UNIVERSITY OF GOTHENBURG

Implementing incremental and parallel parsing

A subtitle that can be rather long

Master of Science Thesis in Computer Science

Tobias Olausson

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

Implementing incremental and parallel parsing
A subtitle here that can be quite long

Tobias Olausson

Examiner: Patrik Jansson

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

Cover: an image that is used as a cover image

Department of Computer Science and Engineering
Göteborg, Sweden, May 2014

Abstract

This is an abstract

Contents

Chapter 1

Background

1.1 Introduction

The topic of this thesis is to do **parsing** in an **incremental** fashion that can easily be **parallelizable**, using a **divide-and-conquer approach**. In this section, I will give a brief explanation of the topics covered, and end with a motivation for why this is interesting to do in the first place.

1.1.1 Divide-and-conquer

Divide-and-conquer algorithms are a fundamental class of algorithms to computer science. The name refers to the techinque of breaking down a problem into sub-problems, where the same rule is applied recursively (the divide step). Each sub-problem can usually be solved independently, and the results of the sub-problems are then combined, finally becoming the result of the initial problem (the conquer step) [KT06, p.209]. A typical example is mergesort, where a list of elements is broken down to lists of single elements (obviously sorted), that are then combined using an improved insertion sort, observing that each sub-list is sorted. For this project, the tree structure that will be used has been shown to be suited for divide-and-conquer by Bernardy and Claessen [BC13].

1.1.2 Incrementality

Doing something incrementally means that one does it step by step, and not longer than neccessary. The concept has been used since the 70s [WDT76], but is especially relevant for code editor purposes, where one typically only wants information about the snippet currently displayed in the editor. For large files, this can save lots of work, so that rather than parsing 1000 lines,

one may only have to parse 50 of them. The text editor use case has been described in more detail by Bernardy [Ber09] using the Yi editor as subject.

1.1.3 Parallelism

In modern day processors, rather than increasing the clock frequency, efforts are put into building processors with many cores, being able to run instructions in parallel. Programmers have for many years written code that runs several instructions seemingly simultaneous, even on single-core processors. With multi-core processors this can be done truly in parallel. Since divide-and-conquer algorithms usually work on several independent sub-problems, they are well-suited for parallelization. For this to become a reality, however, both the compiler and the source code must be written in a certain way to permit parallelization.

1.1.4 Parsing

To parse is a to check if some given input corresponds to a certain language's grammar, and in this thesis I will use **context-free grammars** for programming languages. Many programming errors are syntactical ones, such as misspelled keywords, missing parenthesis or semicolons and so on. All such errors are caught in parsing. Parsing will be described in more detail in section 1.4.

1.1.5 Motivation

In compilers, lexing and parsing are the two first phases. The output of these is an abstract syntax tree (AST) which is fed to the next phase of the compiler. But an AST could also provide useful feedback for programmers, already in their editor, if the code could be lexed and parsed fast enough. With a lexer and parser that is incremental and that can also be parallelized could real-time feedback in the form of an AST easily be provided to the programmer. Most current text editors give syntax feedback based on regular expressions, which does not yield any information about depth or the surrounding AST.

TODO: Something something about connecting to a type-checker.

1.2 Lexing

In compilers, a *lexer* reads input source code and groups the characters into sequences called *lexemes* so that each lexeme has some meaning in the language the compiler is built for [ALSU07, p. 5, p. 109]. The lexemes are wrapped in *tokens* that denote what function and position each lexeme has. The tokens are then passed on to the parser for syntactic analysis.

For a language like C, the code in figure 1.1 would be valid, and can serve as an example of how lexing is done.

```
1 while(i < 5) {
2     i++;
3 }
```

Figure 1.1: A while loop that would be valid in C

A lexer for C would recognise that `while` is a keyword and place it in its own token. It would also observe that `(`, `)`, { and } are used for control grouping of code. Furthermore, `i` is an identifier and `5` is a number, < and `++` are operators and `;` denote separation of statements. All of these will be forwarded as tokens to the parser.

1.2.1 LexGen

As a master's thesis, a generator for incremental divide-and-conquer lexers was developed in 2013 by Hansson and Hugo [HH14]. Since the aim of this thesis is to write an incremental divide-and-conquer parser, their work is well-suited as a starting point, and as something to build on. Their lexer utilised Alex for core lexing routines and relied heavily on the use of arrays and finger trees, which we will see more of later. It is important to be able to use an incremental lexer when building an incremental parser, since we would otherwise have to lex the whole character stream before even getting to the parsing stage.

1.3 Context-free grammars

Context-free grammars are a way to describe formal languages, such as programming languages. They describe both the alphabet of a language and the rules for how to combine words in that language.

Formally, a context-free grammar is a 4-tuple: $G = (V, \Sigma, P, S)$ [HMU03, p.171]. V is a set of non-terminals, or variables. Σ is the set of terminal symbols, describing the content (or alphabet) that can be written in the language. P is a set of productions (or rewrite rules) that constitutes the recursive definition of the language. S is the start symbols, where $S \in V$.

The language recognized by a context-free grammar G is denoted $L(G)$ and is defined as

$$L(G) = \{w \in \Sigma^* \mid S \overset{*}{\underset{G}{\Rightarrow}} w\}$$

That is, all words in the language that can be derived using the rules from the grammar and starting from the start symbol [HMU03, p. 177]. A language L is said to be context-free if there is a context-free grammar G that recognizes the language, meaning that $L = L(G)$.

We can exemplify this using a simple made-up language of if-then-else clauses. The language terminals are *if, then, else* and *true and false*. There are two variables, I (for if) and R (for recursive) described by a total of four productions. The starting symbol is I - so just writing *true* would not be valid for this language. The formal definition of such a language can be seen in figure 1.2. Note that while P is not explicitly defined, each of the rules for I and R constitute P. Each production (partially) defines a variable and contains terminals and/or symbols on its right-hand side [HMU03, p.171].

$$\begin{aligned}
&G = (V, \Sigma, P, I)\\
&V = \{I, R\}\\
&\Sigma = \{true, false, if, then, else\}\\
&I \rightarrow if\ R\ then\ R\ else\ R\\
&R \rightarrow I\\
&R \rightarrow true\\
&R \rightarrow false
\end{aligned}$$

Figure 1.2: Context-free grammar for a recursive if-then-else language

1.3.1 Chomsky Normal Form

Chomsky Normal Form (CNF) is a subset of context-free grammars that was first described by linguist Noam Chomsky. Productions in CNF are restricted to the following forms:

A	$\rightarrow$	BC,	A is a variable, B and C are productions
A	$\rightarrow$	a,	A is a variable, a is a terminal symbol

Figure 1.3: Rules allowed in Chomsky Normal Form

Since grammars in CNF are restricted to branches or single terminal symbols, they are well suited for usage in divide-and-conquer algorithms. There are several existing algorithms to convert context-free grammars into CNF, so one does not have to write their grammars in CNF in the first place [LL09].

1.3.2 Backus-Naur Form

Context-free grammars are often used to describe the syntax of programming languages. Such descriptions are often given in a **Backus-Naur form** [Bac59], where each rule is written on the following form:

$$Label.\ Variable ::= Production$$

This labelled Backus-Naur form is what we will be using in this thesis, and is the format also used in the **BNF Converter (BNFC)** [bnf], a lexer and parser generator tool developed at Chalmers. Given such a grammar, BNFC generates, among other things, a lexer and a parser, implemented in one of several languages, for the language described in that grammar. According to its documentation, usage of BNFC saves approx 90% of source code work in writing a compiler front-end.

1.4 Parsing

The role of the parser is, given a list of tokens, to determine if that those tokens can be written in that order for a specific language. More precise, and connecting to the language of a context-free grammar above, the parser is given a string w and checks if

$$w \in \Sigma^*, S \overset{*}{\underset{G}{\Rightarrow}} w$$

that is, to check if the given string can be generated by applying the grammar rules recursively. There are many different algorithms to do this, most common are LL(k) and LR(k) parsers that are bottom-up and top-down parsers, respectively [ALSU07, p.192]. This project however, will use a variant of the CYK algorithm, which can be said to be a bottom-up parser, but employs a more general technique.

1.4.1 CYK algorithm

The CYK algorithm is named after its inventors Cocke, Younger and Kasami, who independently discovered the algorithm in the late 1960s [You67]. The algorithm works on a context-free grammar in CNF, yields a matrix with the following properties:

> This recognition algorithm will be framed in terms of a recognition matrix. This matrix lists, for each substring of the test string S_t , all the symbols in N which generate that substring. In particular, this matrix lists the symbols which generate the full string S_t : if special symbol S is contained in this list, the string S_t is then accepted as a sentence in the language; if not, it is rejected.

Figure 1.4: Description of recognition matrix by Daniel H. Younger.

Note that N in figure 1.4 refers to the set of variables, which we denote as V. The algorithm creates a square matrix of dimension $|S_t| + 1$, whose closure C is computed as follows:

$$C_{i,i+1} = \{A | A ::= S_t[i] \in P\} \tag{1.1}$$

$$C_{ij} = \sum_{k=i+1}^{j} C_{ik} \cdot C_{kj} \tag{1.2}$$

$$x \cdot y = \{A | A_0 \in x, A_1 \in y, A ::= A_0 A_1 \in P\} \tag{1.3}$$

As we can see, just above the diagonal of C, we place all variables that can match that substring of the input as a terminal. Anything below the diagonal is zero, and anything that is not just above the diagonal is computed by checking if there are any rules $A ::= BC$ as seen in equation 1.3.

1.4.2 Valiant

In 1975, Leslie G. Valiant refined the algorithm by showing that context-free recognition can be reduced to matrix multiplication of boolean matrices [Val75]. This was done by first reducing recognition to the transitive closure of upper-triangual matrices. Valiant then showed that closure could be reduced to multiplication, and furthermore to boolean multiplication, of matrices.

1.4.3 Improvement by Bernardy & Claessen

Running time analysis. Oracle for lists.

1.5 Dependently typed programming

In this project, programming with dependent types is a core technique in the parsing process. Therefore, it is good to know what this means before diving further into it.

In a strictly typed programming language like Haskell, every value has a type that is enforced. Assigning an integer a floating-point value would not type-check and therefore would not compile. While this is useful and saves debugging time, it does not say anything about the contents of the values.

A motivating example often used is implementation of a vector type. Vectors in this case is a fixed-length list of some type. In a typical Haskell setting we may have a type as follows:

```
1 data Vec a = Nil | V a Vec
2
3 head :: Vec a → a
4 head Nil = error "empty vector"
5 head (V a _) = a
```

Figure 1.5: Vector type and head function

This looks good, but it has the inherent problem that any code that tries to access the head of an empty Vec will compile but result in a runtime error.

In dependently typed programming, types may contain other types, acting as values, so that they relate the same way a value relates to its type. While Haskell is not a dependently typed programming language, there are ways to use dependent types even in Haskell. For our vector example that would look something like this:

```
1 data Nat = Z | S Nat
2 data Vec a n where
3     Nil :: Vec a Z
4     V :: a → Vec a s → Vec a (S s)
5
6 head :: Vec a (S b) → a
7 head Nil = error "empty vector" -- this does not type-check
8 head (V a _) = a
```

Figure 1.6: Dependently typed vector with head function

We created a new type Nat (for natural numbers) to keep track of the size of our vector. The new Vec type is *dependent* on the Nat type, while still holding values of some type `a`. What this code does not permit, however, is the Nil case for head. Because the type of head requires the Vec to be non-nil (with (S b) in its type signature) there is no need to check for a Nil vector here. In fact, the compiler will not pass the above code, as indicated by the comment, since the first case in head does not type check. The main advantage of this vector type is that any code that uses head and passes type-checking will be guaranteed to never encounter an empty vector. This way, even more bugs are caught at compile-time.

Chapter 2

Implementation

The actual goal of this project was to implement a parser that would be incremental and easily parallellizable. In short, the parser hooks into a previously written lexer, uses a tree structure and some matrix multiplication code, and is as a whole generated from a grammar specification using BNFC.

Before going into details about the implementation of this parser, there are a couple of libraries and programming techniques one has to be familiar with before moving forward. I will first describe those, and then move on to describe changes to the lexer I inherited, and the implementation of the parser.

2.1 Finger trees

A finger tree is a finite data structure with logarithmic time access and concatenation. The finger tree is similar to a general binary tree, where each branch has a couple of *fingers* (values) so that adding a new value does not neccessarily add a new branch to the tree. The tree structure makes the data structure suitable for a divide-and-conquer algorithm. TODO: Referens till paper om finger trees

A Haskell implementation suitable for everyday needs exists in the `Data.Sequence` package, and a more general structure is available in the `Data.FingerTree` package. The more general one is the one that will be used for this project, and is the one that was used for the LexGen project.

2.1.1 Measuring and Monoids

Two specific features in the general FingerTree data type are monoids and measuring. These are fundamental to the parser, so we will look more deeply

into them here.

A *monoid* is a mathematical object with an identity element and an associative operation. In Haskell, this is provided by writing instances of this type class:

TODO: Src-referens

```
1 class Monoid a where
2     -- Identity of mappend
3     mempty  :: a
4     -- An associative operation
5     mappend :: a → a → a
```

Figure 2.1: The Monoid type class

This means that anything that is a Monoid has an identity element (that can be accessed with *mempty*) and an associative operation to append monoids together (*mappend*). A simple list example illustrates this:

```
1 instance Monoid [a] where
2     mempty = []
3     mappend = (++)
```

Figure 2.2: Monoid instance for lists

The FingerTree type has a notion of being able to *measure* its elements. In this case, to measure means to have a function that, given an element of the type the FingerTree contains, yields a value of some type – the measure of that element. Furthermore, any type that the elements can be measured to has to be a monoid. The existance of a measured instance is ensured by the FingerTree API.

TODO: src-referens

```
1 -- Things that can be measured
2 class Monoid v ⇒ Measured v a | a → v where
3     measure :: a → v
4
5 -- FingerTrees are parametrized on both v (measures) and a (values)
6 data FingerTree v a
7
8 -- Create an empty finger tree
9 empty :: Measured v a ⇒ FingerTree v a
```

Figure 2.3: Measuring and the FingerTree type

This means that, in order to use the FingerTree, one need to fulfil a few criteria first. Let's say you want to have a FingerTree of Strings and that the measure should be the (combined) length of the Strings, then your type would be `FingerTree Int String`. For that to work, you first need to be able to convert between Strings and Int, by writing an instance of Measured for String Int. This would typically just be the length function. However, for that to work you need a Monoid instance for Ints. Perhaps it would look something like this:

```
1 type MyTree = FingerTree Int String
2 instance Measured String Int where
3     measure = length
4 instance Monoid Int where
5     mempty = 0
6     mappend = (+)
```

Figure 2.4: One possible measure from String to Int

It should be noted however, that since instances cannot be hided, writing a general Monoid instance for Ints over addition is perhaps not the best idea. Wrapper types with instances over addition and multiplication are available in the `Data.Monoid` library.

2.2 Lexing

For lexing code into tokens, the results from the LexGen project was used. However, some modifications had to be done to LexGen in order to be easily

generated from BNFC as an Alex file. One large change was done to the lexing code, however, which is due to the fact that not only the lexer, but also the parser, should work incrementally. In the LexGen code, the output structure is a `Sequence` of tokens. Since `Sequence` is a less general implementation of finger trees, they cannot be measured, and is therefore not as suitable to use in an incremental setting. Hence, instead of outputting tokens as a `Sequence`, the code was changed to output as another `FingerTree`, from which the tokens could then be measured (see section 2.3.2 below). TODO: Code here?

2.3 Parsing

TODO: Move this part down? Duplicate in BNFC TODO: An illustration here perhaps?

Observing that the lexer could easily be generated from BNFC, it was natural to plug it in, and given that, the parser code is so general that it can also be generated from any LBNF grammar.

2.3.1 BNFC

There was an existing reference implementation in BNFC for the optimisation to Valiant's algorithm, that could be accessed using the `--cnf` option. That option generated large tables needed for combining different tokens. Since the this project is similar to the reference implementation, it was natural to generate the new parser by using a new option.

For the Haskell backend, BNFC uses Alex as a lexer, as did LexGen, so it was easy to use the LexGen core and have BNFC and Alex generate the DFA needed for the lexer to work.

2.3.2 Pipeline of measures

TODO: Clarify that ONE char does not become the whole AST. Tree structure! TODO: Paste code with Measured constraints on the Measured instance!

Using the `FingerTree` type, the lexer could use that data type to measure characters into an intermediate type for lexing. That intermediate `FingerTree` could then in turn be measured to the type used for parsing.

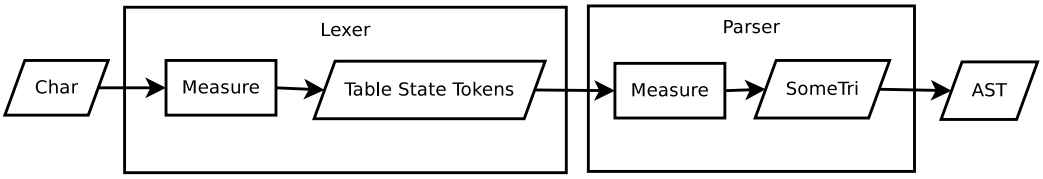


Figure 2.5: Diagram of measuring pipeline

Looking at simple testing code shows easily how the data progresses through the pipeline. `stateToTree` is an auxiliary function extracting a `FingerTree` from the internal lexer state.

```
1 test :: FilePath → IO (SomeTri [(CATEGORY,Any)])
2 test filename = do
3     file ← readFile filename
4     let mes = measure $ makeTree file
5         tri = measure $ stateToTree mes
6     return (results tri)
```

Figure 2.6: Code showing the measuring pipeline

Note that figure 2.5 is restricted to a single char. This process is done for every char in the input source code, and the results are merged using the monoid implementations of `mappend` for the lexer and the parser. This behaviour is done automatically with the call to `measure`. The lexer measure yields a lexer state containing tokens, which are then in turn measured into the matrix type `SomeTri [(CATEGORY,Any)]` by the parser, where each tuple holds a value of the `CATEGORY` type, representing an intermediate parser state, such as an almost-complete function header, and `Any` is a universal type that can hold any value, and is used as an intermediary for the generated AST. Since the lexer was not our main concern, we will not look further into its inner workings here.

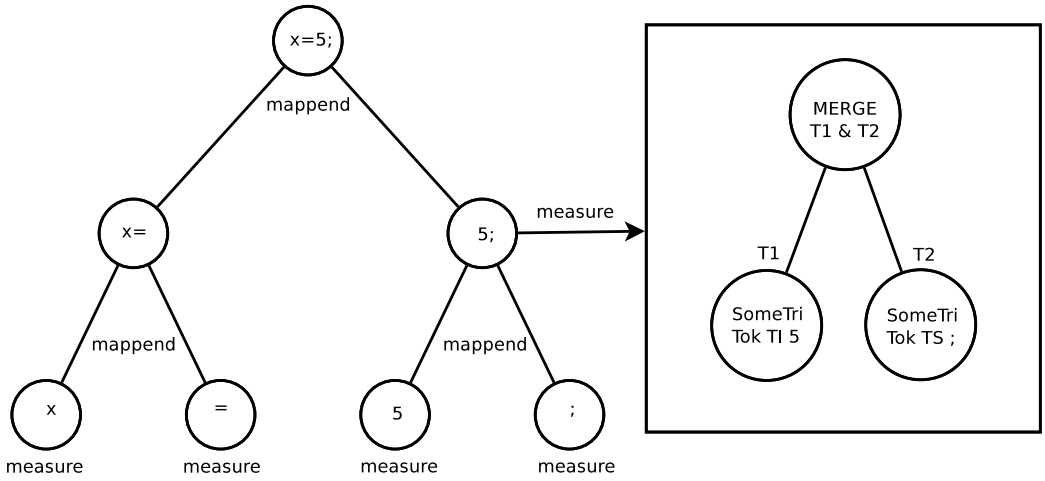


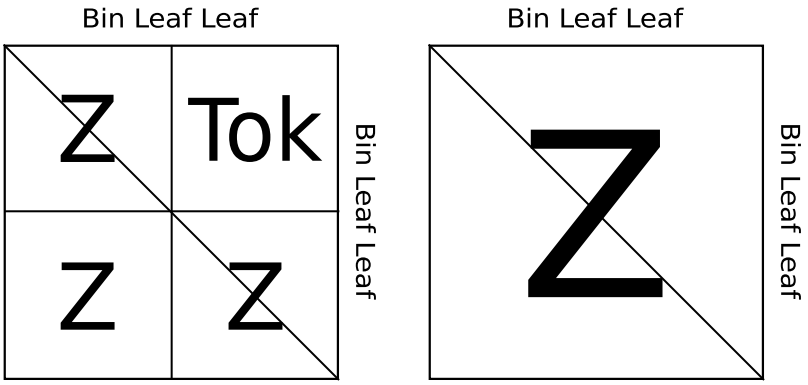
Figure 2.7: Graphical representation of lexing and parsing using trees

The `Measured` instance for the lexer was written as part of the LexGen project, and was only slightly modified to fit the parser. The `Measured` instance for the parser is far more interesting, though. We can se how it works in figure 2.8

```
1 instance Measured (SomeTri [(CATEGORY,Any)]) IntToken where
2     -- Note: place the token just above the diagonal
3     measure tok = T (bin' Leaf' Leaf') (q True :/: q False)
4       where q b = quad zero (t b) zero zero
5             select b = if b then leftOf else rightOf
6             t b = case intToToken tok of
7                 Nothing    → Zero
8                 Just token → One $ select b $ tokenToCats b token
```

Figure 2.8: Measure from token to upper-triangular matrix

We create a 2x2 matrix, and place the token in the upper-right corner – just above the diagonal. If the lexer did not return a token, a zero matrix of the same size is created. This is shown in figure 2.9. The large zero matrix is due to the `quad` function pattern matching on its arguments to support the optimization of matrix multiplication.



(a) 2x2 matrix with a token (b) 2x2 matrix without a token

Figure 2.9: A graphical representation of the Measured instance for SomeTri

Finally, the actual parsing happens in the monoid instance for SomeTri, where the call to merge in turn creates a call to closeDisjointP, which in turn uses mul. mul is a function defined in the typeclass RingP, and uses the combine tables generated by the reference implementation in BNFC.

```
1  -- In the parser, [(CATEGORY,Any)] is used as the parametrized type.
2  instance RingP a ⇒ Monoid (SomeTri a) where
3      mempty = T Leaf' (Zero :/: Zero)
4      t0 `mappend` t1 = unsafePerformIO $ do
5        b ← randomIO
6        return (merge b t0 t1)
7
8  instance RingP [(CATEGORY,Any)] where
9    mul p a b = trav [map (app tx ty) l :/: map (app tx ty) r
10                     | (x,tx) ← a, (y,ty) ← b
11                     , let l:/:r = combine p x y]
12     where trav :: [Pair [a]] → Pair [a]
13           trav [] = pure []
14           trav (x:xs) = (++) <$> x <*> trav xs
15           app tx ty (z,f)  = (z, f tx ty)
```

Figure 2.10: Monoid instance for SomeTri, and RingP instance for the parser data

The call to combine in figure 2.10 is the programming version of checking if there exists a rule such that $A = BC$ in the grammar.

2.3.3 Dependently typed programming with charts

The `SomeTri` type is the type representing upper-triangular matrices in the code. Merging these matrices is the central to the parsing process. Merging employs extensions to the code used as a reference to the paper by Bernardy and Claessen. The matrix code, available from BNFC as a library called `Data.Matrix.Quad` is written in a dependently typed fashion.

First, the matrix type `Mat` is dependent on another type, `Shape`, that describes the shape of a matrix as a binary tree.

```
1  data Shape = Bin Shape Shape | Leaf
2
3  data Mat :: Shape → Shape → ∗ → ∗ where
4    Quad :: !(Mat x1 y1 a) → !(Mat x2 y1 a) →
5            !(Mat x1 y2 a) → !(Mat x2 y2 a) →
6            Mat (Bin x1 x2) (Bin y1 y2) a
7    Zero :: Mat x y a
8    One :: !a → Mat Leaf Leaf a
9    Row :: Mat x1 Leaf a → Mat x2 Leaf a → Mat (Bin x1 x2) Leaf a
10   Col :: Mat Leaf y1 a → Mat Leaf y2 a → Mat Leaf (Bin y1 y2) a
```

Figure 2.11: The `Mat` type with its dependent `Shape` type. Note that shapes are used both for x-axis and y-axis size

In a setting without using FingerTrees and Monoids, such as the reference implementation from the paper, it is possible to merge matrices using a single element as glue. Such an approach simplifies the merging a lot, since elements are placed just above the diagonal and that means a single element can fill the small void in the merged matrix, as illustrated in figure .

Till vänster, mergein MED element Till höger, merge UTAN element

Figure 2.12: `merge` with and without a single element as glue

Because of the absence of an extra element, the existing mergein function could not be used, but a merge function had to be implemented. Observing that the V function from Valiant's algorithm that computes the uppermost matrix when merging uses a single element, we want to imitate that behaviour. The solution: chop off the first row in the second argument, and recompute all but the leftmost elements when applying V.

TODO: Figur här som förklarar chop-grejen.

```
1  merge :: Bool → SomeTri a → SomeTri a → SomeTri a
2  merge p (T y l) (T x r) = chopShape x $ λchopper x' →
3      let (rTopL, rL') = chopFirst chopper (leftOf r)
4          (rTopR, rR') = chopFirst chopper (rightOf r)
5          cdp = closeDisjointP p (leftOf l)
6                  (mkLast' y $ sequenceA (rTopL :/: rTopR)) rR'
7      in T (bin' y x') (quad' l cdp zero (rL' :/: rR'))
```

Figure 2.13: merge without middle element

Looking at the code in figure 2.13, there are a few things to note before we look more deeply into this. First, `chopShape` operates on the x-axis shape in the matrix, returning a continuation containing `chopper` and a new x-shape. `chopper` is of type `ChopFirst`, which provides an easy way to pattern match on the x-shape of a `Mat`, as we can se if we look at `chopFirst`, which utilises this.

```
1  data ChopFirst x x' where
2    Stop :: ChopFirst (Bin Leaf x) x
3    Continue :: ChopFirst x x' → ChopFirst (Bin x x0) (Bin x' x0)
4
5  chopFirst :: ChopFirst x x' → Mat x x a
6                              → (Mat x' Leaf a, Mat x' x' a)
7  chopFirst _ Zero = (Zero,Zero)
8  chopFirst Stop (Quad a b c d) = (b,d)
9  chopFirst (Continue q) (Quad a b c d) =
10   let  (e, a') = chopFirst q a
11        (b',f)  = chopFirstRow q b
12   in (row e f,quad a' b' zero d)
```

Figure 2.14: ChopFirst type and corresponding function

Note that chopFirst, as seen in figure 2.14 does not match the `Col`, `Row` or `One` constructors. For the `One` case it is quite obvious because we cannot chop a 1x1 matrix. For the other two, this is due to the type of `chopFirst`, where the input is a square matrix: `Mat x x a`. Because both `Col` and `Row` cannot be square (unless they're 1x1, in which case the `One` construct should be used instead, which cannot be chopped anyway), there is no need to check for them, and actually writing those cases would trigger a type error when compiling.

Finally, before calling `closeDisjointP` (which represents the V function from Valiant's algorithm), we throw away all but the lower leftmost value,

using `mkLast'`. The resulting new matrix is a quad, where the left matrix is left untouched, the right is chopped and a new matrix that serves as top-right is computed. As usual, all values below the diagonal are zero.

TODO: Explain the recomputation of one row. TODO: Figur, typ fyrfältare. Left, CDP, Zero, ChoppedRight.

2.3.4 Oracle and unsafePerformIO

The use of an oracle presented a bit of a problem in implementing a monoid instance for the parser, for the simple reason that it is very hard to simply pick a bool at random in Haskell without it being always `True` or always `False`. The call to `merge` in `mappend` illustrates this clearly:

```
1 instance Monoid (SomeTri a) where
2     t0 `mappend` t1 = merge True t0 t1
```

Figure 2.15: Parser Monoid instance before random oracle

The call to `merge` requires a `Bool` acting as the oracle as an argument. Since a Monoid has no context outside its own type, it is hard, if not impossible, to generate a Bool using only the SomeTri type. One could argue for creating a newtype wrapper around a tuple of a SomeTri and a StdGen, used to generate the bool at each step, but that only moves the problem to the mempty call, where a fresh StdGen would have to be picked at each instance.

The solution to this problem came in the form of a call to unsafePerformIO. While this is controversial, it was also not completely obvious to implement. If an unsafe call to randomIO was made separately from the merge call, this call would be evaluated only once, rendering the solution useless. The trick here was to put the whole call inside an unsafe wrapper, so that the call to merge, and with that the call to randomIO, became dependent on the input.

```
1 instance Monoid (SomeTri a) where
2     t0 `mappend` t1 = unsafePerformIO $ do
3       b ← randomIO
4       return $ merge b t0 t1
```

Figure 2.16: Parser Monoid instance with oracle

Now usually, for `unsafePerformIO` to be safe one should make sure that the call is free from side effects and *independent of its environment.* [Com].

Since none of those two requirements are fulfilled here this calls for discussion. In general, one does not want a call to unsafePerformIO to be evaluated more than once – but in this case this is a requirement for the code to behave as expected, and that's why it is indeed dependent on its environment. The only side effect in this snippet is the use of the global random number generator and that should not affect any other part of the program, and can thus be considered safe.

Chapter 3

Results

We have managed to write a parser that is incremental and which, when given correct input, produces correct output in the form of an AST. The parser, and the accompanying lexer can be generated by BNFC. When given incorrect input however, the output is not especially satisfactory. The lexer can tell if an incorrect token is part of the input, but it cannot tell where in the input that token is placed. The parser can also recognise that the input tokens does not follow the grammar of the language, but it cannot give any information about where the syntactic error was made.

3.1 Testing

3.2 Measurements

How fast is it? What is the complexity?

Chapter 4

Discussion

4.1 Pitfalls

Look through the LOG to remember whatever happened. Describe sort of chronologically?

4.1.1 Too many result branches

When the parser was finally working, there was a bug that seemed a bit weird. Whenever a file was successfully parsed, the parser returned a number of results, from 4 to 1024, all identical to each other. This led us to believe that there was branching done in places where no branching was motivated. It turned out to be due to a bug in `merge`, where the row that was chopped (see 2.3.3). Initially, `merge` was written so that the chopped off row was included as a part of the upper-right matrix as an argument to the V function. This led that row to be combined with itself, because that row had already been computed using the V function. The solution was to remove all but the first element, so that nothing would be recomputed.

* Describe the other bug, unknown source.

4.2 Future work

The result of this project is a parser that can parse correct input and does that well. There are two main features missing however; position information on tokens, and good error reporting.

4.2.1 Position information

Position information for tokens is a feature that is currently missing in the parser, much due to the fact that it is missing in the lexer. Discussions with Hannson & Hugo revealed that this is due to that not being a priority. The most likely way to implement position information would be by using relative positions for tokens, because of the tree structure where nodes are not aware of each other. That way, position information, or lexical errors, can be promoted using mappend. There are, however several ways to integrate the relative positions into the structure, but the most obvious would be to create a newtype wrapper for tuples where the lexer state and position information are dependent on each other, as opposed to the monoid instance for regular tuples.

4.2.2 Error information

Related to the issue of position information, the error reporting in the lexer and parser is poor to say the least. Invalid tokens are reported by the lexer, but invalid syntax is only reported by saying that there were more than one parse result. For the lexer, the only thing missing in error reporting is the said position information. This is of course true for the parser as well, but due to the structure of the parser it is harder to know where an error was made.

The reason for why it is hard to know where an error was made owes to the matrix structure and how rules are combined as $A ::= BC$. Using the CYK algorithm, it would be possible to have overlaps in the parse results (where one would have to choose one to move further, as shown in figure **??**), and an error in the middle of a code snippet could lead to the parsing resulting in many small results that lack structural *glue*, as shown in figure **??**.

TODO: Bild på parser-fel och varför det är lurigt. TODO: Label parseoverlap och missingglue.

Bibliography

[ALSU07] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: principles, techniques, & tools*. Pearson/Addison Wesley, Boston, 2nd edition, 2007.

[Bac59] John W. Backus. The syntax and semantics of the proposed international algebraic language of the Zurich ACM-GAMM conference. *Proceedings of the International Conference on Information Processing, UNESCO*, 1959.

[BC13] Jean-Philippe Bernardy and Koen Claessen. Efficient divide-and-conquer parsing of practical context-free languages. In Greg Morrisett and Tarmo Uustalu, editors, *ICFP*, pages 111–122. ACM, 2013.

[Ber09] Jean-Philippe Bernardy. Lazy functional incremental parsing. In Stephanie Weirich, editor, *Haskell*, pages 49–60. ACM, 2009.

[bnf] The bnf converter. `http://bnfc.digitalgrammars.com`.

[Com] The Glasgow Haskell Compiler. System.IO.Unsafe. `http://hackage.haskell.org/package/base-4.7.0.0/docs/System-IO-Unsafe.html`.

[HH14] Christoffer Hansson and Jonas Hugo. A generator of incremental divide-and-conquer lexers. Master's thesis, Chalmers University of Technology, 2014.

[HMU03] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to automata theory, languages, and computation - international edition (2. ed)*. Addison-Wesley, 2003.

[KT06] Jon M. Kleinberg and Éva Tardos. *Algorithm design*. Addison-Wesley, 2006.

[LL09] Martin Lange and Hans Leiß. To cnf or not to cnf? an efficient yet presentable version of the cyk algorithm. *Informatica Didactica*, 8, 2009.

[Val75] Leslie G. Valiant. General context-free recognition in less than cubic time. *J. Comput. Syst. Sci.*, 10(2):308–315, 1975.

[WDT76] Thomas R. Wilcox, Alan M. Davis, and Michael H. Tindall. The design and implementation of a table driven, interactive diagnostic programming system. *Commun. ACM*, 19(11):609–616, November 1976.

[You67] Daniel H. Younger. Recognition and parsing of context-free languages in time n^3. *Information and Control*, 10(2):189–208, 1967.