Implementing incremental and parallel parsing

Master of Science Thesis in Computer Science

Tobias Olausson

University of Gothenburg
Chalmers University of Technology
Department of Computer Science and Engineering
Göteborg, Sweden, June 2014
The Author grants to Chalmers University of Technology and University of Gothenburg the non-exclusive right to publish the Work electronically and in a non-commercial purpose make it accessible on the Internet. The Author warrants that he is the author to the Work, and warrants that the Work does not contain text, pictures or other material that violates copyright law.

The Author shall, when transferring the rights of the Work to a third party (for example a publisher or a company), acknowledge the third party about this agreement. If the Author has signed a copyright agreement with a third party regarding the Work, the Author warrants hereby that he has obtained any necessary permission from this third party to let Chalmers University of Technology and University of Gothenburg store the Work electronically and make it accessible on the Internet.

Implementing incremental and parallel parsing
Tobias Olausson

© Tobias Olausson, June 2014

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

Department of Computer Science and Engineering
Göteborg, Sweden, June 2014
Abstract
Using recent improvements to Valiant’s algorithm for parsing context-free languages, we present an implementation of a generator of parsers that works incrementally, that can be parallelized and generated from a grammar specification. Using a tree structure makes for both easy use of incrementality and parallelization. The resulting code is reasonably fast and handles correct input in a satisfactory way, and would be suitable for use in a text editor setting, where small changes are frequent but only should lead to minimal work.
Acknowledgements

The author would like to thank his supervisor, Jean-Philippe Bernardy, for his insights and valuable advice throughout the project. The author would also like to thank everyone hanging out in the computer science lunch room Monaden, this thesis would not have been possible without you.
# Contents

1 Background
1.1 Topics involved
- Divide-and-conquer
- Incrementality
- Parallelism
- Parsing
- Motivation
1.2 Lexing
- LexGen
1.3 Context-free grammars
- Chomsky Normal Form
- Backus-Naur Form
1.4 Parsing
- CYK algorithm
- Valiant
- Improvement by Bernardy and Claessen
1.5 Dependently typed programming

2 Implementation
2.1 Finger trees
- Measuring and Monoids
2.2 Lexing
2.3 Parsing
- Pipeline of measures
- Dependently typed programming with charts
- Merging charts
- Chopping matrices
- Oracle and unsafePerformIO
## Contents

<table>
<thead>
<tr>
<th>Chapter</th>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3 Results</td>
<td>3.1 Branching in the parser</td>
<td>26</td>
</tr>
<tr>
<td></td>
<td>3.2 Possible text editor usage</td>
<td>26</td>
</tr>
<tr>
<td></td>
<td>3.3 Testing</td>
<td>27</td>
</tr>
<tr>
<td></td>
<td>3.4 Measurements</td>
<td>27</td>
</tr>
<tr>
<td></td>
<td>3.4.1 Behaviour</td>
<td>28</td>
</tr>
<tr>
<td></td>
<td>3.4.2 Merging matrices</td>
<td>28</td>
</tr>
<tr>
<td></td>
<td>3.4.3 Total running time</td>
<td>29</td>
</tr>
<tr>
<td>4 Discussion</td>
<td>4.1 Pitfalls</td>
<td>30</td>
</tr>
<tr>
<td></td>
<td>4.1.1 Too many parse results</td>
<td>30</td>
</tr>
<tr>
<td></td>
<td>4.1.2 Loosen constraint on Matrix type</td>
<td>31</td>
</tr>
<tr>
<td></td>
<td>4.2 Future work</td>
<td>31</td>
</tr>
<tr>
<td></td>
<td>4.2.1 Position information</td>
<td>31</td>
</tr>
<tr>
<td></td>
<td>4.2.2 Error information</td>
<td>31</td>
</tr>
<tr>
<td></td>
<td>4.3 Conclusions</td>
<td>32</td>
</tr>
<tr>
<td>Bibliography</td>
<td></td>
<td>34</td>
</tr>
<tr>
<td>A Javalette Light</td>
<td>A.1 LBNF grammar</td>
<td>36</td>
</tr>
<tr>
<td></td>
<td>A.2 Sample code</td>
<td>38</td>
</tr>
</tbody>
</table>
Chapter 1

Background

This thesis is about parsing in an incremental fashion that can easily be parallelized, using a divide-and-conquer approach. First, we look at the involved topics, and later in this chapter, the concept of dependently typed programming is discussed, because it is a technique used in the implementation of the parser.

1.1 Topics involved

In this section, we give a brief explanation of the topics involved in the implementation of this parser algorithm, and end with a motivation for why this is interesting in the first place.

Divide-and-conquer One important class of algorithms in computer science are divide-and-conquer algorithms. The name refers to the technique of breaking down a problem into sub-problems, where the same rule is applied recursively (the divide step). Each sub-problem can be solved independently, and the results of the sub-problems are then combined, finally becoming the result of the initial problem (the conquer step) [Kleinberg and Tardos, 2006, p.209]. A typical example is mergesort, where a list of elements is broken down to lists of single elements (trivially sorted), that are then combined using an improved insertion sort, observing that each sub-list is sorted. It was shown by Bird [1987] that the conquer step has to be associative, so that grouping of items does not effect the outcome of the algorithm.

Trees are a class of data structures that are especially suited for divide-and-conquer algorithms, because of their structure as trees with subtrees, naturally following the divide-step. To conquer is just to reduce, or in functional programming terms: fold, the tree. It was shown by Bernardy and
1. Background 1.1. Topics involved

Claessen [2013] that for trees of symbols in a finite alphabet, such a reduction can be made in a way that is associative, thus preserving the structure of the input.

Incrementality In an interactive system, such as a text editor, one typically want to do as little work as possible when some input in the system is changed. For the text editor case, we do not want to recompute all syntax highlighting information when just one word is changed. Techniques capturing this behaviour are said to work incrementally, and was perhaps first described by [Wilcox et al., 1976]. In a setting with lazy evaluation, it can be especially interesting to use, as shown by [Bernardy 2009].

Parallelism With computer architectures being parallel these days, the ability to run many threads simultaneously has become reality. Writing code that can be parallelized is crucial to make use of these features, and thus having code that run physically in parallel. Because 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 special way to permit parallelization. While our work is not explicitly written to run in parallel, the algorithm can be easily parallelized, as goes for the main third party library used.

Parsing To parse is to check if some given input corresponds to a certain language’s grammar, and if so, return a parse tree of that input. In this thesis we will use context-free grammars to describe 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.

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. 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, real-time feedback in the form of an AST could easily be provided to the programmer. Most current text editors give syntax feedback based on regular expressions, which does not yield any information about, for example, nesting or the surrounding AST. A fast incremental parser, can also be connected to a type checker to get even more information, possibly in real-time, while
not having to recompile the whole file to get such information, but any real world use cases are beyond the scope of this thesis.

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 [Aho et al. 2007, p. 5, p. 109]. The lexemes are wrapped in tokens that denote what function and position each lexeme relates to. 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. 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 will be forwarded as tokens to the parser.

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

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

1.2.1 LexGen

A generator for incremental divide-and-conquer lexers was developed by Hansson and Hugo [2014] as a master’s thesis. The aim of the present thesis is to write an incremental divide-and-conquer parser, so their work is well-suited as a starting point, and as something to build on. Their lexer uses Alex [Marlow 2003] for core lexing routines and relies 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.
1. Background 1.3. Context-free grammars

Formally, a context-free grammar is a 4-tuple: \( G = (V, \Sigma, P, S) \) [Hopcroft et al., 2003, p.171]. \( V \) is a set of non-terminals, or variables. \( \Sigma \) is the finite 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 constitute the recursive definition of the language. \( S \) is the start symbol, where \( S \in V \).

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

\[
\alpha A \beta \Rightarrow_G \alpha \gamma \beta \text{ iff. } (A ::= \gamma \in P)
\]

\[
L(G) = \{w \in \Sigma^* \mid S \xrightarrow{G} w\}
\]

That is, all words in the language that can be derived by recursively applying rules from the grammar when starting from the start symbol (denoted by the double arrow: * for closure, \( G \) for the grammar) [Hopcroft et al., 2003, p. 177]. A language \( L \) is said to be context-free if there is a context-free grammar \( G \) that recognises the language, meaning that \( L = L(G) \).

We can exemplify by using a simple made-up language of if-then-else clauses. The language terminals are \( \text{if, then, else, true and false} \). 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 \( \text{true} \) would not be a string of this language. The formal definition of this language can be seen in figure 1.2. Note that \( P \) is defined by the rules for \( I \) and \( R \), and each production (partially) defines a variable and contains terminals or symbols on its right-hand side [Hopcroft et al., 2003, p.171].

\[
G = (V, \Sigma, P, I)
\]
\[
V = \{I, R\}
\]
\[
\Sigma = \{\text{true, false, if, then, else}\}
\]
\[
P = \{
    I \rightarrow \text{if } R \text{ then } R \text{ else } R
   , R \rightarrow I
   , R \rightarrow \text{true}
   , R \rightarrow \text{false}
\}
\]

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 canonical way to write context-free grammars that was first described by [Chomsky 1959]. Productions in CNF are restricted to the following forms:

\[
\begin{align*}
A & \rightarrow BC, \quad A \text{ is a variable, } B \text{ and } C \text{ are variables} \\
A & \rightarrow a, \quad A \text{ is a variable, } a \text{ is a terminal symbol}
\end{align*}
\]

Figure 1.3: Rules allowed in Chomsky Normal Form

Because 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 grammars in CNF in the first place [Lange and Leiß 2009].

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 Backus-Naur form [Backus 1959], where each rule is written on the following form:

\[
\begin{align*}
\text{Variable} & := \text{Production}
\end{align*}
\]

However, for practical programming purposes, one usually uses a labelled version of BNF, where the rules are extended to be on this form:

\[
\begin{align*}
\text{Label. Variable} & := \text{Production}
\end{align*}
\]

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) [Ranta and Forsberg], 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 programming languages, for the language described in that grammar. According to its documentation, usage of BNFC saves around 90% of source code work in writing a compiler front-end.

1.4 Parsing

The technical role of the parser is, given a list of tokens, to determine if those tokens can be written in that order for a specific language, and if so, return
an abstract syntax tree corresponding to that list of tokens. More formally, and connecting to the language of a context-free grammar above, the parser is given a string \( w \) and checks if

\[
w \in \Sigma^*, S \xrightarrow{A} w
\]

that is, to check if, and how, 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 [Aho et al., 2007, p.192]. This project will use an improved version of the CYK algorithm, a bottom-up parser.

### 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 [Younger, 1967]. The algorithm works on a context-free grammar in CNF, and yields a matrix with the following properties, as stated by Younger [1967].

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.

Note that \( N \) refers to the set of variables, which we denote as \( V \). The algorithm creates a square matrix \( W \) of dimension \(|S_t| + 1 \). It then computes the rest of its entries using dynamic programming and the definitions below.

\[
W_{i,j} = \begin{cases} 
\emptyset & \text{if } j \leq i \\
\{A | (A := S_t[i]) \in P \} & \text{if } j = i + 1 \\
\bigcup_{k=i+1}^{j} W_{ik} \cdot W_{kj} & \text{otherwise} 
\end{cases} \quad (1.1)
\]

\[
x \cdot y = \{A | A_0 \in x, A_1 \in y, (A := A_0 A_1) \in P \} \quad (1.2)
\]

As we can see, just above the diagonal of \( W \), we place the set of 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 on the form $A ::= BC$ as seen in equation 1.2. A graphical representation of these rules in action is shown in figure 1.4. The input string is placed just above the diagonal, and matching against the grammar rules are then applied recursively using dynamic programming.

![Figure 1.4: Upper-triangular matrix for which the CYK algorithm has been applied. $X ::= S_t[i..j]$ and $Y ::= S_t[j..k]$ are terminal rules in the grammar this matrix is built from, and $Z ::= XY$ is a nonterminal one](image)

### 1.4.2 Valiant

Valiant improved the CYK algorithm by showing that context-free recognition can be reduced to matrix multiplication of boolean matrices [Valiant, 1975]. This was done by first reducing recognition to the transitive closure of upper-triangular matrices. The closure of a matrix $W$, denoted $W^+$, is defined as the matrix $C$ such as $C = C \cdot C + W$. Valiant then showed that closure could be reduced to matrix multiplication by employing a divide-and-conquer approach, and furthermore only having to consider boolean matrices.

Most important for this project, the step to reduce closure to matrix multiplication described a recursive function that we will call $V$, such that given input matrices $A$ and $B$, and a partial matrix $X$ only used for strings that starts in $A$ and ends in $B$, it computes a matrix $Y$, where $Y = AY + YB + X$ [Bernardy and Claessen, 2013]. The function $V$ is responsible for parsing $Z ::= XY$ in figure 1.4 above.

### 1.4.3 Improvement by Bernardy and Claessen

Bernardy and Claessen [2013] showed that for many inputs, most of the sets in the matrix in Valiant’s algorithm would be empty. By optimizing the algorithm to handle empty matrices as a special case and avoiding multiplication
of those empty matrices, they managed to lower the time complexity of the
algorithm from that of matrix multiplication, which is $O(n^y)$, $2 \leq y \leq 3$ to
$O(\log^3 n)$.

In the same article, another improvement that regarded sequential input,
such as lists of statements in a while loop, was made. Such input can have
rules as

$$Stms ::= \epsilon$$
$$Stms ::= Stm Stms$$

which are by nature linear and do not fit well for parallelization. The
solution is to introduce tagging of all non-terminals that indicated if they
should be on the left (tagged 0) or right (tagged 1) side of the tree, and
then adding a new rule for constructing the whole tree of the nonterminal:
$$Y ::= Y^0Y^1.$$ This restricts the number of branches that can be explored,
and thus helps to avoid the otherwise linear behaviour of such rules. The
tagging bit should be selected by an oracle, so the algorithm must behave
the same regardless of how each bit is set. In practice, the bit can be set
by using a random number generator, which is what is done in this project
and is discussed more in section 2.3.3 or one could simply use an alternating
stream of 0s and 1s, which is what the reference implementation uses.

1.5 Dependently typed programming

In this project, the implementation of the parsing algorithm uses dependent
types 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 typing is useful and
saves debugging time, it usually does not say anything about the contents of
the values, at least that is the case in Haskell.

A motivating example often used is the 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. Background

1.5. Dependently typed programming

```haskell
data Vec a = Nil | V a Vec
head :: Vec a -> a
head Nil = error "empty\_vector"
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 not only other types (i.e. when using polymorphism), but also values. That way, a type can be dependent on a value of some other type that is embedded in the first one. While standard Haskell is not a dependently typed programming language, there are ways to use dependent types even in Haskell, for example by enabling the DataKinds extension to GHC. For our vector example that would look something like this:

```haskell
data Nat = Z | S Nat
data Vec a n where
  Nil :: Vec a Z
  V :: a -> Vec a s -> Vec a (S s)
head :: Vec a (S b) -> a
head Nil = error "empty\_vector" -- this does not type-check
head (V a _) = a
```

Figure 1.6: Dependently typed vector with head function. Note that the DataKinds extension for GHC is needed for this to work.

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, because 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 is to implement a parser that would be incremental and easily parallelizable. 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 the details of the implementation, there are a couple of libraries and programming techniques one has to be familiar with before moving forward. We will first describe those, and then move on to describe changes to the lexer we inherited from [Hansson and Hugo 2014], and the implementation of the parser.

The source code for this project can be found both in the Quad.hs module in a fork of BNFC [Bernardy and Olausson 2014], but also in source/src/BNFC/Backend/Haskell/ directory in the same repository, where the files CFToAlex3Incremental.hs and ToCYK.hs are the most significant contributions. Before integration to BNFC, work was done in a separate repository [Olausson 2014].

2.1 Finger trees

A finger tree is a finite data structure with logarithmic access time and concatenation time. 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 necessarily add a new branch to the tree [Hinze and Paterson 2006]. The tree structure makes the data structure suitable for a divide-and-conquer algorithm.

A Haskell implementation suitable for general use exists in the package Data.Sequence, and a more general structure is available in the package Data.FingerTree. 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. Except for the concepts related to measuring, the reader can think of these as regular balanced binary trees.

### 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, monoids are provided by writing instances of this type class (available in *Data.Monoid*):

```haskell
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:

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

Figure 2.2: Monoid instance for lists

The *FingerTree* type has a notion of *measure* on 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 existence of a measured instance is ensured by the *FingerTree* API in *Data.FingerTree*.
2. Implementation  

2.1. Finger trees

```haskell
class Monoid v ⇒ Measured v a | a → v where
    measure :: a → v

-- FingerTrees are parametrized on both v (measures) and a (values)
data FingerTree v a

-- Create an empty finger tree
empty :: Measured v a ⇒ FingerTree v a
```

Figure 2.3: Measuring and the FingerTree type. The Measured class is constrained on both the existence of a monoid instance and the existence of a functional dependency between \(a\) and \(v\), so that the type \(v\) can be uniquely determined from having only type \(a\) [Jones, 2000].

This means that, in order to use the FingerTree, one need to fulfil a few criteria first. Let us 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 String and Int, by writing an instance of Measured for String Int. For our use case this is just the length function. However, you also need a Monoid instance for Int. In this case, the following definitions would give us the desired behaviour:

```haskell
type MyTree = FingerTree Int String
instance Measured String Int where
    measure = length
instance Monoid Int where
    mempty = 0
    mappend = (+)
```

Figure 2.4: One possible measure from String to Int

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

An important feature of the FingerTree type, especially in an incremental setting such as in a text editor, is that measures are cached at each node. An update at one node does not force recomputation of the measure for the
whole tree, but only for the nodes leading to the changed node, which are no more than $O(\log n)$, i.e. the height of the tree.

### 2.2 Lexing

For lexing code into tokens, the results from the LexGen project was used. Some modifications had to be done to LexGen in order to be easily combined with an incremental parser. In the LexGen code, the output structure is a *Sequence* of tokens. Because *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, where we want to do several transformations at the nodes. Hence, instead of outputting tokens as a *Sequence*, the code was changed to output another *FingerTree*, from which the tokens could then be measured (see section 2.3.1 below).

### 2.3 Parsing

There is an existing reference implementation in BNFC for the optimisation to Valiant’s algorithm, that can be accessed using the `--cnf` flag [Bernardy and Claessen, 2013]. That option generates large tables needed for combining different tokens. Because this project is similar to the reference implementation, but with an incremental approach, it is natural to generate the new parser by using a new flag.

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 automaton needed for the lexer to work. The pipeline to obtain an abstract syntax tree is as follows:

1. Input to lexer are characters placed in a finger tree
2. The characters are measured. The measure is a data structure containing another finger tree of tokens
3. Measure each node in the finger tree of tokens into a representation of upper-triangular matrices.
4. When the measure is `mappend`ed, the matrices are merged, using the improved Valiant’s algorithm.
5. The resulting AST will be whatever is found at the topright position in the matrix
We will therefore first look at the pipeline from Char to AST, and then we will dive into the code for merging matrices, that being the core of this project.

2.3.1 Pipeline of measures

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.

\[
\text{instance } (\text{Measured } v \text{ IntToken}) \Rightarrow \text{Measured IntToken Char where}
\]

Figure 2.5: Diagram of measuring pipeline. The type signature for the measured instance in the lexer shows the constraint: to measure a Char into an IntToken, one has to be able to measure from IntToken to some type \(v\), which is defined as SomeTri, a type for upper-triangular matrices, in the parser.

We will describe what SomeTri is in more detail in section 2.3.2 so for now it can be thought of as the internal parser state. 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.

\[
\begin{align*}
\text{test} :: \text{FilePath} & \rightarrow \text{IO [(CATEGORY,Any)]} \\
\text{test filename} & = \text{do} \\
& \quad \text{file} \leftarrow \text{readFile filename} \\
& \quad \text{let lexed} = \text{measure $ makeTree file} \\
& \quad \text{parsed} = \text{measure $ stateToTree lexed} \\
& \quad \text{return (results parsed)}
\end{align*}
\]

Figure 2.6: Code showing the measuring pipeline

Note that figure 2.5 is restricted to a single character. 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 internally in the finger tree 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.

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 see how it works in figure 2.7.

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

Figure 2.7: Measure from token to upper-triangular matrix. The T construct guarantees a square matrix of a given size. The call to quad makes sure the observation about empty matrices by Bernardy and Claessen [2013] is handled properly when creating a matrix. The `intToToken` function is an auxiliary which is due to implementation details in the lexer. The operator `/:` is a constructor for a type `Pair` used to capture the behaviour with oracle. The functions `leftOf` and `rightOf` are used to deconstruct a `Pair`.

We create a 2x2 matrix, and place the lexed 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 2.8. In the zero case, the call to quad enables the optimisation for empty matrices by pattern matching and possibly choosing another matrix constructor (all constructors are shown in figure 2.11 later).
2. Implementation

2.3. Parsing

![Graphical representation of the two possible cases for measuring to matrices in the parser.](image)

Figure 2.8: Graphical representation of the two possible cases for measuring to matrices in the parser. \{Tok\} represents the set of all rules $A$ such that $A ::= \text{token}$ where token is the lexed token.

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. The mul function is defined in the typeclass RingP, our instance uses the combine tables generated by the reference implementation in BNFC.

```haskell
instance RingP a ⇒ Monoid (SomeTri a) where
  mempty = T Leaf' (Zero :/: Zero)
  t0 'mappend' t1 = unsafePerformIO $ do
    b ← randomIO
    return (merge b t0 t1)

instance RingP [(CATEGORY, Any)] where
  mul p a b = trav [map (app tx ty) l :/: map (app tx ty) r |
    (x,tx) ← a, (y,ty) ← b
  , let l :/: r = combine p x y]
  where trav :: [Pair [a]] → Pair [a]
    trav [] = pure []
    trav (x:xs) = (++) <$> x <*> trav xs
    app tx ty (z,f) = (z, f tx ty)
```

Figure 2.9: Monoid instance for SomeTri, and RingP instance for the parser data

The call to combine in figure 2.9 is the programming version of checking if there exists a rule on the form $A ::= BC$ in the grammar.
2. Implementation

2.3. Parsing

Figure 2.10: Graphical representation of lexing and parsing using trees, with the measured parts enclosed in rectangles. This is still a simplification, all calls to \texttt{mappend} are not necessary in order to move forward in the process, and it is indeed possible to get an AST from just one character by just measuring at one leaf, as shown in figure 2.5. Remember that measures are cached at each node in the finger trees, so changing one char will not cause a complete recomputation.
2. Implementation  2.3. Parsing

2.3.2  Dependently typed programming with charts

When merging the matrices, combining elements to create new, larger matrices, it is important to keep the sizes of these matrices correct to avoid bugs that would be hard to catch otherwise. The way this is done in the library available in BNFC is by using dependent types [Bernardy and Olausson, 2014]. The existing code for merging could not be used, but had to be extended to work in the tree/monoid setting. More on this in section 2.3.2.1.

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

\begin{verbatim}
data Shape = Bin Shape Shape | Leaf

data Mat :: Shape → Shape → * → * where
  Quad :: !(Mat x1 y1 a) → !(Mat x2 y1 a) →
         !(Mat x1 y2 a) → !(Mat x2 y2 a) →
         Mat (Bin x1 x2) (Bin y1 y2) a
  Zero :: Mat x y a
  One :: !a → Mat Leaf Leaf a
  Row :: Mat x1 Leaf a → Mat x2 Leaf a → Mat (Bin x1 x2) Leaf a
  Col :: Mat Leaf y1 a → Mat Leaf y2 a → Mat Leaf (Bin y1 y2) a

data SomeTri a where
  T :: Shape’ s → Pair (Mat s s a) → SomeTri a
\end{verbatim}

Figure 2.11: The \texttt{Mat} type with its dependent \texttt{Shape} type. Note that shapes are used both for x- and y-axis size. Also note the \texttt{SomeTri} type that captures two square matrices, used for the oracle functionality, which is described more in section 2.3.3.

Here are some example matrices, just to get a feel for how they are constructed. Pay attention to the width of the lines, used to show how constructors are grouped inside the matrices.
2. Implementation

2.3. Parsing

2.3.2.1 Merging charts

In a setting without using finger trees and monoids, such as the reference implementation by Bernardy and Claessen [2013], where this was implemented as the `mergein` function, it is possible to merge matrices using an additional single element as glue. Such an approach simplifies the merging a lot, because 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 2.13.

Figure 2.13: Merging with and without a single element as glue.

Because of the absence of an extra element, the existing function `mergein`
2. Implementation

2.3. Parsing

Figure 2.14: Successful merge without a middle element. The first row on the right matrix is chopped off, we discard elements D and E and put A in the leftmost bottom position, placing it just above the diagonal as wanted.

could not be used, but a merge function had to be implemented. Without the extra element, the diagonal would be broken, and the algorithm would not be able to move forward. We thus want to imitate the behaviour of having a middle element. The solution: chop off the first row in the second argument, and recompute all but the leftmost elements when applying the V function from Valiant’s algorithm.

2.3.2.2 Chopping matrices

Before looking at the actual merge code, we should look at how the chopping works. By pattern matching on the Shape in our SomeTri we can get a data structure where the relation between a larger and smaller matrix can be expressed. This data structure is ChopFirst. Once we obtain such a value, we can pattern match on it to control our recursion for chopping, and thus being able to guarantee that the chopped matrix is exactly one row smaller, and that the chopped row has height 1. This can be seen in figure 2.15. To make it clear, these are the steps:

1. Call chopShape to obtain ChopFirst value (see figure 2.15).
2. Use ChopFirst value to chop off row and column in matrix, returning chopped off row and smaller matrix (see figure 2.16).
3. Discard all but the leftmost element in the chopped off row, and extend the row to match the sizes of the input matrices, done by mkLast (see figure 2.17).
4. Use chopped off row and smaller matrix to apply Valiant’s V function, called closeDisjointP here, facilitating the merge (see figure 2.18).
2. Implementation

2.3. Parsing

\[
\begin{align*}
\text{chopShape} & : \text{Shape'} x \\
& \quad \rightarrow (\forall x'. \text{ChopFirst } x x' \rightarrow \text{Shape'} x' \rightarrow k) \rightarrow k \\
\text{chopShape Leaf'} k &= \text{error } "\text{chopShape: can't chop!}" \\
\text{chopShape } (\text{Bin'} \_ \text{Leaf'} y) k &= k \text{ Stop } y \\
\text{chopShape } (\text{Bin'} \_ y1 y2) k &= \\
& \quad \text{chopShape } y1 \$ \lambda q y1' \rightarrow k \ (\text{Continue } q) \$ \text{bin'} y1' y2 \\
\quad \quad \text{-- intuitively, } x = x' + 1 \\
\text{data } \text{ChopFirst } x x' \where \\
\text{Stop } &= \text{ChopFirst } (\text{Bin } \text{Leaf } x) x \\
\text{Continue } &= \text{ChopFirst } x x' \rightarrow \text{ChopFirst } (\text{Bin } x x0) (\text{Bin } x' x0)
\end{align*}
\]

Figure 2.15: The chopShape function, returning a continuation with both an instance of ChopFirst together with a smaller (chopped) Shape.

The result of running chopShape is used when chopping off the first row and first column in a given matrix. The column is discarded, since it is always empty due to being below the diagonal, while the row is returned together with a smaller matrix. This is shown in figure 2.16.

\[
\begin{align*}
\text{chopFirst} & : \text{ChopFirst } x x' \rightarrow \text{Mat } x x a \\
& \quad \rightarrow (\text{Mat } x' \text{ Leaf } a, \text{Mat } x' x' a) \\
\text{chopFirst } \_ \text{ Zero } &= (\text{Zero}, \text{Zero}) \\
\text{chopFirst } \text{ Stop } (\text{Quad } a b c d) &= (b,d) \\
\text{chopFirst } (\text{Continue } q) (\text{Quad } a b c d) &= \\
& \quad \text{let } (e, a') = \text{chopFirst } q a \\
& \quad \quad (b', f) = \text{chopFirstRow } q b \\
& \quad \quad \text{in } (\text{row } e f, \text{quad } a' b' \text{ zero } d)
\end{align*}
\]

Figure 2.16: The chopShape function, ChopFirst type and corresponding function.

Note that chopFirst, as seen in figure 2.16 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 have shape 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 the function corresponding to the V function from Valiant’s algorithm, we need to 1) throw away all but one value from the chopped off row, and 2) extend the row to match the size of the matrices we merge with. This is done by a function called mkLast, shown in figure 2.17.

\[
\text{mkLast} :: \text{RingP} a \Rightarrow \text{Shape’ y} \rightarrow \text{Mat x Leaf a} \rightarrow \text{Mat x y a}
\]

\[
\begin{align*}
\text{mkLast} \ (\text{Bin’ _ _ y}) \ \text{Zero} &= \text{zero} \\
\text{mkLast} \ (\text{Bin’ _ _ y}) \ (\text{One a}) &= \text{col zero} \ (\text{mkLast y (one a)}) \\
\text{mkLast} \ (\text{Bin’ _ _ y}) \ (\text{Row a b}) &= \text{quad zero zero} \ (\text{mkLast y a}) \ \text{zero}
\end{align*}
\]

Figure 2.17: Implementation of the mkLast function

We now have everything we need to create a new, larger, better matrix containing soon-to-be abstract syntax trees. The code for merge is included in figure 2.18, resulting in a new quad, where the left matrix is left untouched (as seen in figure 2.14), but we get a new, smaller, right matrix, and the top-right part is computed using Valiant’s algorithm. As usual, all values below the diagonal are zero.

\[
\begin{align*}
\text{merge} :: \text{Bool} \rightarrow \text{SomeTri a} \rightarrow \text{SomeTri a} \rightarrow \text{SomeTri a} \\
\text{merge} \ p \ (\text{T y l}) \ (\text{T x r}) &= \text{chopShape x } \lambda \text{chopper x’} \\
\text{let} \ (\text{rTopL, rL’}) &= \text{chopFirst chopper (leftOf r)} \\
(\text{rTopR, rR’}) &= \text{chopFirst chopper (rightOf r)} \\
\text{cdp} &= \text{closeDisjointP p (leftOf l)} \\
(\text{mkLast’ y}$ sequenceA (\text{rTopL} ::: \text{rTopR})) \text{ rR’}
\end{align*}
\]

Figure 2.18: The function merge without middle element

2.3.3 Oracle and unsafePerformIO

The use of an oracle as described by Bernardy and Claessen [2013] 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 boolean value at random in Haskell without it being always True or always False. The call to merge in mappend illustrates this clearly:
2. Implementation

2.3. Parsing

```haskell
instance Monoid (SomeTri a) where
  t0 'mappend' t1 = merge True t0 t1
```

Figure 2.19: First attempt at parser Monoid instance before random oracle. True in this case means that the left matrix will always be chosen, leading to linear behaviour, exactly the opposite of what we want to achieve.

The call to `merge` requires a `Bool` acting as the oracle as an argument, to choose the left or right matrix. Because 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 `SomeTri` and `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`. Not only is this 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.

```haskell
instance Monoid (SomeTri a) where
  t0 'mappend' t1 = unsafePerformIO $ do
    b ← randomIO
    return $ merge b t0 t1
```

Figure 2.20: 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. None of those two requirements are fulfilled here, so 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 is 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

Our results are as follows.

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 by using the \texttt{--incremental} flag. 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. In the rest of this chapter, we support and discuss these results.

3.1 Branching in the parser

A bug was discovered late in the project and has, due to time constraint, not been investigated. The bug consists of the parser giving too many parse results, all being identical. The number of results is deterministic and depends solely on certain constructs in the input source code. It has not been possible to pinpoint the source of this bug, but the behaviour suggests an error in \texttt{merge}, or possibly ambiguities in a given grammar.

3.2 Possible text editor usage

One of the motivations for writing this thesis was the possible usage of the parsing algorithm in a text editor, where speed is of the essence, but where one does not want to redo the whole parsing when just small parts are updated. What we can see from the results is that we have managed to achieve,
through the use of the FingerTree data structure, an incremental implementation. Furthermore, the same data structure also enables use of divide-and-conquer techniques to parallelize the algorithm.

If this algorithm was to be used in an editor, care would need to be taken when handling any parse errors, and in order to use it at all, position information is absolutely essential. This is discussed more in section 4.2. However, for real use, the algorithm is currently performing poorly, probably due to high memory consumption in the lexer, where, at each leaf, a table of each output state given every possible input state. Currently, for files larger than about 1000 lines, the runtime system will run out of stack space. Since there is no sharing between these tables, this is very inefficient. For example, the token ‘;’ is typically used in the same way at many positions in the source code. If the tables for that token could be shared, it would save both time and memory. The same is partly true for the parser as well, since some combinations of tokens are more common, the parser could save memory if some sharing was done there as well.

### 3.3 Testing

To test this parser algorithm was rather easy, since a parser conforming to the algorithm could be generated by from a LBNF grammar using BNFC. The main testing language was Javalette Light, a small subset of C, but still with the ability to create interesting parse trees. Later, both the Javalette language and C were used to test the parser. When testing with C a serious bug was discovered, this is discussed in 3.1.

Testing was not automated, but instead simple input files were used, where the source code was either correct or had some syntax error. This proved to be sufficient for this project, but for future versions, when handwritten cases might not be enough to cover all trivial uses, one would probably want to have tests generated by something like QuickCheck [Hughes et al.]. Although one would probably get quite far by being systematic and using the given grammar as a base for test cases.

### 3.4 Measurements

We have been using criterion [O’Sullivan 2009] as a benchmarking library to test the implementation.

Benchmarking included both measuring of the merge step with two previously parsed subtrees as well as measuring the time to parse input of increas-
3. Results

3.4. Measurements

ing sizes. Measuring was done on a computer running an Intel i7 processor clocked at 3.40 GHz, with a sample size of 1000.

The benchmarks were carried out in such a way that lexing was done prior to measurement of parser measurement, so that lexer performance would not affect measurement of parser performance. The complete benchmark code can be found in the project repository under `src/Test/Javalette/BenchParser.hs` [Olausson, 2014].

It should be noted that testing large inputs for this project has been hard due to large memory consumption, possibly owing to the structure of the lexer. An experiment was conducted where the lexer was switched to the standard Alex lexer generated by BNFC. These tests however, were even worse when it came to memory consumption.

Because not all tests look the same, this constraint affect different measurements differently, so the same input sizes have not been possible to use for the merge test and the more general running time test.

3.4.1 Behaviour

The input to the parser is a fingertree, generated by the lexer. We will ignore the behaviour of the lexer, because it is not the main scope of this project, but instead look a bit about how the parser should behave when it comes to time complexity. The steps taken to parse the input is to first create the initial matrix at each leaf. The time at each leaf is independent on the input size, and is therefore $O(1)$, but it is done at each leaf, so the total is $O(n)$. The second step is the merging, which at each point also does the matrix multiplications — shown by [Bernardy and Claessen, 2013] to be $O(\log^2 n)$. Implementation of the merging step was relatively straightforward, and should therefore satisfy these conditions.

3.4.2 Merging matrices

The core of the parser is the merging of matrices corresponding to previously parsed subtrees. The merge process uses the $V$ function from Valiant’s algorithm, and does matrix multiplication, so it should depend heavily on the input size. Tests designed to measure this behaviour were inconclusive due to technical limitations in memory, but for the small files we were able to test, the behaviour was similar to that of the total running time. Such a behaviour can be expected, because all parsing work is done in `merge`, and the number of calls to `merge` is linear in the input size.
3.4.3 Total running time

From stress testing the parser with large inputs, it seems that the parser behaves as expected relative to the input size, growing in a linear fashion with it. This should be expected, since $O(n)$ dwarfs $O(\log^3 n)$ for inputs just below 1000 and upwards.

![Figure 3.1: Mean running time for files where the input size denote the number of functions (of equal size) in that file. The green regression line shows that the parser more or less behaves linearly with respect to input size.](image-url)
Chapter 4

Discussion

In this chapter, we will look at some things that are not strictly part of the result, but related to it in some way. First, we will discuss some mistakes and how they were remedied, later we will look at some possible future work with the parser and lexer, and we will end with some conclusions of the project as a whole.

4.1 Pitfalls

During implementation, a few mistakes worth mentioning were made. These are discussed here.

4.1.1 Too many parse results

When the parser was finally working, there was an issue that, 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. Branching in this sense means that there are more than one possible AST for given input. In a grammar without ambiguities, this should not be possible on the top-level.

The problem turned out to be a bug in merge, in the subroutine taking care of the row that was chopped (see §2.3.2.1). 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 from Valiant’s algorithm. 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.
4. Discussion

4.1.2 Loosen constraint on Matrix type

When writing \texttt{merge}, one attempt was made at loosening the constraints on the \texttt{Row} and \texttt{Col} constructors so that they could correspond to more than one row or col, respectively. This turned out to be a bad idea when one wanted to control recursion using these constructors. Such a change also introduced an ambiguity in the semantics of matrices, because then a 4x4 matrix could be created by using \texttt{Quad} (the right way), or by using the less strict versions of \texttt{Col} (a wide column of height four) or \texttt{Row} (a tall row of width four). Having the different constructors constrained to a certain type of matrix proved easier in the end, and the extra work in terms of pattern matching was worth it to avoid the extra work needed every time a \texttt{Row} or \texttt{Col} was encountered — just to check their height and width, respectively.

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. Furthermore, the bug described in 3.1 needs to be fixed before the library can be used in a real-world application.

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 Hansson and 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 \texttt{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, or zero, parse results. For the lexer, the only thing missing in error reporting
is said position information. This is, as a consequence, 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 4.1b), 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 4.1a.

(a) Example chart where the tokens from $j$ to $k$ cannot be parsed, and therefore $C$ and $Z$ could be given as parse results. 

(b) Example chart where $A$ and $B$ overlap. Here one has to decide on how to proceed if there is no rule that can parse from $j$ to $k$.

4.3 Conclusions

In this thesis, we have seen an implementation of an incremental version of the CYK algorithm, using optimisations devised by both Valiant and Bernardy and Claessen. Given a grammar file, both the parser and a matching lexer can be generated from BNFC. The parser uses a tree structure, which, together with the CYK algorithm, would make it suitable for an implementation utilising parallelism. The tree structure, and the fact that measures in the tree are cached, also makes for efficient incremental use.

However, there is a serious bug in the code, making the parser return several identical parse trees. The parser, and the lexer, is also lacking both position information and usable error reporting. Before the library can be used, these two issues will definitively have to be dealt with. The implemen-
4. Discussion

4.3. Conclusions

tation also has unreasonably high memory requirements.
Bibliography


Aarne Ranta and Markus Forsberg. The BNF converter. [http://bnfc.digitalgrammars.com](http://bnfc.digitalgrammars.com)

The GHC Team. System.IO.Unsafe. [http://hackage.haskell.org/package/base-4.7.0.0/docs/System-IO-Unsafe.html](http://hackage.haskell.org/package/base-4.7.0.0/docs/System-IO-Unsafe.html)


Appendix A

Javalette Light

A.1 LBNF grammar

```
-- ordinary rules

Prog.    Prog ::= [Fun];
Fun.      Fun ::= Typ Ident "(" ")" [Stm];

SDecl.   Stm ::= Typ Ident ";" ;
SAss.     Stm ::= Ident "=" Exp ";" ;
SIncr.    Stm ::= Ident "++" ";" ;
SWhile.   Stm ::= "while" "(" Exp ")" [Stm];

Llt.      Exp ::= Exp1 "<" Exp1;
EPlus.    Exp1 ::= Exp1 "+" Exp2;
ETimes.   Exp2 ::= Exp2 "*" Exp3;
EVar.     Exp3 ::= Ident;
EInt.     Exp3 ::= Integer;
EDouble.  Exp3 ::= Double;

delimiters Fun "__BEGIN_PROGRAM" "__END_PROGRAM" ;
delimiters Stm "( ")" ;

-- coercions

_.        Stm ::= Stm ";" ;
coercions Exp 3 ;

TInt.     Typ ::= "int" ;
TDouble.  Typ ::= "double" ;

-- pragmas
```
internal ExpT. Exp ::= Typ "{" Exp "}" ;
comment "/=" "*/" ;
comment "///" ;
entrypoints Prog;
A.2 Sample code

```java
int main() {
    int p;
    int x;
    x = 2;
    p = 2;
    while(p < 5) {
        x = x * x;
        p++;
    }
}
```