UNIVERSITY OF GOTHENBURG

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, May 2014

Implementing incremental and parallel parsing
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

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

Abstract

Using recent improvements to Valiant's algorithm for parsing context-free languages, I present an implementation of a parser 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. It is however lacking important features such as good error reporting.

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 involved, and end with a motivation for why this is interesting to do in the first place. Later in this chapter, the concept of dependently typed programming will be discussed, because it is a vital technique in the implementation of the parser.

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 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 (obviously 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 sub-trees, naturally following the divide-step. To conquer is just to reduce the tree. It was shown by Bernardy and Claessen [2013] that such a reduction can be made for trees of symbols in a finite alphabet that is associative, thus preserving the structure of the input.

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 [Wilcox et al., 1976], 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 [Bernardy, 2009] using the Yi editor as subject.

1.1.3 Parallelism

With computer architectures being parallel these days, with the ability to run many threads simultaneosly, writing code that can be parallelized is crucial in order 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.

1.1.4 Parsing

To parse is 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 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. With a fast incremental parser, one could also

connect it to a type checker to get even more information in almost real-time, not having to recompile to get such information.

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 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. 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 valid in C

1.2.1 LexGen

A generator for incremental divide-and-conquer lexers was developed in 2013 by Hansson and Hugo [2014] as a master's thesis. The aim of this 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 utilise Alex for core lexing routines and rely 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: $G = (V, \Sigma, P, S)$ [Hopcroft et al., 2003, p.171]. V is a set of non-terminals, or variables. Σ 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 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 by recursively applying rules from the grammar and 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 recognizes 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 *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 *true* would not be a string of 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 [Hopcroft et al., 2003, 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 canonical way to write context-free grammars that was first described by Chomsky [1957]. 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

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 their 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 a **Backus-Naur form** [Backus, 1959], 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)** [et. al.], 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 [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.

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

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

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

As we can see, just above the diagonal of W, 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 on the form $A ::= BC$ as seen in equation 1.3. A graphical representation of these rules in action is shown in figure 1.4. The input string is placed on the diagonal, and matching against the grammar rules are then applied recursively using dynamic programming.

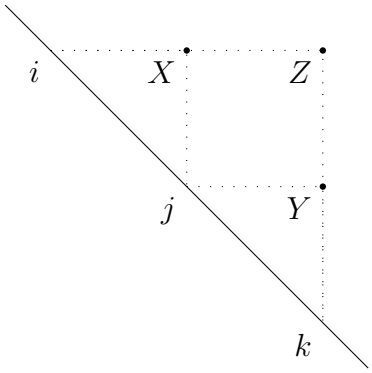


Figure 1.4: Upper-triangular matrix for which the CYK algorithm has been applied. $X ::= S_t[i..j]$ and $Y ::= S_t[j..k]$ are terminal rules in the grammar this matrix is built from, and $Z ::= XY$ is a nonterminal one

1.4.2 Valiant

Valiant [1975] 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-triangual 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.

1.4.3 Improvement by Bernardy and Claessen

In 2013, Bernardy and Claessen [2013] showed that for many inputs, most of the matrices 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 $O(n^y)$ where y is between 2 and 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. Such input could have rules as

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

which are by nature linear and would not fit well for parallelization. The solution was 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 restricted the number of branches that could be explored, and thus helped to avoid the otherwise linear behaviour of such rules. The tagging bit should be selected by an oracle, so the algorithm works no matter which 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 use just an alternating stream of 0s and 1s, which is what the reference implementation used.

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 typing 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. 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 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 the details of the implementation, 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 from Hansson and Hugo, and the implementation of the parser.

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 neccessarily 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:

```
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 *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 existance of a measured instance is ensured by the `FingerTree` API.

```
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. The `Measured` class is constrained on both the existance of a monoid instance and the existance of a functional dependency between `a` and `v`.

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 `String`s 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`. This typically is just the length function. However, for that to work you need a `Monoid` instance for `Int`. 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 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 $O(log\ n)$.

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

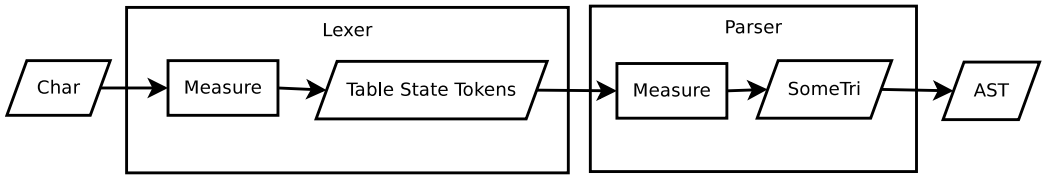
For the Haskell backend, BNFC uses Alex as a lexer, as did LexGen, so it was easy to use the LexGen core and have BNFC and Alex generate the automaton needed for the lexer to work. The pipeline to obtain an abstract syntax tree is as follows:

1. Input to lexer are characters placed in a finger tree
2. The chars 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.




```
1 instance (Measured v IntToken) ⇒ 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 later, 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.

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

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 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

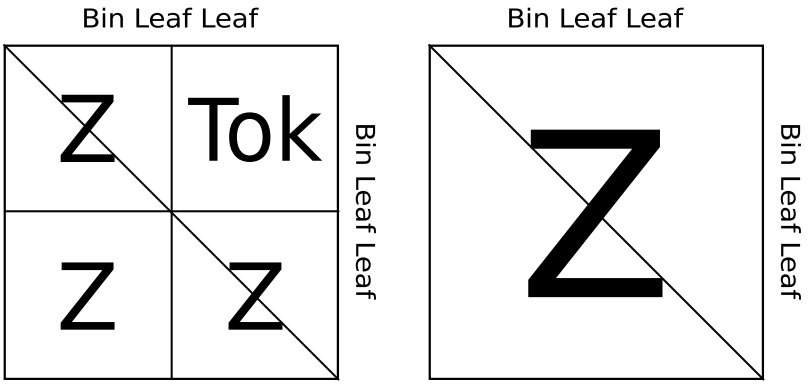
`Any` is a universal type that can hold any value, and is used as an intermediary for the generated AST.

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

```
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 2.7: 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 is handled properly when creating a matrix.

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).



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

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 $A ::= tok$ where tok 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.

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

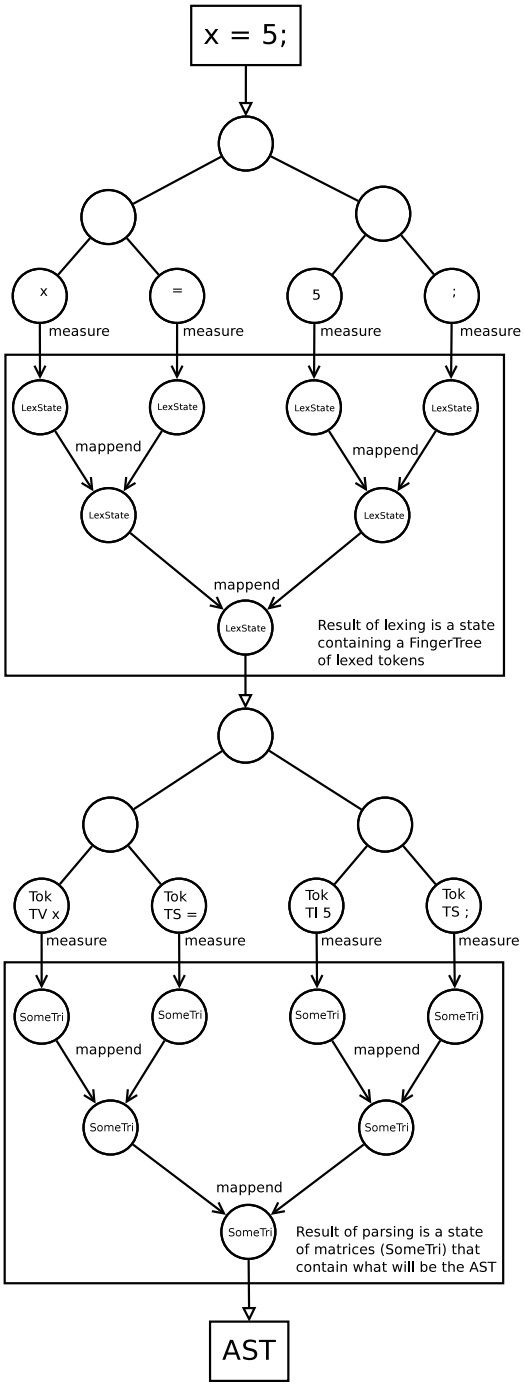


Figure 2.10: Graphical representation of lexing and parsing using trees, with the measured parts enclosed in rectangles. Note that this is still a simplification, all calls to `mappend` are not neccessary in order to move forward in the process.

2.3.2 Dependently typed programming with charts

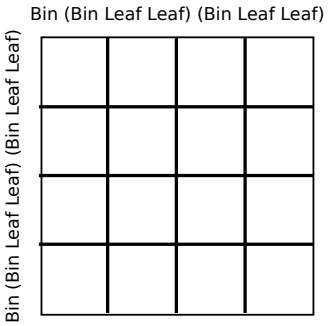
When merging the matrices, combining elements to create new, larger matrices, it is important to keep the sizes of these matrices correct to avoid bugs that would be hard to catch otherwise. The way this is done in the library available in BNFC is by using dependent types. The existing code for merging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

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

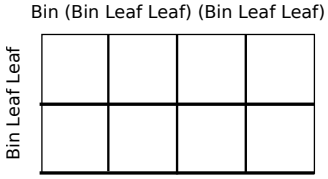
```
1  data Shape = Bin Shape Shape | Leaf
2
3  data Mat :: Shape → Shape → ∗ → ∗ where
4    Quad :: !(Mat x1 y1 a) → !(Mat x2 y1 a) →
5            !(Mat x1 y2 a) → !(Mat x2 y2 a) →
6            Mat (Bin x1 x2) (Bin y1 y2) a
7    Zero :: Mat x y a
8    One :: !a → Mat Leaf Leaf a
9    Row :: Mat x1 Leaf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

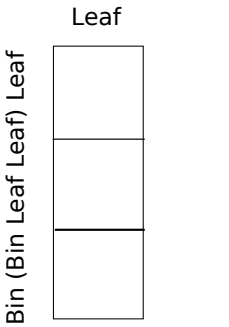
Some example matrices, just to get a feel for how they work



(a) `Quad q1 q2 q3 q4` where q1-q4 are `Quad a b c d`



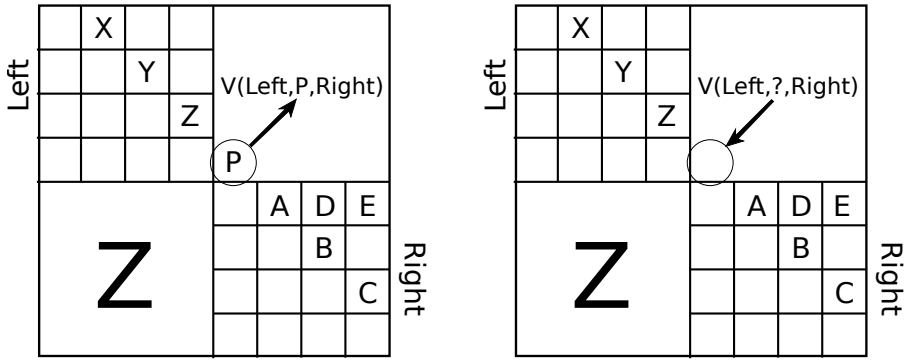
(b) `Quad r1 r2 r3 r4` where r1-r4 are `Row a b`



(c) `Col a b c`

Figure 2.12: Example matrices with their `Shape`s written out, and the `Mat` constructor used to create them

In a setting without using finger trees 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, 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.



(a) Merging with middle element (b) Merge without middle element

Figure 2.13: Merging with and without a single element as glue. Without an element the diagonal is broken – we cannot continue!

Because of the absence of an extra element, the existing mergein function 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 V.

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 give some guarantees about both the chopped matrix, and the row that was chopped. This can be seen in figure 2.15.

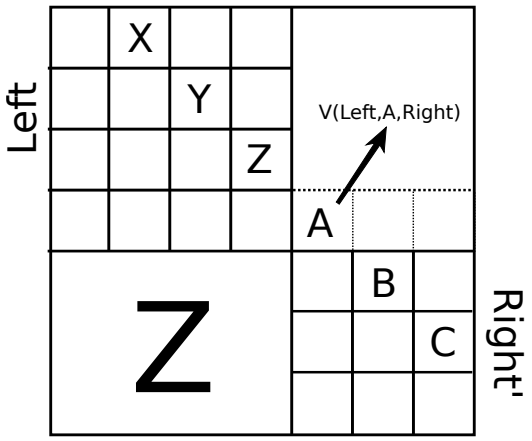


Figure 2.14: Successful merge without a middle element. The first row on the right matrix is chopped off, we discard elements D and E and put A in the leftmost bottom position, placing it just above the diagonal as wanted.

```
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 2.15: The `ChopFirst` type and corresponding function. Note that `chopFirst` discardes the first column of the matrix, as seen in the `Stop` case

Note that `chopFirst`, as seen in figure 2.15 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 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 2.16.

```
1 mkLast' :: RingP a ⇒ Shape' y → Mat x Leaf a → Mat x y a
2 mkLast' Leaf' m = m
3 mkLast' (Bin' _ _ y) Zero = zero
4 mkLast' (Bin' _ _ y) (One a) = col zero (mkLast' y (one a))
5 mkLast' (Bin' _ _ y) (Row a b) = quad zero zero (mkLast' y a) zero
```

Figure 2.16: 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 2.17, 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.

```
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 2.17: The function `merge` without middle element

2.3.3 Oracle and unsafePerformIO

The use of an oracle as described by Bernardy and Claessen 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:

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

Figure 2.18: First attempt at parser Monoid instance before random oracle.

The call to `merge` requires a `Bool` acting as the oracle as an argument. 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`. 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 2.19: 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.* [team]. 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'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

Being able to generate lexer and parser using BNFC, only needing a LBNF grammar file, testing for different language was made easy. 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 was used to test the parser. When testing with C a serious bug was discovered, this is discussed in 4.1.1.

Testing was not automated, but instead simple input files was 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 hand-written cases might not be enough to cover all trivial uses, one would probably want to have tests generated by something like quickcheck. Although one would probably get quite far by being systematic and using the given grammar as a base for test cases.

3.2 Measurements

We have been using criterion as a benchmarking library to test the implementation. Benchmarking included both measuring of the merge step with two previously parsed sub-trees as well as measuring the time to parse input of increasing sizes. Most measuring was done on a computer running an Intel i7 processor clocked at 3.40 GHz, with a sample size of 1000.

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. 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.

3.2.1 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, se 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 to be $O(log^3 n)$. Implementation of the merging step was relatively straight-forward, and should therefore satisfy these conditions.

3.2.2 Merging matrices

The core of the parser is the merging of matrices corresponding to previously parsed sub-trees. This should be fast, and not depend on the input size at all. It turns out that works pretty good, as far as measuring goes.

3.2.3 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, which corresponds well to the expected behaviour.

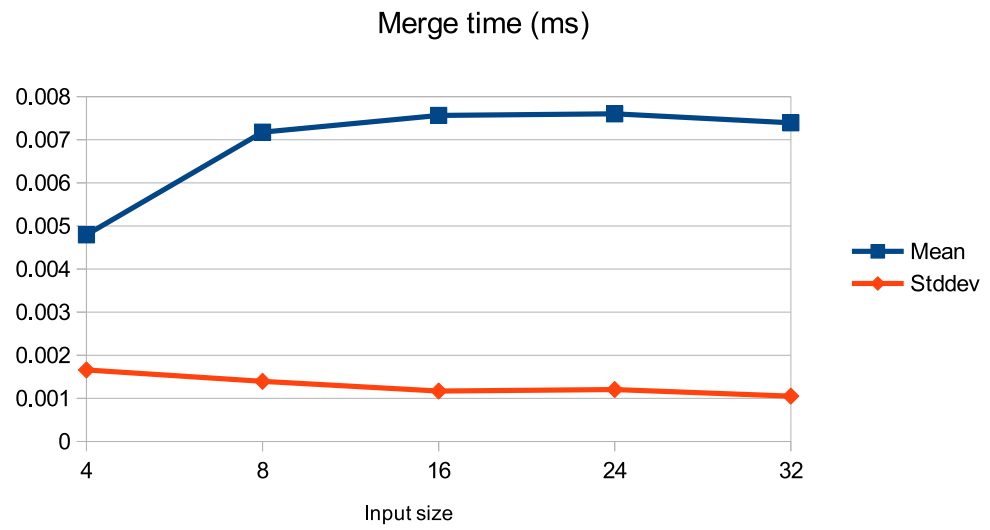


Figure 3.1: Merge times for files where the input size denote the number of functions (of equal size) in that file

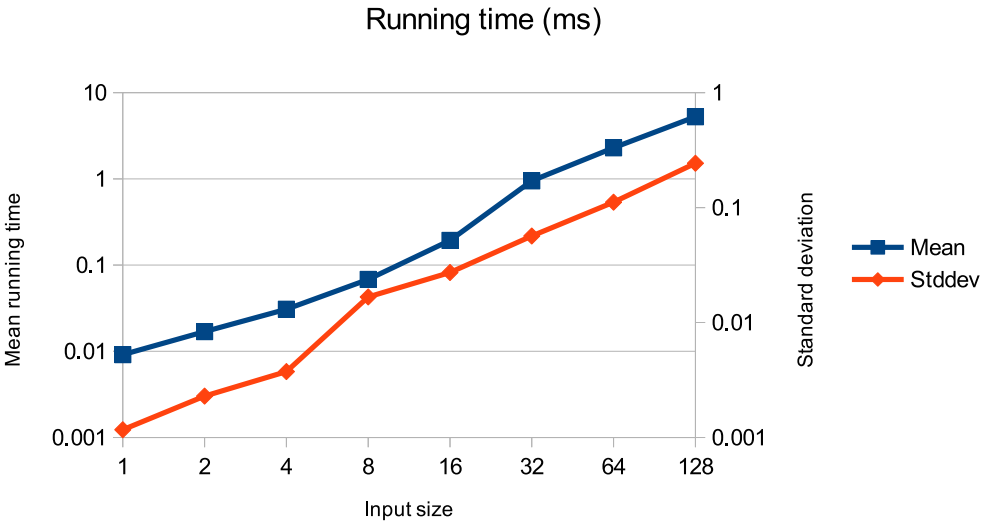


Figure 3.2: Running time for files where the input size denote the number of functions (of equal size) in that file

Chapter 4

Discussion

4.1 Pitfalls

During implementation, a few mistakes that are worth mentioning was made. These are discussed here. There were of course more than these, but most of them were reasonably small, and were not theoretically difficult to solve, but rather straight-forward implementation bugs.

4.1.1 Too many parse results

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 2.3.2). 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.

A later bug, similar to the one described above, was discovered late in the project and has thus not been investigated as much. The results are the same – too many parse results, but they are always the same number, and depend on certain constructs in the input source code. Due to time constraint it has not been possible to pinpoint the source of this bug, but the behaviour suggests an error in `merge`.

4.1.2 Loosen constraint on Matrix type

When writing `merge`, one attempt was made at loosening the constraints on the `Row` and `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 `Quad` (the right way), or by using the less strict versions of `Col` (a wide column of height four) or `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 `Row` or `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.

4.2.1 Position information

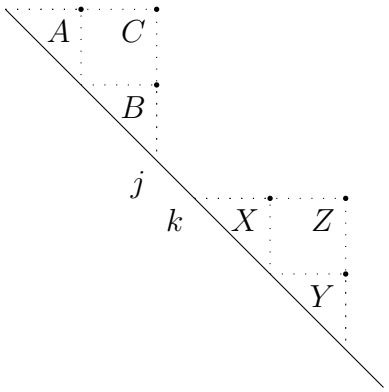
Position information for tokens is a feature that is currently missing in the parser, much due to the fact that it is missing in the lexer. Discussions with Hannson & Hugo revealed that this is due to that not being a priority. The most likely way to implement position information would be by using relative 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

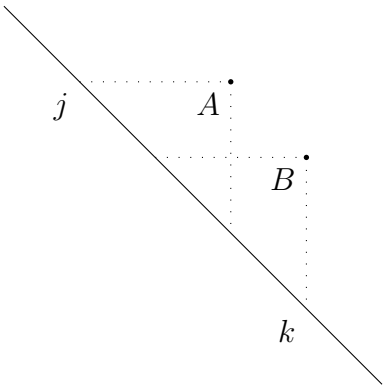
Related to the issue of position information, the error reporting in the lexer and parser is poor to say the least. Invalid tokens are reported by the lexer, but invalid syntax is only reported by saying that there were more than one parse result. For the lexer, the only thing missing in error reporting 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 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 are 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.

Appendix A

Javalette Light

A.1 LBNF grammar

```
1  Prog.      Prog     ::= [Fun];
2  Fun.       Fun      ::= Typ Ident "(" ")"  [Stm] ;
3
4  SDecl.     Stm      ::= Typ Ident ";"  ;
5  SAss.      Stm      ::= Ident "=" Exp ";"  ;
6  SIncr.     Stm      ::= Ident "++" ";"  ;
7  SWhile.    Stm      ::= "while" "(" Exp ")" [Stm]  ;
8
9  ELt.       Exp      ::= Exp1 "<" Exp1 ;
10 EPlus.     Exp1     ::= Exp1 "+" Exp2 ;
11 ETimes.    Exp2     ::= Exp2 "*" Exp3 ;
12 EVar.      Exp3     ::= Ident ;
13 EInt.      Exp3     ::= Integer ;
14 EDouble.   Exp3     ::= Double ;
15
16 _.         Stm      ::= Stm ";" ;
17 coercions Exp 3 ;
18
19 TInt.      Typ  ::= "int" ;
20 TDouble.   Typ  ::= "double" ;
```

A.2 Sample code

```
1  int main() {
2      int p;
3      int x;
4      x = 2;
5      p = 2;
6      while(p < 5) {
7          x = x * x;
8          p++;
9      }
10 }
```

Bibliography

Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: principles, techniques, & tools*. Pearson/Addison Wesley, Boston, 2nd edition, 2007. ISBN 0321491696.

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.

Jean-Philippe Bernardy. Lazy functional incremental parsing. In Stephanie Weirich, editor, *Haskell*, pages 49–60. ACM, 2009. ISBN 978-1-60558-508-6.

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. ISBN 978-1-4503-2326-0.

R. S. Bird. An introduction to the theory of lists. In *Proceedings of the NATO Advanced Study Institute on Logic of Programming and Calculi of Discrete Design*, pages 5–42, New York, NY, USA, 1987. Springer-Verlag New York, Inc. ISBN 0-387-18003-6.

Noam Chomsky. *Syntactic Structures*. Mouton & Co, 1957.

Markus Forsberg et. al. The BNF converter. *http://bnfc.digitalgrammars.com*.

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

Ralf Hinze and Ross Paterson. Finger trees: a simple general-purpose data structure. *J. Funct. Program.*, 16(2):197–217, 2006.

John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to automata theory, languages, and computation - international edition (2. ed)*. Addison-Wesley, 2003. ISBN 978-0-321-21029-6.

Jon M. Kleinberg and Éva Tardos. *Algorithm design*. Addison-Wesley, 2006. ISBN 978-0-321-37291-8.

Martin Lange and Hans Leiß. To CNF or not to CNF? an efficient yet presentable version of the CYK algorithm. *Informatica Didactica*, 8, 2009.

The GHC team. System.IO.Unsafe. *http://hackage.haskell.org/package/base-4.7.0.0/docs/System-IO-Unsafe.html*.

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

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. ISSN 0001-0782.

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