CHALMERS

A Generator of Incremental Divide-and-Conquer Lexers

A Tool to Generate an Incremental Lexer from a Lexical Specification

Master of Science Thesis [in the Programme MPALG]

JONAS HUGO

KRISTOFER HANSSON

Chalmers University of Technology
Department of Computer Science and Engineering
Göteborg, Sweden, March 2014

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

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

Department of Computer Science and Engineering
Göteborg, Sweden March 2014

Abstract

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

Acknowledgements

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

Jonas Hugo & Kristofer Hansson, Göteborg March 2014

Contents

1

Introduction

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

1.1 Background

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

1.2 Scope of work

Dan Piponi has writen a blogpost on how to determend in a incremental way if a string furfills a regular expression. This is done by using Monoids, Fingertrees and tabulate functions. [10]

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

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

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

2

Lexer

A lexer, lexical analyser, is a pattern matcher. Its job is to divide a text into a sequence of tokens (such as words, punctuation and symbols). The Lexer is a front end of a syntax analyser [12]. The syntax analyser in turn takes the tokens generated by the lexer and returns a set of expressions and statements. This can be done by using regular expressions, regular sets and finite automata, which are central concepts in formal language theory [1]. The rest of this chapter describes the concepts of the lexer in detail.

2.1 Lexing vs Parsing

Lexers usually work as a pass before parser; giving their result to the syntax analyser. There are several reasons why a compiler should be separated in to a lexical analyser and a parser (syntax analyser).

First, simplicity of design is the most important reason. When dividing the task into these two sub tasks, it allows the programmer to simplify each of these sub-tasks. For example, a parser that has to deal with white-spaces and comments would be more complex than one that can assume these have already been removed by a lexer. When the two tasks have been seperated into sub-tasks it can lead to cleaner overall design when designing a new language. The only thing the syntax analyser will see is the output from the lexer, tokens and lexemes. The lexer usually skips comments and white-spaces, since these are not relevant for the syntax analyser.

Second, overall efficiency of the compiler can be improved. When separating the lexical analyser it allows for use of specialised techniques that can be used only in the lexical task.

Third and last, compiler portability can be enhanced. That is Input-device-specific peculiarities can be restricted to the lexical analysis [2]. Therefore the lexer can detect syntactical errors in tokens, such as ill-formed floating-points literals, and report these errors to the user [12]. Finding these errors allows the compiler to break the compilation before running the syntax analyser, thereby saving computing time.

2.2 Token Specification

The job of the lexical analyser is to translate 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

Example 2.2.1 (Valid C Idents [2])**.** Using regular expressions to express a set of valid C identifiers is easy. given an element $letter \in \{\text{a}\ldots\text{z}\} \cup \{\text{A}\ldots\text{Z}\} \cup \{\_\}$ and another element $digit \in \{0\ldots9\}$ Then using a regular expression, the definition of all valid C identifiers could look like this: $letter(letter|digit)*$.
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 r matches x_i for $1 \leq i \leq n$, in particular $(r)*$ matches the empty string, ε.
6. If r is a regular expression, then (r) is a regular expression that matches the same string as r.

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

2.2.2 Languages

An alphabet is a finite set of symbols, for example Unicode, which includes approximately 100,000 characters. A language is any countable set of strings of some fixed alphabet [2]. The term "formal language" refers to languages which can be described by a body of systematic rules. There is a subset of languages to formal languages called regular language, these regular languages refers to those languages that can be defined by regular expressions [11].

2.2.3 Regular Definitions

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

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

where:

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

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

2.3 Tokens, Patterns and Lexemes

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 describe the structures which the lexical analyser use for representing the abstract data; what these structures are for and what is forwarded to the syntactical analyser.

A lexical analyser uses three different concepts. The concepts are described below.

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

Pattern is a description of what form a lexeme may take [2]. For example, a keyword is the sequence of characters that forms the keyword, an int is a sequence consisting of integers from 0 to 9. This can be described by a regular expression.

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

As mentioned before a token consists of a token name and an optional attribute value. This attribute is used when one pattern can match more then one lexeme. For example the pattern for a digit token matches both 0 and 1, but it is important for the code generator to know which lexeme was found. Therefore the lexer often returns not just the token but also an attribute value that describes the lexeme found in the source program corresponding to this token [2].

$$\langle \textit{letter} \rangle \in \{\text{`}\texttt{a}\text{'} \text{ - } \text{`}\texttt{z}\text{'}\} \cup \{\text{`}\texttt{A}\text{'} \text{ - } \text{`}\texttt{Z}\text{'}\} \cup \{\text{`}\texttt{\_}\text{'}\}$$

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

$$\langle \textit{identifier} \rangle ::= \langle \textit{letter} \rangle \; (\langle \textit{letter} \rangle \mid \langle \textit{digit} \rangle)^*$$

$$\langle \textit{integer} \rangle ::= \langle \textit{digit} \rangle +$$

$$\langle \textit{multi-line comment} \rangle ::= \text{`}\texttt{/*}\text{'} \; ([\wedge \text{`}\texttt{*}\text{'}] \mid \text{`}\texttt{*}\text{'} \; [\wedge \text{`}\texttt{/}\text{'}])^* \; \text{`}\texttt{*/}\text{'}$$

$$\langle \textit{reserved-words} \rangle ::= \text{`}\texttt{(}\text{'} \mid \text{`}\texttt{)}\text{'} \mid \text{`}\texttt{\{}\text{'} \mid \text{`}\texttt{\}}\text{'} \mid \text{`}\texttt{;}\text{'} \mid \text{`}\texttt{=}\text{'} \mid \text{`}\texttt{++}\text{'} \mid \text{`}\texttt{<}\text{'} \mid \text{`}\texttt{+}\text{'} \mid \text{`}\texttt{-}\text{'} \mid \text{`}\texttt{*}\text{'}$$

Figure 2.1: Grammar rules for example 2.3.1 & example 2.4.1

A lexer collects chars into logical groups and assigns internal codes to these groups according to their structure, where the groups of chars are lexemes and the internal codes are tokens [12]. In some cases it is not relevent to return a token for a pattern, in these cases the token and lexeme is discarded and the lexer continues, typical examples of this is whitespaces and comments which have no impact on the code [2].

An example follows how a small piece of code would be divided given the regular language described in appendix A.
Example 2.3.1 (Logical grouping [12])**.**
Consider the following text; to be lexed:

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

Given the regular languaged defined in appendix A, the lexical analyser would use the rules defined in fig. 2.1 In order to produce the following tokens.

Token	Lexeme
Identifier	result
Reserved	=
Identifier	oldsum
Reserved	−
Identifier	value
Reserved	/
Integer	100
Reserved	;

2.4 Recognition of Tokens

The topic in the previous section covers how to represent a pattern using regular expressions and how these expressions relates to tokens. A pattern is used to determine if a string matches a token. This section is highlighting how to transform a sequence of characters into a sequence of abstract tokens using patterns.

2.4.1 Transition Diagrams

A state transition diagram, or transition diagram is a directed graph, where the nodes are labelled with state names. Each node represents a state which could occur during the process of scanning the input, looking for a lexeme that matches one of several patterns [2]. The edges are labelled with the input characters that causes transitions among the states. An edge may also contain actions that the lexer must perform when the transition is a token [12].

There are different type of states in the the diagram where one state state is said to be initial. The transition diagram always begins at this state, before any input symbols have been read. Some states are said to be accepting (final). They indicate that a lexeme has been found. If the found token is the longest match (see section 2.4.2) then the token will be returned with any additional optional values, mentioned in previous section, and the transition is reset to the initial 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 rule is not always enough since the lexer has to explore all legal edges, even if the current state is accepting. If the lexer is in a state that is not accepting and don't have any legal edge out of that state, the lexer can't 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 wasn't 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.

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

Allthough this text is not legal code, there is no lexical errors in it. Since the text starts with a multi line comment sign the lexer will try to lex it 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. 2.1 the rest of the rules can be found in appendix A.
The result:

Token	Lexeme
Reserved	/
Reserved	$*$
Identifier	result
Reserved	=
Identifier	oldsum
Reserved	$-$
Identifier	value
Reserved	/
Integer	100
Reserved	;

2.4.3 Finite Automata

Transition diagrams of the form used in lexers are representations of a class of mathematical machines called finite automata. Finite automata can be designed to recognise members of a class of languages called regular languages, mentioned above [12]. A finite automaton is essentially a graph, like transitions diagrams, with some differences:

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

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

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

Both these forms of finite automata are capable of recognising the same subset of languages, all regular languages [2].

Non-deterministic Finite Automata

An NFA accepts the input; x if and only if there is a path in the transition diagram from the start state to one of the accepting states, such that the symbols along the way spells out x [2]. The formal definition of a non-deterministic finite automaton follows:
Definition 2.4.2 (Non-deterministic Finite Automata [13])**.** 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. $\delta : Q \times \Sigma \rightarrow P(Q)$ is a transition function,
4. $q_0 \in Q$ is the start state, and
5. $F \subseteq Q$ is the accept state.

The transition function doesn't map to one particular state from a state and element tuple. This is because one state may have more then one edge per element. An example of this can be seen in example 2.4.3.

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

$$(a|b) * abb$$

The transition diagram in fig. 2.2 is representing this regular expression.

It could also be converted into the transition table shown in fig. 2.3

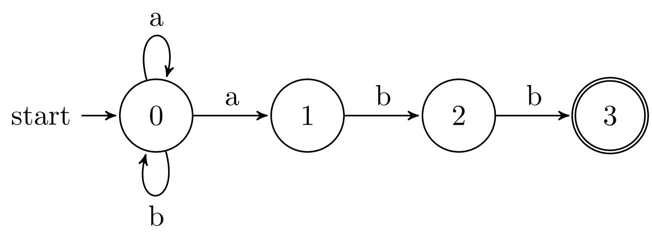


Figure 2.2: Transition Diagram, accepting the pattern $(a|b) * abb$

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

Figure 2.3: Transition Table, accepting the pattern $(a|b) * abb$

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

Deterministic Finite Automata

DFA is a special case of an NFA where,

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

A NFA is one abstract representation of an algorithm to recognise a string in one language, the DFA is a simple concrete algorithm for recognising strings. Every regular expression can be converted in to a NFA. Also every NFA can be converted in to a DFA and then converted back to a regular expression [2]. It is the DFA that is implemented and used when building lexical analysers. The formal definition of a deterministic finite automaton follows:

Definition 2.4.4 (Deterministic Finite Automata [13])**.** 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. $\delta : Q \times \Sigma \rightarrow Q$ is a transition function,
4. $q_0 \in Q$ is the start state, and
5. $F \subseteq Q$ is the set of accept states.

Example 2.4.5 (DFA representation of RegExp [2])**.** A DFA representation of same regular expression from example 2.4.3 is shown in fig. 2.4

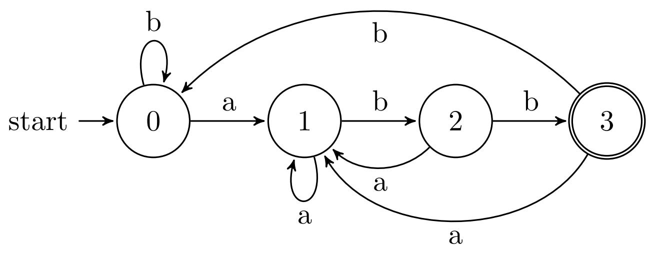


Figure 2.4: DFA, accepting the regular expression: $(a|b) * abb$

3

Divide-and-Conquer Lexer

An incremental divide and conquer lexer works by dividing the sequence, to be lexicaly analysed, into small parts and analyse them; and then combining 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 token 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 indepently and then combine the results. A Divide and Conquer algorithm always consists of a pattern with these three steps [7].

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: Recursively solve the subproblems associated with the subset.

Conquer: Given the solutions to the subproblems, combine the results to solve the original problem.

3.1.2 Associative Function

An associative function, or operator, is a function that doesn't care in what order it is applied. An example of such a function is +, which is associative since it has the propert in example 3.1.1.

In divide and conquer algorithms 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't 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 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])**.**
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 sub-problems at each recursive step, n/β is the size of each sub-problem, and $f(n)$ is the running time of dividing up the problem space into α parts, and combining the sub-results together.)
If we let $e = log_\beta \alpha$, then

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

■

3.1.4 Hands on Example

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

divide: The algorthim 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 equaly 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 recursivly calling itself twice with S_1 and S_2 as arguments respectivly.

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

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

Algorithm 1: MergeSort

Data: Sequence of integers S containing n integers
Result: Sorted sequence S
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 mergesort 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 master theorem applies, since

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

So the solution will be:

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

3.1.5 Incremental Computing

To be incremental means that, whenever some part of the data to the algorithm changes the algorithm tries to save time by only recomputing the changed data and the parts that depend on this changed data [14].

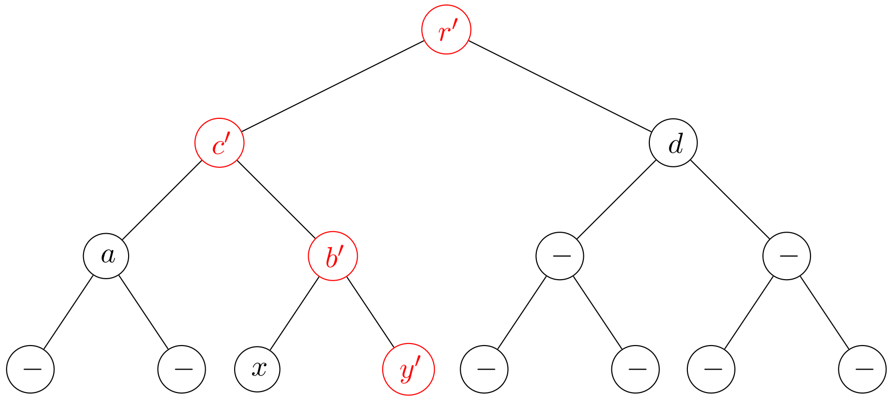


Figure 3.1: Incremental Computing, the updated nodes when a leaf changes

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 recurrsivly until the root of the tree is reached. The expected result of this would be that when a character is added to the code of 1024 tokens, instead of recalculate 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.

3.2 Fingertree

Fingertree is a tree structure which is incremantal in its nature and has good preformance. To ensure that an incremental divide and conquer algorithm can access the intermediate states a data structure like fingertrees can be used. Before describing how the fingertree is defined, an introduction to the fingertrees building blocks is given.

3.2.1 Fundamental Concepts

Fingertrees uses monoids which in abstract algebra is a set, S, and a binary operation $\bullet$ which fulfills the following three properties:

Closure $\forall a,b \in S : a \bullet b \in S$

Associativity $\forall a,b,c \in S : (a \bullet b) \bullet c = a \bullet (b \bullet c)$

Identity element $\exists e \in S : \forall a \in S : e \bullet a = a \bullet e = a$

Fingertrees uses Right and Left Reductions. This is a function which collapses a structure of $f\ a$ into a single value of type a. The base case for when the tree is empty is replaced with a constant value, such as $\emptyset$. Intermediate results are combined using a binary operation, like the monoids $\bullet$. Reduction with a monoid always return the same value, independent of the argument nesting. But for a reduction with an arbitrary constant and binary operation there must be a specified nesting rule. If combining operation are only nested to the right, or to the left, the obtained result will be a skewed reductions, which can be singled out as a type class described in fig. 3.2 [8].

```
class Reduce f where
reducer :: (a -> b -> b) -> (f a -> b -> b)
reducel :: (b -> a -> b) -> (b -> f a -> b)
```

Figure 3.2: Reduction function in Haskell

3.2.2 Simple Sequence

The defenition of the fingertrees can be described by comparing it to an already known datastructure and how that datastructer represent data. Lets take a look at the definition on a 2-3 fingertree and how they can implement a sequence. Lets start by looking at an ordinary 2-3 tree.

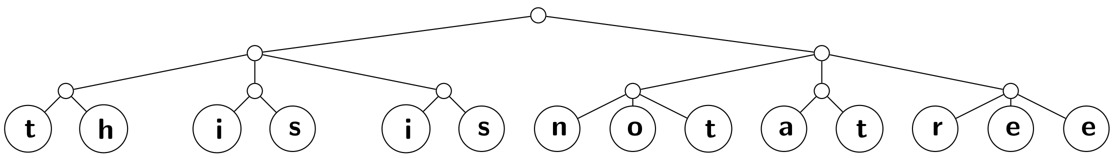


Figure 3.3: Ordinary 2-3 tree

The tree shown in the fig. 3.3 stores all it's data in the leafs. This can be expressed by defining an non-regular or nested type, as shown in fig. 3.4.

```
data Tree a = Zero a | Succ (Tree (Node a))
data Node a = Node2 a a | Node3 a a a
```

Figure 3.4: Definition of a 2-3 Fingertree

Operations on these types of trees usually takes logarithmic time in the size of the tree. But for sequence representations a constant time complexity is preferable for adding or removing element from the start or end of the sequence.

A finger is a structure which provides efficient access to nodes near the distinguished location. To obtain efficient access to the start and end of the sequence represented by the tree, there should be fingers placed at the left and right end of the tree. In the example tree, taking hold of the end and start nodes of and lifting them up together. The result should look like in fig. 3.5

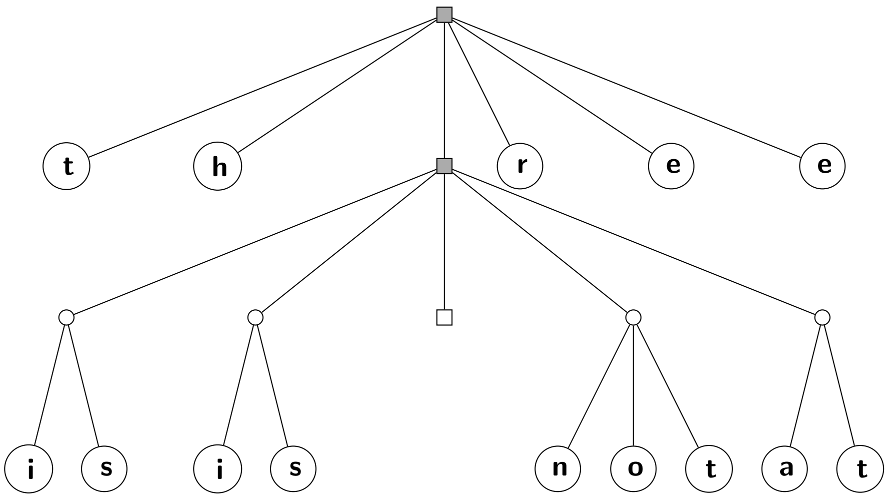


Figure 3.5: 2-3 Fingertree

Since all leafs in the 2-3 tree is at the same level, the left and right spine has the same length. Therefor the left and right spines can be pair up to create a single central spine. Branching out from the spine is 2-3 trees. At the top level there are two to three elements on each side, while the other levels have one or two sub-trees, whose depth increases down the spine. Depending on if the root node had 2 or 3 branches in the original 2-3 tree, The bottom node will have either a single 2-3 tree or an empty tree. This structure can be described as shown in fig. 3.6.

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

type Digit a = [a]
```

Figure 3.6: Definition of the Fingertree data type

Where Digit is a buffer of elements stored left to right, here represented as a list for simplicity

The non-regular definition of the $FingerTree$ type determines the unusual shape of these trees, which is the key to there performance. The top level of the tree contains elements of type a. Next level contains elements of type $Node\ a$. At the nth level, elements are of type $Node^n\ a$. which are 2-3 trees with a depth of n. This gives that a sequence of n elements is represented by a $FingerTree$ of depth $\Theta(\log n)$. An element at position d from the nearest end is stored at a depth of $\Theta(\log d)$ in the $FingerTree$

In fingertrees and nodes the reduce function mentioned in fundamental concepts is genericly defined to the following types. Reduction for the node which is shown in fig. 3.7.

```
instance Reduce Node where
    reducer (-<) (Node2 a b) z   = a (-<) (b (-<) z)
    reducer (-<) (Node3 a b c) z = a (-<) (b (-<)(c (-<) z))

    reducel (>-) z (Node2 a b)   = (z (>-) b) (>-) a
    reducel (>-) z (Node3 c b a) = ((z (>-) c) (>-) b) (>-) a
```

Figure 3.7: Reduction of a fingertrees node

Reduction of fingertrees single and double lifting of the binary operation as shown in fig. 3.8 [8].

```
instance Reduce FingerTree where
  reducer (-<) Empty z         = z
  reducer (-<) (Single x) z    = x (-<) z
  reducer (-<) (Deep pr m sf) z = pr (-<') ( m (-<'') ( sf (-<') z
      ))
    where (-<')  = reducer (-<)
          (-<'') = reducer (reducer (-<))

  reducel (>-) z Empty         = z
  reducel (>-) z (Single x)    = z (>-) x
  reducel (>-) z (Deep pr m sf) = ((z (>-') pr ) (>-'') m ) (>-')
     sf
    where (>-')  = reducel (>-)
          (>-'') = reducel (reducel (>-))
```

Figure 3.8: Reduction of a Fingertree

3.2.3 Double-ended Queue Operations

After showing how the Fingertrees basic structure is defined. Lets take a look on how fingertrees makes efficient Double-ended Queue, a queue which can be accessed from both ends, where both the operations having the time complexity $\Theta(1)$.

Adding an element to the beginning of the sequence is strait forward, except when the initial buffer (*Digit*) already is full. In this case, push all but one of the elements in the buffer as a node, leaving behind two elements in the buffer, shown in fig. 3.9.

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

Figure 3.9: Adding an element to the begining of the sequence

Adding to the end of the sequence is a mirror image of the code in fig. 3.9 and is shown in fig. 3.10.

```
(|>)                             :: 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 3.10: Adding an element to the end of the sequence

An insertion operation in the basic 2-3 tree, where the data is stored in the leafs, is done with a time complexity of $\Theta \log n$. In the fingertree the expected time complexity can be argued in the following way. Digits of two or three elements (which is isomorphic to elements of type *Node a*) is classified as safe and those of one or four elements is classified as dangerous. A double-ended queue operation can only propagate to the next level from a dangerous element. By doing so making that dangerous element safe, which means that the next operation reaching that digit will not propagate. This will result in that at most half of the operations descend one level, at most 1 quarter two levels, and so on. This will give that in a sequence of operations the average cost is constant.

The same bound hold in a persistent setting if subtrees are suspended using lazy evaluation. Laziness makes sure that changes deep in the spine do not take place until a subsequent operation need to go that far. By the above properties of safe and dangerous digits, by that time enough cheap shallow operations will have been performed to pay for the more expensive operation [8].

3.2.4 Concatenation Operations

Concatenation is a simple operation for most cases, except for the case when two *Deep* trees are being concatenated. Since *Empty* is the identity element, concatenation with an *Empty* yields the other tree. Concatination with a *Single* will reduce to $<|$ or $|>$. For the hard part when there are two *Deep* trees, the prefix of the first tree will be the final prefix. Suffix of the second tree will be the suffix of the final tree. The recursive function *app*3 shown in fig. 3.11 combines two trees and a list of $Nodes$ (basically the old prefix and suffixes down the spines of the old trees).

```
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
```

Figure 3.11: Help function for concatenating two fingertrees

Where $(<|')$ and $(|>')$ are the functions defined in fig. 3.12 and *nodes* groups a list of elements into $Node$s as shown in fig. 3.13.

```
(<|') :: (Reduce f) => f a -> FingerTree a -> FingerTree a
(<|') = reducer (<|)

(|>') :: (Reduce f) => FingerTree a  -> f a -> FingerTree a
(|>') = reducel (|>)
```

Figure 3.12: Help function for transforming a list of element in to a list of Nodes

```
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 3.13: Help function for transforming a list of element in to a list of Nodes

The concatenation of the Fingertrees calls on *app*3 with an empty list between the two trees, as shown in fig. 3.14.

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

Figure 3.14: Concatination function for Fingertree

The time spent on concatenation can be reasoned in this way. Each invocation of *app*3 arising from $(><)$ the argument list has a length of at most 4, which means that each of these invocations takes $\Theta(1)$ time. The recursion terminates when the bottom of the shallower tree has been reached, with up to 4 insertions. So the total time complexity is $\Theta(\log min\{n_1, n_2\})$ where n_1 and n_2 are the number of elements in the two trees [8].

3.2.5 Measurements

Fingertrees has been shown to work well as catenble double-ended queues. A measurement is a property describing the state of the tree. This section presents a modification of the fingertree, which provides positional and set-theoretic operations. For example taking or dropping the first n elements. These operations involve searching for an element with a certain property. To implement additional operations with good performance there must be a way to steer this search. A way to measure the tree.

A measuremenet can be viewed as a cached reduction with some monoid. Reductions, possibly cached, are captured by the class declaration in fig. 3.15. Where a is the type of a tree and v the type of an associated measurement. v must be of a monoidal structure so measurements of subtrees easily can be combined independent of the nesting. Take the size of a tree as an example. Measure maps onto the monoid over the set of natural numbers and with the binary operator of addition [8].

```
class (Monoid v) => Measured a v where
  || · || :: a -> v
```

Figure 3.15: Definition of the Measure class

Caching measurements

It should be cheap to obtain a measurement. The fingertree should ensure that an measurement can be obtian with a bounded number of $\bullet$ operations. Therefore fingertrees cache the measurements in the 2-3 nodes. In fig. 3.16 the Measure of Node is shown. The constructors and the instance declaration are completely generic: they work for arbitrary annotations.

Digits are measured on the fly. As the length of the buffer Digit is bounded by a constant, the number of $\bullet$ operations is also bounded.

Fingertrees are modified in a similar manner to 2-3 nodes. The top level of a measured fingertree contains elements of type a, the second level of type $Node\ v\ a$, the third of type $Node\ v\ (Node\ v\ a)$, and so on. The Measure function is shown in fig. 3.18. The tree type a changes from level to level, whereas the measure type v remains the same. This means that Fingertree is nested in a, but regular in v

[8].

```
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 (‖a‖ • ‖b‖) a b

node3 :: (Measured a v) => a -> a -> a -> Node v a
node3 a b c = Node3 (‖a‖ • ‖b‖ • ‖c‖) a b c

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

Figure 3.16: Measure of the data type Node

```
instance (Measured a v) => Measured (Digit a) v where
measure xs = reducel (\i a -> i • ‖a‖) • xs
```

Figure 3.17: Measure of the data type Digit

```
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 (‖pr‖ • ‖m‖ • ‖sf‖) pr m sf

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

Figure 3.18: Fingertrees Measure function

3.2.6 Sequences

A sequence in Haskell is a special case of the fingertree that has no measure. The performance is therefor superior to that of standard lists. Where a list in Haskell has $\Theta(n)$ for finding, inserting or deleting elements, that is in a list there is only known current element and the rest of the list. Results in finding the last element of a list, the computer must look at every element until the empty list has been found as the rest of the list. Where in a sequence the last element can be obtained in $\Theta(1)$ time. Adding an element anywhere in the sequence can be done in worst case, $\Theta(logn)$ [8].

3.3 Divide and Conquer Lexing in General

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

3.3.1 Treestructure

The incremental divide and conquer lexer should use a structuer where the code-lexemes can be related to its tokens, current result can be saved and easy recalculated. A divide and conquer lexer should therefore use a tree structure to save the lexed result in. Since every problem can be divided in to several subproblems, until the basecase is reached. This is cleraly a tree structure of solutions, where 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. We will henceforth refer to this as a *transition*.

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

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

```
type Transition_map = [Transition]
```

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 '/' part of a transition map might look like the following.

In the examples below 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.

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

$NoState$ transition is used to tell the lexer that using that particular transition will result in a lexical error. For reasons being covered in **??** 5, they can't be discarded.

Conquer Step The conquer step of the algorithm is to combine two transition maps in to one transition map. This is done by, for every transtion 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 byt the following logical statement where T_1 and T_2 refers to the first and second transition map.

$$\forall . t_1 \in T_1 \; \exists . t_2 \in T_2 \; o_1 = i_2, o_1 = outState(t_1), i_2 = inState(t_2) \vdash t_{new} = merge(t_1, t_2)$$

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:

Concat: 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.

```haskell
concat :: Tokens -> Tokens -> Tokens
concat tokens1 tokens2 = tokens1 >< tokens2
```

Combine: 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 combined with the first token in the second transition in to one token and put between the two sequences.

```haskell
combine :: Tokens -> Tokens -> Tokens
combine tokens1 tokens2 = prefix1 |> newToken >< suffix2
  where prefix1 |> token1'   = tokens1
        tokens2' <| suffix2  = tokens2
        newToken             = token1' `combinedWith` tokens2'
```

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.

$$\begin{bmatrix} 0 & Single'/' & 1 \\ 1 & Single'/' & Accepting5 \end{bmatrix} `combineTokens` \begin{bmatrix} 0 & Single'/' & 1 \\ 1 & Single'/' & Accepting5 \end{bmatrix} = \begin{bmatrix} 0 & Single'//' & Accepting5 \\ 1 & Multiple'/'[]'/' & Accepting1 \end{bmatrix}$$

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

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

$$\begin{bmatrix} 0 & Single'//' & Accepting5 \\ 1 & Multiple'/'[]'/' & 1 \end{bmatrix} `combineTokens` \begin{bmatrix} 0 & Single'\backslash n' & Accepting6 \\ 1 & Single'\backslash n' & 1 \\ 5 & Single'\backslash n' & NoState \end{bmatrix} = \begin{bmatrix} 0 & Multiple'//'[]'\backslash n' & Accepting6 \\ 1 & Multiple'/'[]'/\backslash n' & 1 \end{bmatrix}$$

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

3.3.3 Expected Time Complexity

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

The Bankers Method

The bankers method is a technique used to calculate the practical time assumption where it accounts for accumulated debt. Each debit represents a constant amount of suspended work. When a computation initially suspends, it create a number of debits proportional to it's shared cost and associate each debit with a location in the object. The choice of location for each debit depends on the nature of the computation. If the computation is monolithic (i.e., once begun, it runs to completion), then all debits are usually assigned to the root of the result, which the incremental lexer is not. But if the computation is like the lexer a incremental, then the debits may be distributed among the roots of the partial results.

The amortized cost of an operation is the unshared cost of the operation plus the number of debits discharged by the operation. Note that the number of debits created by an operation is not included in its amortized cost. The order in which debits should be discharged depends on how the object will be accessed; debits on nodes likely to be accessed soon should be discharged first.

Incremental functions play an important role in the bankers method because they allow debits to be dispersed to different locations in a data structure, each corresponding to a nested suspension. Then, each location can be accessed as soon as its debits are discharged, without waiting for the debits at other locations to be discharged. This means that the initial partial results of an incremental computation can be paid for very quickly, and that subsequent partial results may be paid for as they are needed [9].

Banker Method on the Fingertree

The argument for the amortized time can be expressed using the Banker method. This is done by assigning the suspension of the middle tree in each Deep node as many debits as the node has safe digits. (0,1 or 2) A double-ended queue operation which descends k levels turns k dangerous digits into safe digits. By doing so creates k debits to pay for the work done. Applying the bankers method of debit passing to any debits already attached to these k nodes. It can be shown that each operation must discharge at most one debit. Therefore the double-ended queue operations run in $\Theta(1)$ amortized time [8].

3.4 Lexical Errors

Since the lexer has to be able to handle any kind of possible not ”complete” tokens, error handling has to take this into account. for instance if an illegal character would be found somewhere in the code the lexer can’t return a lexical error right away since said character could be part of a comment. One way to handle this is to tie the lexical errors to transitions. in doing so the transition map can have one transition that is valid and another that produces a lexical error. to include this the transition data type can be modified as follows:

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

If the transition is ($State_i$,$Nothing$) there is a lexical error for the string lexed if the incoming state is $State_i$.

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 of 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 then 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. Therefor there are advantages of using sequences instead of lists. 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 be on one of three different forms. The part of the text lexed can end in:

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

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.

This data type is used in the lexical routines. The reason for using transition maps is that the lexer doesn't 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 needs to be stored since it ensures that the tokens are computed.

```
type State = Int

type Transition = State -> Tokens

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

Figure 4.3: Transition Data Type

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 doesn't guarantee that the actual tokens are computed which may result in slow performance at a later stage in the lexing.

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. 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 leafs and with the table format transition map as it's measure. The *Table* data type has to be a monoid in order to be a legal measure of the fingertree.

In order for the table data type to be a legal measure of the fingertree the table data type has to be a monoid.

```
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 have 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.

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.

```
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

4.5 Lexical routines

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

4.5.1 combineTokens

```
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 generatd are valid, the tokens are passed on to *combineWithRHS* together with the second transition.

4.5.2 combineWithRHS

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

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.

```
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

isEmpty If *toks*2 is empty *toks*1 is returned.

isValid If *toks*2 is valid it means that the last token from the *toks*1 can be combined into one token with the first token in *toks*2.

otherwise If *toks*2 is not valid the lexer checks the suffix of *toks*1 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 *toks*1.
- 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 doesn't end in a valid state and *InvalidTokens* will be returned.

4.5.3 mergeTokens

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

- 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*1 is combined with *suff*1. When two suffixes are combined some extra checks needs to be done. 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*.

```
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

4.5.4 mergeSuff

mergeSuff checks which pair of suffix it has and takes the appropriate actions.

```
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

- 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 $mergeTokens$.

4.5.5 appendTokens

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 as mentioned before three requirements, it should be Robust, Efficient and Precise. Robustness means that the lexer doesn't crash when it encounter an error in the syntax. That is, if a string would yield an error when lexed from the starting state the lexer doesn't 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 code.

For it to be efficient the feedback to the user must be instant, or more formaly the combination of two string should be handled in $O(log(n))$ time.

Finally to be precise the lexer must give a correct result. This chapter will talk about how these requirements are tested and what the results were.

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 is 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 subtext of a text and be able to combine two subtexts. If the lexical analysis of one subtext doesn't result in any legal tokens it must be able to be combined with other subtexts that makes it legal tokens. The lexical analysis of a subtext might not always result in the same tokens that the combination of the subtext with another text would give.

To test these cases a test was constructed that does a lexical analysis on two subtexts 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 text using the incremental lexer and the result using a sequential lexer.

It is not enough to test if the combination of two subtexts yields the same sequence of tokens as the text. To test that the result of the incremental lexer is the correct sequence of tokens generated, it is compared to what a sequential lexer generates. This comparison is an equality test of the text, it checks token for token that they are the same kind of token and have 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.

5.2 Performance

To measure the perfomance of the incremental step we created the fingertree for two pieces of code. By creating the two fingertrees the transition map for the code in those trees are created aswell. The benchmarking was then done on the combination of the two trees. The results of the incremental lexer benchmarking suggests a running time of $O(log(n))$.

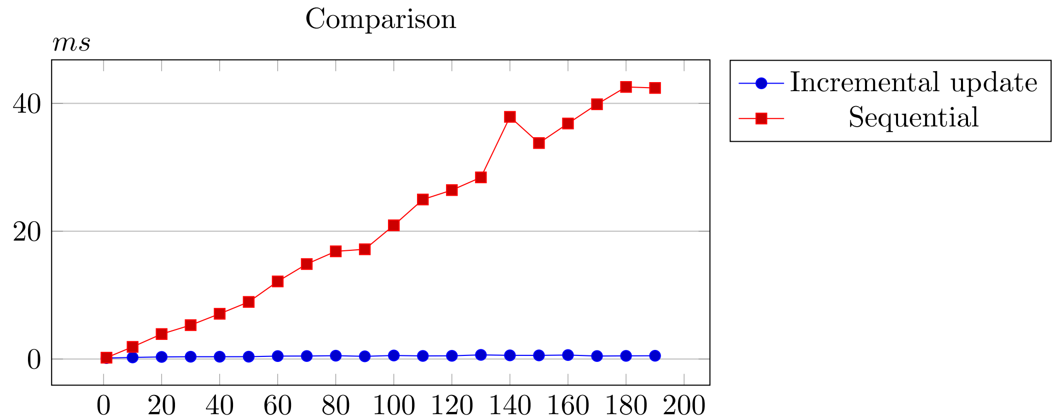
To get a reference point the same text was lexed using a sequential lexer.

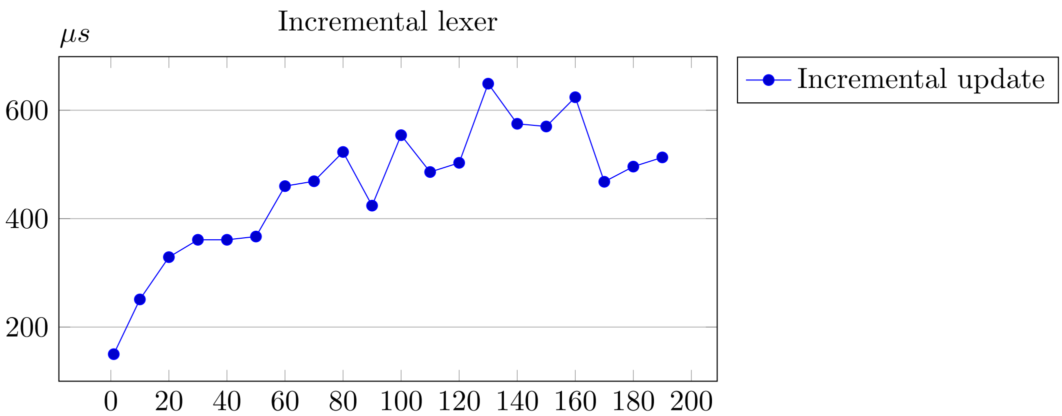
#Lägga till nåt om minnes utrymmet som krävs.

#Bättre förklaring till graferna.

#Tester för nybyggning av träd?

Example 5.2.1 (Benchmarking times of the incremental and sequential lexer).





The space complexity for the fingertree structure is O(n). According to our tests 100 characters of code translates to roughly 1MB of datastructure.

6

Discussion

#Discuss discuss!!

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

#possibility of using sequential lexing first time?

#When is it advantagous with incremental lexing

6.1 Pitfalls

This section will describe techniques that were tried under the constuction of the incremental divide and conquer lexer but was shown to give bad results.

6.1.1 Brutefore

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

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

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

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

This also has some problems though. When keywords like “else if” are introduced the lexer will start to lex like in example 6.1.1. To solve this the lexer checks when two tokens are completely uncombinable if the first of these have an accepting state as outgoing state. If the token don’t have an accepting out state, the lexer tries to break up the token until it does. The exception to this rule is single characters which are permitted to not have no accepting out states.

Example 6.1.1 (else if lexing)**.** Somewhere in the middle of the code “... 1 else 0 ...”

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

Example 6.1.2 (Devide and Append)**.** The lexer will always try to build as lage tokens as possible. When it realizes that this cant be done it has to backup and try to combine the parts in a different way. This example will show how this is done in theory.

The code segment for this example is:

”else return”.

The tree in fig. 6.1 shows the first step of the token combine rutine. Clearly this returns a nonexsisting token. From here when the lexer has found that there are no tokens for this lexeme it will try to split the left child token.

split(”else ”) => [”else”,” ”]

Now the lexer has a pair of two lexems that represent valid tokens. The lexer knows that combinding these two lexems in the pair returns in a NoToken result. So The only thing to do is to try to combine the right token in the pair with the right child token and let the token to the left in the pair stand alone. This also return a NoToken. So the same thing will be done again. The lexer tries to split the left child before NoToken was given. In this case the whitespace.

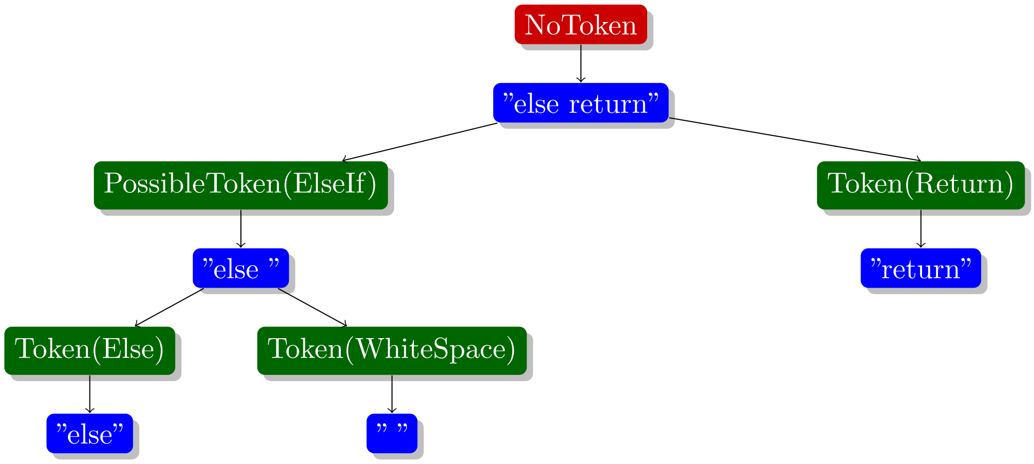


Figure 6.1: Lexer thinks "else " is an "else if" pattern.

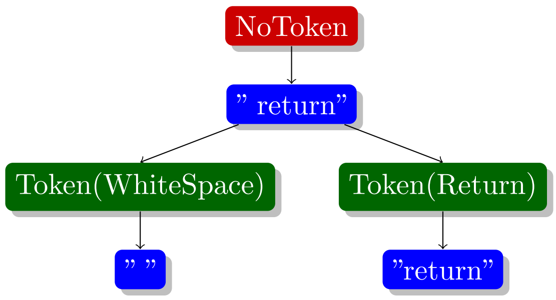


Figure 6.2: Lexer tries to combine an white space with a return statement

$$\text{split(" ")} => []$$

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

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

6.1.2 Dont know what to call this!

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

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

are considered accepting.

Example 6.1.3 (Transition map for a token)**.** A hypothetical transition map for the char 'i'.

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

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

Example 6.1.4 (Combining two tokens)**.** 'if' can be an ident (state 1) or part of 'else if' (state 5).

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

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

Example 6.1.5 (Combining a token a part of the second token)**.** 'ie' ends in the accepting state for ident (1) and '_' starts in the starting state.

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

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

Perhaps remove this part

However the remainder may not have the start state as a possible incomming state. In this case the lexer tries to find the largest possible token (that has the starting state as incomming state) and tries to construct a token of the rest of the remainder, repeating this procedure until the entire remainder has been split into acceptable tokens. All the tokens accept the one that is on the very end of the sequence will have all but their accepting states stripped. This case does occur quite frequently since most languages has comments and strings which can contain anything.

Example 6.1.6 (Handling the remainder)**.** '_' starts in the starting states and ends in an accepting state and 'e' starts in the starting state, it doesn't have to end in an accepting state.

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

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

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

7

Conclusion and Futher Work

#what will our Minions do????

#implementation of ropes

Bibliography

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

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

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

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

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

[6] 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] Chris Okasaki. *Purely Functional Data Structures*. Cambridge University Press, New York, NY, USA, 1999.

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

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

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

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

[14] 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.

A

Java Lette Light

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

<table>
<tr><td colspan="4">Character sets</td></tr>
<tr><td>capital</td><td>→</td><td>[A-Z]</td><td></td></tr>
<tr><td>lower</td><td>→</td><td>[a-z]</td><td></td></tr>
<tr><td>letter</td><td>→</td><td>[a-zA-Z]</td><td></td></tr>
<tr><td>digit</td><td>→</td><td>[0-9]</td><td></td></tr>
<tr><td>ident</td><td>→</td><td>letter | digit | [_’]</td><td>Identifier Chara</td></tr>
<tr><td>white</td><td>→</td><td>[\t\r\n\v\f]</td><td>The white spac</td></tr>
<tr><td colspan="4">Rules</td></tr>
<tr><td></td><td></td><td>whie+</td><td></td></tr>
<tr><td></td><td></td><td>// [.]*</td><td>Single line comm</td></tr>
<tr><td></td><td></td><td>/*) ([^*] | (*) [^/])* (*)+ /</td><td>Multi line comm</td></tr>
<tr><td>Identifier</td><td>→</td><td>letter ident*</td><td></td></tr>
<tr><td>Integer</td><td>→</td><td>digit+</td><td></td></tr>
<tr><td>Double</td><td>→</td><td>digit+ \. digit+</td><td></td></tr>
<tr><td>Reserved</td><td>→</td><td>\(| \) | \{ | \} | ; | = | \+\+ | < | \+ | - | * |</td><td>Reserved chara</td></tr>
</table>

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
```