be stored since it ensures that the tokens are computed.

When combining lexed tokens it is useful to use functional composition since it ensures that no unnecessary states will be computed. The drawback is that it does not guarantee that the actual tokens are computed which may result in slow performance at a later stage in the lexing. Since Haskell does not evaluate functional composition to $(f.g)x$ but rather $f(g\ x)$ all incrementality will be lost with this data structure.

Both these representations are used in the incremental divide and conquer lexer. The table format is used when storing the tokens in the fingertree to allow for fast access and incrementality. The function composition is used when combining tokens to ensure that only needed data is computed.

4.4 Fingertree

The fingertree is constructed with the characters of a text being the leaves and with the table format transition map as it is measure. The *Table* data type has to be a monoid in order to be a legal measure of the fingertree.

```haskell
type LexTree  = FingerTree Table Char

type Table = Array State Tokens
```

Figure 4.4: The data type for storing the tokens and text

The monoid class in Haskell has two different functions, *mempty* which is the identity element and *mappend* which is an associative operator that describes how two elements are combined. As can be seen in fig. 4.5, *mempty* creates an array filled of empty *Tokens*. *mappend* extracts the functions from the old tables, combines them using *combineTokens* then creates a new table filled with the combination.

```
tabulate :: (State,State) -> Transition -> Table
access :: Table -> Transition

tabulate range f = listArray range [f i | i <- [fst range..snd
    range]]
access a x = a ! x

instance Monoid Table where
  mempty = tabulate stateRange (\_ -> emptyTokens)
  f `mappend` g = tabulate stateRange $ combineTokens (access f)
      (access g)
```

Figure 4.5: The tabulate functions and monoid implementation

There are two helper functions that convert between the table format that is stored as the measure and the function composition format that is used in the lexical routines. These can be seen in fig. 4.5.

4.5 Lexical routines

The lexical routines are divided into five functions. They each handle different parts of the lexical steps that are needed in an incremental divide and conquer lexer.

4.5.1 Combination of Tokens

```
combineTokens :: Transition -> Transition -> Transition
combineTokens trans1 trans2 in_state
     | isInvalid toks1 = toks1
     | isEmpty toks1   = trans2 in_state
     | otherwise = combineWithRHS toks1 trans2
  where toks1 = getTokens trans1 in_state
```

Figure 4.6: The *combineTokens* function

combineTokens is the function called when two fingertrees are combined. The function starts by checking if the tokens generated from *in_state* from the first transition is empty or invalid in which case the output is trivial. If the tokens generated are valid, the tokens are passed on to *combineWithRHS* together with the second transition.

4.5.2 Combine Tokens With Right Hand Side

combineWithRHS checks how the tokens from the first transition are to be combined with the second transition.

```
combineWithRHS :: Tokens -> Transition -> Tokens
combineWithRHS toks1 trans2 | isEmpty toks2 = toks1
                            | isValid toks2 =
    let toks2' = mergeTokens (lastToken toks1) toks2 trans2
    in appendTokens seq1 toks2'
                            | otherwise = case lastToken toks1 of
    Multi suffToks ->
      let toks2' = combineWithRHS suffToks trans2
      in appendTokens seq1 toks2'
    One tok -> appendTokens (seq1 |> tok) (getTokens trans2
        startState)
    Str s -> invalidTokens s
  where toks2 = getTokens trans2 (outState toks1)
        seq1 = currentSeq toks1
```

Figure 4.7: *CombineWithRHS* function

combineWithRHS starts by creating tokens from the second transition, $toks2$, using the out state from the first tokens, this can result in three different cases, the definition of the variable names can be found in fig. 4.7.

isEmpty If $toks2$ is empty $toks1$ is returned.

isValid If $toks2$ is valid it means that the last token from the $toks1$ can be combined into one token with the first token in $toks2$.

otherwise If $toks2$ is not valid the lexer checks the suffix of $toks1$ to see if it ends in an accepting state or a valid state.

- if the *One* constructor is found the suffix ends in an accepting state which means that tokens created from the start state can be appended to $toks1$.
- If the *Multi* constructor is found the tokens from the suffix, $suffToks$, is extracted and a recursive call to *combineWithRHS* is made with $suffToks$ as argument instead.
- If the *Str* constructor is found the suffix does not end in a valid state and *InvalidTokens* will be returned.

4.5.3 Merge Two Tokens

mergeTokens combines the last token from the first tokens with the first token of the second tokens, for the code see fig. 4.8.

```
mergeTokens :: Suffix -> Tokens -> Transition -> Tokens
mergeTokens suff1 toks2 trans2 = case viewl (currentSeq toks2) of
  token2 :< seq2' -> let newToken = mergeToken suff1 token2
                     in toks2 {currentSeq = newToken <| seq2'}
  EmptyL -> case alex_accept ! out_state of
    [] -> let newSuff = mergeSuff suff1 (lastToken toks2) trans2
          in toks2 {lastToken = newSuff}
    acc -> let lex = suffToStr suff1 <>
                     suffToStr (lastToken toks2)
           in toks2 {lastToken = One $ createToken lex acc}
  where out_state = outState toks2
```

Figure 4.8: *MergeTokens* function

- If there are more then one token in *toks*2, *suff*1 is combined into one token with the first token in *toks*2 and the rest of the tokens in *toks*2 is appended and returned.
- If there is exactly one token in *toks*2, the suffix from *toks*2 is combined with *suff*1. When two suffixes are combined some extra checks are needed. If *toks*2 has an accepting out state, the two suffixes can be combined into one token. If *toks*2 does not have an accepting out state the work is passed on to *mergeSuff*.

4.5.4 Merging Suffixes

mergeSuff checks which pairs of suffixes it has and takes the appropriate actions.

- If the first suffix is of type *Multi* the function calls *combineWithRHS*. If the resulting tokens is invalid a recursive call is made with the suffix from the new tokens as first suffix.
- If the first suffix is of type *Str* the result will always be another *Str* no matter what is in the second suffix so the string is extracted and appended.
- if the first suffix is of type *One* and the second *Str* a new *Multi* suffix is created. A new second tokens is created using the start state on the second suffix, if this results in a valid *Tokens*, the token from the first suffix is prepended. If it is not valid the *Str* is just added to the end of the new suffix.
- When both suffix are *One* they can be combined into a single token.
- When the first suffix is *One* and the second is *Multi* it is passed onto *mergeTokens*.

```
mergeSuff :: Suffix -> Suffix -> Transition -> Suffix
mergeSuff (Multi toks1) suff2 trans2 = Multi $
  let newToks = combineWithRHS toks1 trans2
  in if isValid newToks
     then newToks
     else let newSuff = mergeSuff (lastToken toks1) suff2 trans2
          in toks1 {lastToken = newSuff}
mergeSuff (Str s1) suff2 _ = Str $ s1 <> suffToStr suff2
mergeSuff (One token1) (Str s) trans2 =
  let toks2 = getTokens trans2 startState
  in if isValid toks2
     then Multi $ toks2 {currentSeq = token1 <| currentSeq toks2}
     else Multi $ createTokens (singleton token1) (Str s) (-1)
mergeSuff suff1 (One token2) _ = One $ mergeToken suff1 token2
mergeSuff suff1 (Multi toks2) trans2 =
  Multi $ mergeTokens suff1 toks2 trans2
```

Figure 4.9: $MergeSuff$ function

4.5.5 Append to Sequence of Tokens

appendTokens checks if there is a lexical error in $toks2$. if there is an error, that error is returned, otherwise $toks2$ is appended to $toks1$.

```
appendTokens :: Seq IntToken -> Tokens -> Tokens
appendTokens seq1 toks2 | isValid toks2 =
  toks2 {currentSeq = seq1 <> currentSeq toks2}
                        | otherwise = toks2
```

5

Result

The incremental lexer has three requirements, it should be robust, efficient and precise. Robustness means that the lexer does not crash when it encounters an error in the syntax. That is, if a string would yield an error when lexed from the starting state the lexer does not return that error but instead stores the error and lexes the rest of the possible input states since the current string might not be at the start of the text. The implementation this report propose is robust since it stores errors in the data structure rather then returning an error.

For it to be efficient the feedback to the user must be fast enough, or more formally the combination of two strings should be handled in $\Theta(log(n))$ time.

Finally to be precise the lexer must give a correct result. This chapter is describing how these requirements are tested and what the results are.

In the sections below, any mention of a sequential lexer refers to a lexer generated by Alex using the same Alex file as was used when creating the incremental lexer [5]. The reason why Alex was used was because the DFA generated by Alex was used in the incremental lexer, thus ensuring that only the lexical routines differs.

5.1 Preciseness

For an incremental lexer to work, the lexer must be able to do lexical analysis of any part of a text and be able to combine two partial texts. If the lexical analysis of one partial text does not result in any legal token it must be able to be combined with other partial texts that makes it legal tokens. The lexical analysis of a text might not always result in the same tokens than the combination of the text with another text would give.

To test these cases a test was constructed which did a lexical analysis on two partial texts using the incremental lexer and then combining the results into one text. The result of the combination should be the same as the lexical analysis of the entire text using the incremental lexer and the result using a sequential lexer.

It is not enough to test if the combination of two partial texts yields the same sequence of tokens as the text. To test that the result of the incremental lexer was the correct sequence of tokens, it was compared to what a sequential lexer generated. This comparison was an equality test that compared token for token that they were the same kind of token and had the same lexeme.

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) tokTupple
```

notEqual function is a function which pattern-match on the two 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.

Tests were performed on different files that were cut in different places when the update was tested. In all the tests the incremental lexer produced the same tokens as the sequential lexer. When these tests were done no text that would produce a lexical error was used. Some partial texts did produce lexical errors but the texts passed through the sequential lexer and compared to the result of the incremental lexer did not produce lexical errors.

5.2 Performance

All tests, where time performance was measured, were done with the help of Criterion, a Haskell library. Criterion has tools to ensure that the functions being tested is evaluated to normal form. Criterion runs the tests 100 times with a warm up run by default. The warm up run makes sure that all inputs to the functions being tested are evaluated before the actual testing begins, this ensures that nothing but the function being tested is measured [10].

To make sure that the time performance tests were not skewed under a certain system they were tested on different hardware and operating systems. The results were similiar on all the systems tested. The results presented below were done on an intel i5 quad core at 2900MHz with 8GB memory under the linux Red Hat operating system.

To measure the performance of the incremental step two fingertrees were created, each representing one half of a text. By creating the two fingertrees the transition map for the code in those trees are created as well. The benchmarking was then done on the combination of the two trees. The results of the incremental lexer benchmarking suggested a running time of $\Theta(log(n))$. To get a reference point the same text was lexed using a sequential lexer. The benchmarks can be found in fig. 5.3 and fig. 5.2.

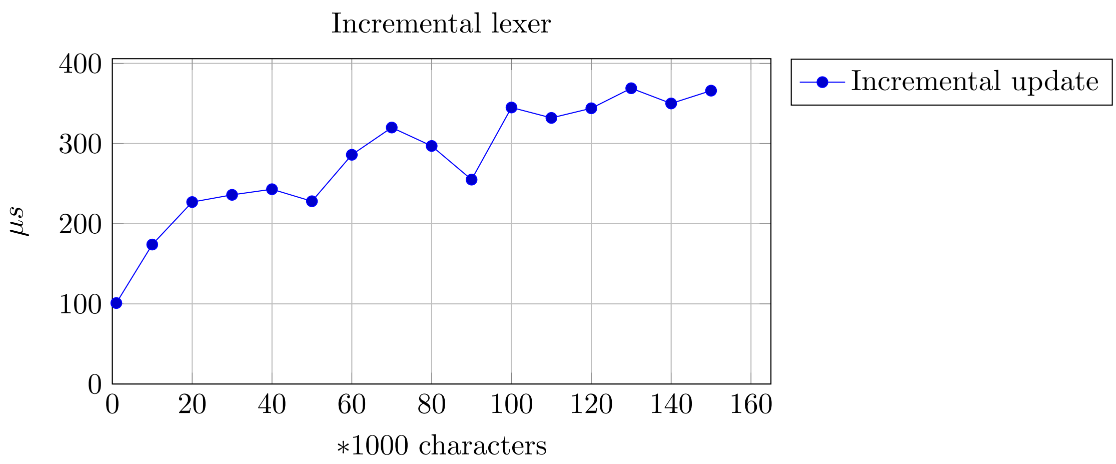

Figure 5.2: Benchmarking times of an incremental update

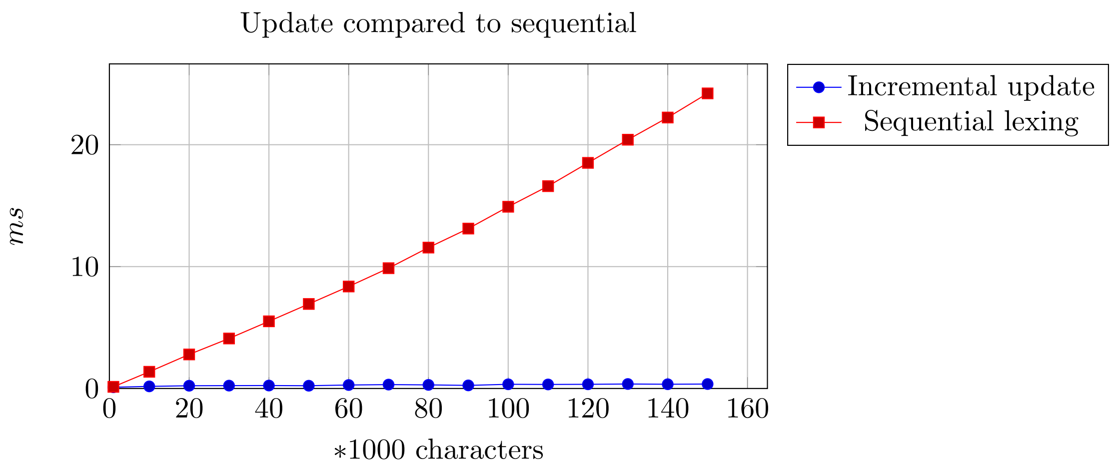

Figure 5.3: Comparison between an incremental update and sequential lexer

The running time when constructing the tree was not as fast as either the update or the sequential lexer. The tests for the running time, when constructing a new tree, sugged $\Theta(n \log(n))$, this was expected since there are $\Theta(\log(n))$ updates for every character. The result can be found in fig. 5.4.

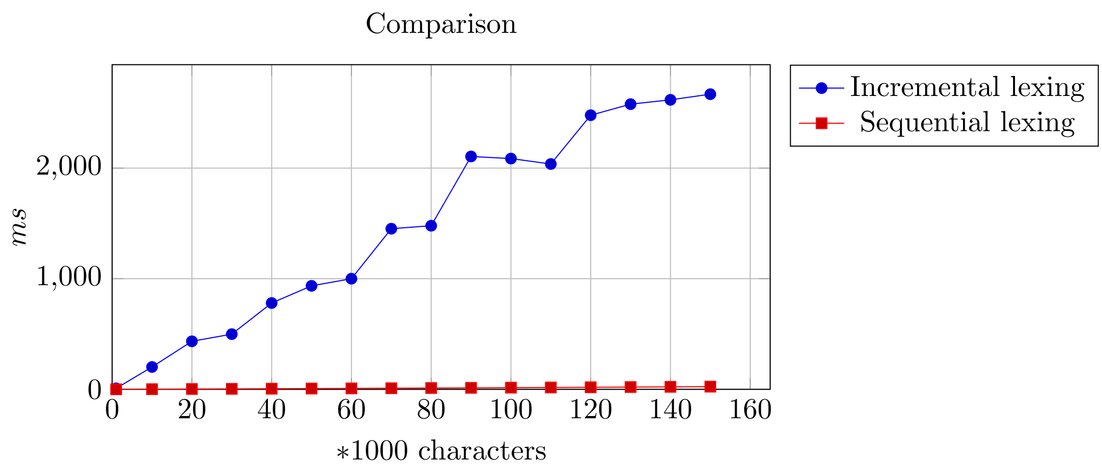

Figure 5.4: Comparison between the sequential lexer and the incremental lexer when lexing an entire file

The space a fingertree takes is dependent on the size of the measurement of the tree. In the case of the incremental lexer the measurement of the tree is the transition map. To test how much space the transition map takes a DFA for an early Java version that has 90 states was used. The transition map was serialized and stored to the disc using the Haskell library *Data.Binary*. The test suggested that the size of the transition map grows linearly with the size of text being lexed, the results can be found in fig. 5.5. Because of how the transition map is constructed it will also grow linearly with the number of states in the DFA. The test shows that the transition map has space complexity $\Theta(mn)$.

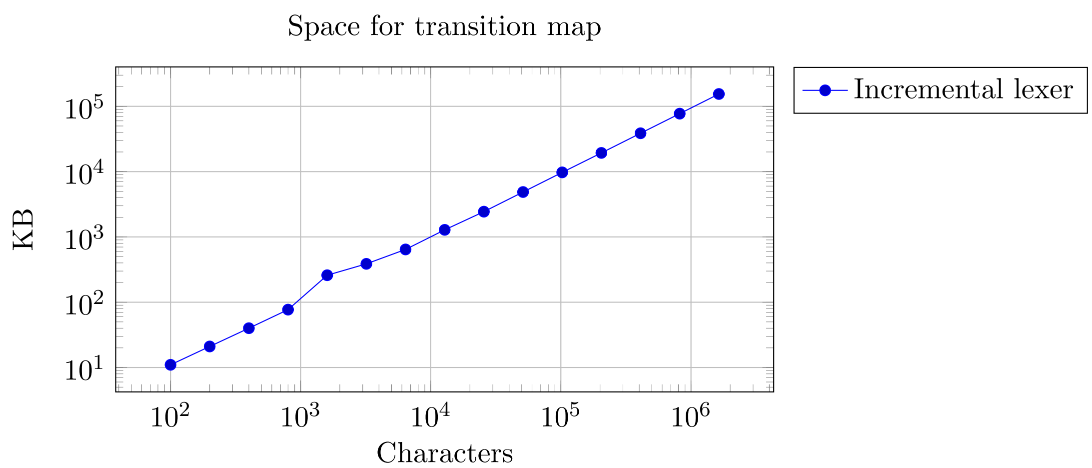

Figure 5.5: Space usage of the transition map using a DFA with 90 states

Our conclussions on the space complexity is based on the assumption that there is no data sharing for our measurement between levels in the fingertree

To measure the space of an entire fingertree each level of the tree must be regarded. As shown in fig. 3.3 the n:th level of a fingertree has two $2-3$ *trees* of depth n with leaves in them. The root level will therefore have a measure for all leaves stored, that is if the measure takes $\Theta(f(n))$ space, the measure in the root will take that much space. For the second level two $2-3$ *trees* of depth 1 has been removed, so the measure will be $\Theta(f(n-2*2))$ in the worst case. For the third level two more $2-3$ *trees* have been removed and in general on the x:th level the measure will take:

$$\Theta(f(n - 2 * \sum_{y=1}^{x} 2^y))$$

Since the measure from all the levels in the tree are stored, the total amount of data the measures takes is a sum over all the levels, the resulting approximation can be seen in fig. 5.6. For the entire equation see appendix C.

$$\sum_{x=0}^{\log(n-1)} (n - 2 \cdot \sum_{y=1}^{x} 2^y) = (n+4)\log(n-1) - 7n + 16 \Rightarrow \Theta(n \log n)$$

Figure 5.6: The number of characters being measured in a tree with n characters

Because of time limitations the size of the fingertrees were never tested. If the approximation of the space needed for a fingertree is correct the space complexity for the trees generated by this lexer will be $\Theta(mn \log n)$.

6

Discussion

During the course of the project there were some setbacks. The first setback was that our initial solution had bad running time, was hard to understand and did not handle longest matching correctly.

After the first solution came a solution that was easier to understand but still had problem with the longest match. The running time was greatly improved from the first version but was still not faster then sequential lexers.

To solve this our last implementation which is described in chapter 4 made use of arrays and a DFA from Alex. The longest match problems that existed in the earlier versions of the lexer was mainly due to difficulty finding the correct solution.

6.1 Used Programming Language and Data Structure

The project is written in Haskell. One of the reasons is that similar research and projects have been done in Haskell, for instance [11] and [8]. Haskell also has the tools and the data structures used in the project. For instance Alex was used in two parts, first the DFA generated was used and second a lexer generated by Alex was used to get a comparison of the time for lexing.

There are other advantages of using Haskell as implementation language, namely higher order functions, lazy evaluation and function composition. Functional composition is useful in the lexical routines since the transitions are implemented as functions and are more or less just evaluated by composing. The lexical routines can be implemented as lazy, however this makes the incremental step in the lexer ineffective. Because of this the lexical routines are implemented strict, that is all values are always calculated.

The project could have been done in other languages. There are for example lexer generators in other languages, lex, Flex and Jlex, which can be used to create an efficient DFA. There have also been earlier articles that handle problems similar to this project written in Java [9].

The advantage of using fingertrees in an incremental lexer is that it is easy to keep track of which tokens correspond to which part of the text, since the tokens are in the measure of the tree. Fingertrees also has the advantage of keeping track of earlier result. When a tree is split up the tokens that match the partial texts are already calculated. The time complexity for combining two trees is low, O(log n), where n is the size of the smallest tree. The lexical routines revolve around combining two lexical results. Because of this it is advantageous to use fingertrees since combination of two fingertrees are fast. The main reason to not use any other type of tree is that another tree would not keep track of measure for the partial trees. In the case of splitting a tree the tokens would have to be recalculated.

6.2 Trials and Errors

The first version of the lexical routines only had the goal to get an idea of how a divide and conquer lexer could be implemented. As a result a lot of unnecessary information was stored and computed and some necessary information was stored and computed more than once. The solution was to calculate uncertain tokens until a satisfactory result was found or all possibilities was exhausted. This meant that for each combination the worst case was $O(2^n)$.

The next step was to make sure that the information was stored in the right places and that no unnecessary information was stored. To solve this an overhaul of the projects data structure was made. The result is close to the structure used in the final result. This solution still had some problem with the running time. The main reason was that the lexical result was not explicitly stored in the fingertrees. That is, the functional composition was stored in the finger trees and since Haskell is lazy in nature the result was not computed resulting in slow running time when combining trees.

The final version of the project ensured that the lexer ran fast and made sure that fringe cases were computed correctly. The solution to this was to use arrays in the measure of the fingertrees since arrays are always explicitly evaluated to normal form and that they have quick lookup time $O(1)$. Since the lexical routines can take advantage of the transitions in the form of functions that representation was used in the lexical routines and functions for converting between the arrays and function representation was implemented.

6.3 Implementation Suggestions

The updating step of the lexer is fast since the only computation needed is the combination of the last token of the first tree with the first token of the second tree and the combination of the actual trees. If an incremental lexer is used the fingertrees should be stored instead of the raw text. If the fingertrees are not stored the fingertree would need to be recalculated each time the file was opened.

7

Conclusion and Future Work

As mentioned in the result chapter the incremental lexer was both robust and precise. This means that without considering the time and space efficiency an incremental lexer will produce the same result as a sequential lexer with the difference being how lexical errors are handled. The incremental lexer is efficient in the sense that updates are done in $\Theta \log(n)$ time. However when a tree is built up from scratch the incremental lexer takes $\Theta n \log(n)$ time compared to the sequential lexer that takes Θn time.

7.1 Conclusions

Incremental lexers are not suited to be used in a stand alone lexer since a sequential lexer is more efficient then an incremental lexer when an entire text is being lexed. If a development environment that uses an incremental lexer was used, the stand alone lexer can be omitted since the tokens are already generated, saving one step in the compilation process.

It is however suited in an environment where updates are likely to happen, for example to give lexical feedback in a text editor where each key stroke would be an update. Insertion of a character in the text will be faster with an incremental lexer compared to a sequential lexer since the lexical analysis does not need to be done on the entire text. This means that the lexer could be run in real time without a user noticing it. The result from an incremental lexer can be passed to an incremental parser, giving parsing feedback to the user instead. However, loading times when opening files will be longer if the tree containing the tokens are not stored.

The space requirements for the incremental lexer grows with the tree. There is information for the entire text in all levels of the tree and each level has information for all possible in states. The space of a tree grows with $\Theta mn \log(n)$, where m is the number of states in the DFA. This means that the memory usage will be big for large files and complex languages.

7.2 Future Work

To solve the problem with the space requirements for this implementation an implementation using sequence of characters could be used instead. That is, instead of using a character as the base case, a sequence of characters is used which is sequentially lexed, an example of a sequence is one line of code. This would shrink the tree from $\log(n)$ to $\log(n/x)$ where x is the mean length of a line. Since lines in the code are roughly of the same length there will be no impact on the worst case scenario time, for instance lexing 10 characters always takes the same amount of time. Since the lines in general are short updating a line will not take long time.

Another solution which could be used is to limit how big a tree can be. When that text is bigger then what fits in a tree, the tree is split into two trees. This will result in smaller trees at the expense of run time since the combination of the trees needs to be calculated on the fly.

In general a lot of in states will have the same sequence of tokens. The implementation suggested by this report will store all such sequences separately. An improvement would be if somehow the sequences of tokens that are identical could be stored in a separate table and the in state in the transition map points to the corresponding sequence for that in state. This would not improve the space complexity, but the practical space needed would shrink.

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] Jean-Philippe Bernardy and Koen Claessen. Efficient divide-and-conquer parsing of practical context-free languages. In *Proceeding of the 18th ACM SIGPLAN international conference on Functional Programming*, pages 111–122, 2013.

[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. http://www.haskell.org/alex/doc/html/index.html, May 2014.

[6] Markus Forsberg and Aarne Ranta. Bnf converter. In *Proceedings of the 2004 ACM SIGPLAN Workshop on Haskell*, Haskell '04, pages 94–95, New York, NY, USA, 2004. ACM.

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

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

[9] Eugene Kirpichov. Incremental regular expressions. http://jkff.info/articles/ire, May 2014. English version of published russian article.

[10] Bryan O'Sullivan. The criterion package. https://hackage.haskell.org/package/criterion, May 2014.

[11] Dan Piponi. Fast incremental regular expression matching with monoids.

http://blog.sigfpe.com/2009/01/fast-incremental-regular-expression.html, January 2009.

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

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

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

[15] 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.

[16] L.G. Valiant. General context-free recognition in less than cubic time. *Journal of computer and system sciences*, 10(2):308–314, 1975.

A

Modified Java Lette Light

Here is a simplified version of Java that only have variables, numbers, strings and some simple operators. The language includes *while* loops but the lexical analyzer will read “while” as an identifier, The syntactical analyzer will later determine if it is a loop. The expressions that matches rules without a name are discarded since they are not needed for the syntactical analyzer.

Character sets

capital → [A-Z]

lower → [a-z]

letter → [a-zA-Z]

digit → [0-9]

ident → *letter* | *digit* | [_’] — Identifier Characters

white → [\t\r\n\v\f] — White space characters

Rules

white+

// [.]* — Single line comment

/*) ([∧*] | (*) [∧/])* (*)+ / — Multi line comment

Identifier → *letter ident**

Integer → *digit+*

Double → *digit+* \. *digit+*

String → \” [∧ \”]* \”

Reserved → \(| \) | \{ | \} | ; | = | \+\+ | < | \+ | - | * |

B

Incremental Lexer Source Code

Below follows the main part of the lexical routines and the data structures used in this project.

```
type State = Int
type Transition = State -> Tokens -- Transition from in state to
    Tokens
data Tokens = NoTokens
            | InvalidTokens !(Seq Char)
            | Tokens { currentSeq :: !(Seq IntToken)
                     , lastToken  :: !Suffix
                     , outState   :: !State}
-- The suffix is the sequence of as long as possible accepting
    tokens.
-- It can itself contain a suffix for the last token.
                deriving Show
-- This is either a Sequence of tokens or one token if it hits an
    accepting
-- state with later characters
data Suffix   = Str !(Seq Char)
              | One !IntToken
              | Multi !Tokens
                deriving Show
type Size     = Sum Int
type LexTree  = FingerTree (Table State Tokens,Size) Char
data IntToken = Token { lexeme   :: !(Seq Char)
                      , token_id :: Accepts}
type Accepts   = [AlexAcc (Posn -> Seq Char -> Token) ()]

tabulate :: (State,State) -> (State -> b) -> Table State b
access :: Table State b -> (State -> b)
```

```
{-- Functional Table variant
newtype Table a b = Tab {getFun :: a -> b}
tabulate _ f = Tab f
access a x = (getFun a) x
--}

type Table a b = Array State b
tabulate range f = listArray range [f i | i <- [fst range..snd
   range]]
access a x = a ! x

instance Monoid (Table State Tokens) where
  mempty = tabulate stateRange (\_ -> emptyTokens)
  f `mappend` g = tabulate stateRange $ combineTokens (access f)
     (access g)

-- The base case for when one character is lexed.
instance Measured (Table State Tokens,Size) Char where
  measure c =
    let bytes = encode c
        cSeq = singleton c
        baseCase in_state | in_state == -1 = InvalidTokens cSeq
                          | otherwise = case foldl automata
                            in_state bytes of
          -1 -> InvalidTokens cSeq
          os -> case alex_accept ! os of
            []  -> Tokens empty (Str cSeq) os
            acc -> Tokens empty (One (createToken cSeq acc)) os
    in (tabulate stateRange $ baseCase, Sum 1)

createToken :: (Seq Char) -> Accepts -> IntToken
createToken lex acc = Token lex acc

createTokens :: Seq IntToken -> Suffix -> State -> Tokens
createTokens seq suf state = if null seq
                             then NoTokens
                             else Tokens seq suf state

invalidTokens :: (Seq Char) -> Tokens
invalidTokens s = InvalidTokens s

emptyTokens :: Tokens
emptyTokens = NoTokens

---------- Combination functions, the conquer step

-- Combines two transition maps
combineTokens :: Transition -> Transition -> Transition
```

```
combineTokens trans1 trans2 in_state | isInvalid toks1 = toks1
                                     | isEmpty toks1    = trans2
                                        in_state
                                     | otherwise = combineWithRHS
                                        toks1 trans2
  where toks1 = trans1 in_state

-- Tries to merge tokens first , if it can't it either appends the
    token or calls
-- itself if the suffix contains Tokens instaed of a single token.
combineWithRHS :: Tokens -> Transition -> Tokens
combineWithRHS toks1 trans2 | isEmpty toks2 = toks1
                            | isValid toks2 =
    let toks2' = mergeTokens (lastToken toks1) toks2 trans2
    in appendTokens seq1 toks2'
                            | otherwise     = case lastToken
                                toks1 of
    Multi suffToks ->
      let toks2' = combineWithRHS suffToks trans2 -- try to
          combine suffix with transition
      in appendTokens seq1 toks2'
    One tok -> appendTokens (seq1 |> tok) (trans2 startState)
    Str s -> invalidTokens s
  where toks2 = trans2 $ outState toks1
        seq1 = currentSeq toks1

-- Creates one token from the last token of the first sequence
    and and the first
-- token of the second sequence and inserts it between the init
    of the first
-- sequence and the tail of the second sequence
mergeTokens :: Suffix -> Tokens -> Transition -> Tokens
mergeTokens suff1 toks2 trans2 = case viewl (currentSeq toks2) of
  token2 :< seq2' -> let newToken = mergeToken suff1 token2
                     in toks2 {currentSeq = newToken <| seq2'}
  EmptyL -> case alex_accept ! out_state of
    [] -> toks2 {lastToken = mergeSuff suff1 (lastToken toks2)
        trans2}
    acc -> let lex = suffToStr suff1 <> suffToStr (lastToken
        toks2)
           in toks2 {lastToken = One $ createToken lex acc}
  where out_state = outState toks2

-- Creates on token from a suffix and a token
mergeToken :: Suffix -> IntToken -> IntToken
mergeToken suff1 token2 = token2 {lexeme = suffToStr suff1 <>
    lexeme token2}

-- Creates the apropiet new suffix from two suffixes
```

```
mergeSuff :: Suffix -> Suffix -> Transition -> Suffix
mergeSuff (Multi toks1) suff2 trans2 = Multi $
  let newToks = combineWithRHS toks1 trans2
  in if isValid $ newToks
     then newToks
     else toks1 {lastToken = mergeSuff (lastToken toks1) suff2
         trans2}
mergeSuff (Str s1) suff2 _ = Str $ s1 <> suffToStr suff2
mergeSuff (One token1) (Str s) trans2 =
  let toks2 = trans2 startState
  in if isValid toks2
     then Multi $ toks2 {currentSeq = token1 <| currentSeq toks2}
     else Multi $ createTokens (singleton token1) (Str s) (-1)
mergeSuff suff1 (One token2) _ = One $ mergeToken suff1 token2
mergeSuff suff1 (Multi toks2) trans2 = Multi $ mergeTokens suff1
   toks2 trans2

-- Prepends a sequence of tokens on the sequence in Tokens
appendTokens :: Seq IntToken -> Tokens -> Tokens
appendTokens seq1 toks2 | isValid toks2 =
  toks2 {currentSeq = seq1 <> currentSeq toks2}
                        | otherwise = toks2

------------ Constructors

makeTree :: String -> LexTree
makeTree  = fromList

measureToTokens :: (Table State Tokens, Size) -> Seq Token
measureToTokens m = case access (fst $ m) startState of
  InvalidTokens s -> error $ "Unacceptable␣token:␣" ++ toList s
  NoTokens -> empty
  Tokens seq suff out_state ->
    snd $ foldlWithIndex showToken (Pn 0 1 1,empty) $ intToks seq
       suff
  where showToken (pos,toks) _ (Token lex accs) =
          let pos' = foldl alexMove pos lex
          in case accs of
             [] -> (pos',toks)
             AlexAcc f:_ -> (pos',toks |> f pos lex)
             AlexAccSkip:_ -> (pos',toks)
        intToks seq (Str str) = error $ "Unacceptable␣token:␣" ++
           toList str
        intToks seq (One token) = seq |> token
        intToks seq (Multi (Tokens seq' suff' _)) = intToks (seq
           <> seq') suff'

treeToTokens :: LexTree -> Seq Token
treeToTokens = measureToTokens . measure
```

```
---------------- Util funs

isValid :: Tokens -> Bool
isValid (Tokens _ _ _) = True
isValid _ = False

isEmpty :: Tokens -> Bool
isEmpty NoTokens = True
isEmpty _        = False

isInvalid :: Tokens -> Bool
isInvalid (InvalidTokens _) = True
isInvalid _ = False

suffToStr :: Suffix -> Seq Char
suffToStr (Str s) = s
suffToStr (One token) = lexeme token
suffToStr (Multi toks) =
  concatLexemes (currentSeq toks) <> suffToStr (lastToken toks)

isAccepting :: Tokens -> Bool
isAccepting (Tokens _ suff _) = case suff of
  Str _ -> False
  One _ -> True
  Multi toks -> isAccepting toks
isAccepting NoTokens = True
isAccepting (InvalidTokens _) = False

concatLexemes :: Seq IntToken -> Seq Char
concatLexemes = foldr ((<>) . lexeme) mempty

insertAtIndex :: String -> Int -> LexTree -> LexTree
insertAtIndex str i tree =
  if i < 0
  then error "index␣must␣be␣>=␣0"
  else l <> (makeTree str) <> r
     where (l,r) = splitTreeAt i tree

splitTreeAt :: Int -> LexTree -> (LexTree,LexTree)
splitTreeAt i tree = split (\(_,s) -> getSum s>i) tree

size :: LexTree -> Int
size tree = getSum . snd $ measure tree

-- Starting state
startState = 0
-- A tuple that says how many states there are
stateRange = let (start,end) = bounds alex_accept
```

```
            in (start -1,end)

-- Takes an in state and a byte and returns the corresponding out
    state using
-- the DFA generated by Alex
automata :: Int -> Word8 -> Int
automata (-1) _ = -1
automata s c = let base   = alex_base ! s
                   ord_c  = fromEnum c
                   offset = base + ord_c
                   check  =  alex_check ! offset
               in if (offset >= (0)) && (check == ord_c)
                  then alex_table ! offset
                  else alex_deflt ! s
```

C

Space Complexity Fingertrees

The equation in fig. C.1 describes the how the space complexity for the measures in a fingertree is reached.

$$\sum_{x=0}^{\log(n-1)} (n - 2 \cdot \sum_{y=1}^{x} 2^y) = n + n\log(n-1) - \sum_{x=0}^{\log(n-1)} 2 \cdot \sum_{y=1}^{x} 2^y =$$
$$n + n\log(n-1) - \sum_{x=0}^{\log(n-1)} 2(2^{x+1} - 2) = n + n\log(n-1) - \sum_{x=0}^{\log(n-1)} (2^{x+2} - 4) =$$
$$n + n\log(n-1) + 4\log(n-1) + 4 - \sum_{x=0}^{\log(n-1)} 2^{x+2} = n + (n+4)\log(n-1) - 2^{\log(n-1)+3} + 8 =$$
$$n + (n+4)\log(n-1) - 8(n-1) + 8 = (n+4)\log(n-1) - 7n + 16 \Rightarrow \Theta(n \log n)$$

Figure C.1: The number of characters being measured in a tree with n characters