Greg Brown

A Typed, Algebraic Parser Generator

Computer Science Tripos – Part II

Queens' College

May 14, 2021

Declaration

I, Greg Brown of Queens' College, being a candidate for Part II of the Computer Science Tripos, hereby declare that this dissertation and the work described in it are my own work, unaided except as may be specified below, and that the dissertation does not contain material that has already been used to any substantial extent for a comparable purpose.

I am content for my dissertation to be made available to the students and staff of the University.

Signed Greg Brown

Date December 2, 2021

Proforma

Candidate Number:	2374B
Project Title:	**A Typed, Algebraic Parser Generator**
Examination:	**Computer Science Tripos – Part II, 2021**
Word Count:	11062[1]
Line Count:	4758[2]
Project Originator:	The dissertation author
Supervisor:	Prof. Alan Mycroft

Original Aims of the Project

The core aims were to design a language *Nibble* to describe formal languages, and then implement the KY type system in a parser generator *Chomp*, which takes Nibble expressions as input. Unlike traditional parser generators, which view formal languages as automata, Chomp instead takes the approach that languages form an algebraic system. Nibble and parsers generated by Chomp would be compared against the traditional analogues, evaluating whether this new approach to parsing has benefits for language designers, conceptually or computationally.

Work Completed

Exceeded success criterion. One extension was completed which added functions to the Nibble language and the KY type system, reducing repetition in Nibble and redundant computations in Chomp. Two different type systems were developed for the Nibble language, including one with polymorphic types. Chomp can successfully generate parsers from Nibble expressions. These parsers have comparable performance to traditionally-generated parsers, and in some cases outperform handwritten parsers.

Special Difficulties

None.

[1] Calculated using `texcount`. `https://app.uio.no/ifi/texcount/`

[2] Calculated using `scc`, ignoring test inputs. (`https://github.com/boyter/scc`)

Contents

List of Figures

List of Listings

Chapter 1

Introduction

A *formal language* is a set of strings over some alphabet of symbols. For example, a dictionary enumerates a language over written characters. Spoken English is a language, where the alphabet consists of phonemes. Programming languages are over an alphabet of ASCII or Unicode characters. IP packets are languages over bytes. The language of a decision problem is the set of valid inputs.

Whilst linear strings are helpful for data storage and transmission, they have limited use for other algorithms. Strings are used to encode other non-linear data. A *parser* is a function that takes a language string and decodes it to return the underlying data. Parsers should be fast; why spend time decoding data when there is useful computation to be done? Unfortunately, hand writing efficient parsers is repetitive, with lots of boiler plate, and requires careful consideration of the order of operations.

To help solve this problem, *parser generators* were developed – programs that take descriptions of languages and output source code for a parser. Typical efficient parser generators operate by parsing languages with a finite state machine. By storing the state transition table in memory, the run-time cost of a parser is minimal.

As optimising compilers become more powerful, more complex source code can be compiled down to equally-efficient machine code. This allows us to approach parsing from a different direction, without incurring a performance penalty. Instead of parsing formal languages using finite state machines, languages can be interpreted as algebraic expressions. One example algebra is μ-regular expressions, which were described by Leiß [6]. This algebraic approach allows for the creation of intuitive function-based parsing algorithms, as opposed to the opaque parsing tables used in traditional parser generators.

Krishnaswami and Yallop took this approach in a recent paper [5]. One key contribution the pair made was the development of a type system (the KY type system) for μ-regular expressions. If a μ-regular expression was well-typed, then they could produce a linear-time parser for it. They produced a implementation of the KY type system using parser combinators, which are higher order functions that combine zero or more parsers together to make a new parser.

1.1 Project Overview

This project seeks to produce a parser generator based on μ-regular expressions. It makes the following contributions:

- I design the *Nibble* language to describe parsers, based on μ-regular expressions (section 3.1). This language adds functions and let bindings to μ-regular expressions, making it more ergonomic. I determine that Nibble is as easy to use as BNF, meaning future parser generators could use Nibble-like languages as input.
- I describe **two** type systems for the Nibble language, extending the KY type system (section 3.2). Both type systems have simple inference algorithms, eliminating the need for type annotations in the Nibble language.
- I implement the *Chomp* parser generator, which produces Rust source code for parsers from a Nibble description (section 3.3). Chomp uses one of the type systems I described to ensure Chomp-generated parsers run in linear time (section 4.4).
- I demonstrate the Nibble language and Chomp parser generator are suitable for use in complex projects by creating *AutoNibble* (section 4.1). AutoNibble is a Chomp-generated parser for the Nibble language. AutoNibble outperforms a handwritten parser for the Nibble language (section 4.4.2).

Chapter 2

Preparation

I begin this chapter by describing the wider computer science necessary to understand the rest of this dissertation. Next, I discuss the requirements for the Nibble language and the Chomp parser generator, and the software engineering techniques used to achieve them. I then mention the starting point of the project.

2.1 Background

This section starts with a recap on formal languages, from the perspective of formal grammars and finite automata. Next it covers μ-regular expressions and the KY type system, which view languages as algebraic objects. It then skips over to discuss translators, in particular the architecture they typically use. The section concludes with a discussion on the features of Rust used by the implementation of the Chomp parser generator.

2.1.1 Formal Languages

A formal language is a set of strings over some finite alphabet. For example, written English words are a formal language over the English alphabet, spoken sentences are a formal language over phonemes, and programming languages such as Rust are formal languages over Unicode characters.

Most useful formal languages have some structure to them, where every string has a derivation that describes this structure. Parsing is the task of computing a derivation from a string. Consider the following example. Sheep can only say "baa" followed by some number of additional "a"s. The derivations for this sheep language could be the natural numbers. A parser would count the total number of "a"s, and subtract two. Notice how a derivation has no connection to the meaning, or semantics, of a string.

A parser generator is a program that takes a description of a formal language and produces a parser for it. Because all of the strings in a language can be generated from a derivation, and a parser finds a derivation for a given string, a parser generator only needs to receive a description of the form of derivations to be able to generate a parser.

The Chomsky Hierarchy

Traditionally, languages have been specified using formal grammars. We extend the original alphabet with some additional *non-terminal* symbols. One of these is the start symbol, S. To create a string in the language of a grammar, we start with the string

$$S \mapsto baaA$$
$$A \mapsto aA$$
$$A \mapsto \epsilon$$

$$S \mapsto baaA \mapsto baaaA \mapsto baaaaA \mapsto baaaa$$

Figure 2.1: An example formal grammar and a derivation.

consisting of the start symbol. Then, we repeatedly apply string rewriting rules called *production rules* until there are no more non-terminal symbols. Every production rule must consume at least one non-terminal, although they can produce many more.

An example grammar, and the derivation of a string in the grammar, are shown in figure 2.1. This grammar describes the language used by sheep. The start symbol gives us the prefix "baa", and a looping non-terminal A. A either pushes an "a" symbol before it, or removes itself from the string.

Chomsky [2] detailed a classification of formal grammars depending on the form of the production rules: type 0 through type 3. The smaller the number, the less restricted the rules are, and the larger the class of possible languages. Chomsky further discovered that each class can be parsed by a different form of finite automata.

The languages of type 2 grammars are commonly called *context-free languages*. These are the most-restrictive grammars in the hierarchy that have matched delimiters, which are essential for programming languages. These grammars take polynomial time to parse in general. Fortunately, there are some sub-classes of context-free languages that can be parsed in linear time. The most common of these are LL and LR languages, covered in the Part IB Compiler Construction course.

Backus-Naur Form

BNF is a formal language to describe grammars. Its syntax is designed to resemble the production rules of the mathematical definition. Literal symbols are surrounded by quotes. Non-terminal symbols are surrounded by angle brackets. Listing 2.1 shows a BNF description of the sheep language. The `<start>` form corresponds to rules for the S non-terminal. Similarly, the `<loop>` form corresponds to the A non-terminal.

```
<start> ::= "baa" <loop>
<loop>  ::= "" | "a" <loop>
```

Listing 2.1: An example BNF description.

BNF has a single global namespace, such that when a form is declared, it can be used anywhere else in the description. For example, `<loop>` is used before its declaration. BNF uses mutually-recursive scope – different forms can refer to themselves in a cycle.

2.1.2 μ-regular expressions

As an alternative to viewing languages as described by grammars, languages are also algebraic objects. This was the viewpoint considered by Leiß when they described μ-regular expressions [6], described in figure 2.2.

$$e = \bot \mid \epsilon \mid c \mid e \cdot e \mid e \vee e \mid \mu x.e \mid x$$

Figure 2.2: The syntax of μ-regular expressions

$$baa \cdot \mu x.(\epsilon \vee a \cdot x)$$

Figure 2.3: An example μ-regular expression.

There are three constant languages: $\bot$ for the empty language, ϵ for the language of the empty string only, and c for a language containing the single-symbol string c only.

These are combined with two binary operators. Concatenation, $g \cdot g'$ takes strings from g and concatenates them with strings from g'. Alternation, $g \vee g'$, forms the union of the languages g and g'. For brevity, sometimes juxtaposition is used instead of the concatenation operator.

The last expression form is the least-fixed-point operator, $\mu x.g(x)$. This finds the smallest language for x that contains all the strings in $g(x)$.

Figure 2.3 shows an example μ-regular expression, again describing the sheep language. Like the BNF example (listing 2.1), it starts with the constant prefix *baa*. Next it has a fixed point expression. This is the union of the empty string and the symbol a followed by the fixed point expression – a string of zero or more "a" symbols.

Leiß [6] found that μ-regular expressions describe the full set of context-free languages only. This means that for every μ-regular expression, there is a BNF description for the same language.

Unlike BNF, where alternatives are split into many small, reusable rules, μ-regular expressions are always part of one long expression. This has problems for readability, especially for some repetitive real-world languages. See figure 2.4 which gives a μ-regular expression for describing a colour in hexadecimal `#RRGGBBAA` format, where the alpha component is optional. The whole list of hexadecimal digits is listed eight times.

The KY Type System

The KY type system is a type judgement for μ-regular expressions. If an expression is well typed, then there exists a linear-time parser for the language of the expression.

There are three properties of languages that are particularly interesting, named null, first and flast. Their definitions are in figure 2.5. To summarise, a language L is null when it contains the empty string. The first set is the set of symbols starting strings in L, and the flast set is the set of symbols that immediately follow strings in L to make a bigger string in L.

A *KY type* τ is a record $\{\textsc{Null} \in \mathbb{B}, \textsc{First} \subseteq \Sigma, \textsc{Flast} \subseteq \Sigma\}$. As types are triples of values, they can be manipulated by functions. Figure 2.6 shows some basic types and some operations on them. It also describes two constraints on types, used by the typing judgement.

$$\# \cdot (0 \vee \cdots \vee F) \cdot (0 \vee \cdots \vee F) \cdot (0 \vee \cdots \vee F) \cdot (0 \vee \cdots \vee F) \cdot (0 \vee \cdots \vee F) \cdot (0 \vee \cdots \vee F) \cdot (\epsilon \vee (0 \vee \cdots \vee F) \cdot (0 \vee \cdots \vee F))$$

Figure 2.4: μ-regular expressions can contain a lot of repetition. The full list of hexadecimal digits must be listed eight times.

$$\operatorname{null} L \iff \epsilon \in L$$

$$\operatorname{first} L = \{c \in \Sigma \mid \exists w \in \Sigma^*.\, cw \in L\}$$
$$\operatorname{flast} L = \{c \in \Sigma \mid \exists w \in \Sigma^+, w' \in \Sigma^*.\, w \in L \wedge wcw' \in L\}$$

Figure 2.5: Definitions of the null, first and flast properties of languages.

$$b \Rightarrow s = \text{if } b \text{ then } s \text{ else } \emptyset$$

$$\begin{aligned} \tau_\perp &= (\text{false}, \emptyset, \emptyset) \\ \tau_\epsilon &= (\text{true}, \emptyset, \emptyset) \\ \tau_c &= (\text{false}, \{c\}, \emptyset) \end{aligned}$$

$$\tau \vee \tau' = \left\{ \begin{array}{rl} \textsc{Null} = & \tau.\textsc{Null} \vee \tau'.\textsc{Null} \\ \textsc{First} = & \tau.\textsc{First} \cup \tau'.\textsc{First} \\ \textsc{Flast} = & \tau.\textsc{Flast} \cup \tau'.\textsc{Flast} \end{array} \right\}$$

$$\tau \cdot \tau' = \left\{ \begin{array}{rl} \textsc{Null} = & \tau.\textsc{Null} \wedge \tau'.\textsc{Null} \\ \textsc{First} = & \tau.\textsc{First} \cup (\tau.\textsc{Null} \Rightarrow \tau'.\textsc{First}) \\ \textsc{Flast} = & \tau'.\textsc{Flast} \cup (\tau'.\textsc{Null} \Rightarrow \tau'.\textsc{First} \cup \tau.\textsc{Flast}) \end{array} \right\}$$

$$\tau \circledast \tau' = (\tau.\textsc{Flast} \cap \tau'.\textsc{First} = \emptyset) \wedge \neg\tau.\textsc{Null}$$
$$\tau \# \tau' = (\tau.\textsc{First} \cap \tau'.\textsc{First} = \emptyset) \wedge \neg(\tau.\textsc{Null} \wedge \tau'.\textsc{Null})$$

Figure 2.6: The KY types, two binary operations on them, and the two constraints $\circledast$ and $\#$.

$$\frac{}{\Gamma;\Delta \vdash \bot : \tau_\bot}\ \text{KYBot} \qquad \frac{}{\Gamma;\Delta \vdash \epsilon : \tau_\epsilon}\ \text{KYEps}$$

$$\frac{}{\Gamma;\Delta \vdash c : \tau_c}\ \text{KYChar} \qquad \frac{}{\Gamma, x : \tau;\Delta \vdash x : \tau}\ \text{KYVar}$$

$$\frac{\Gamma;\Delta \vdash e : \tau \quad \Gamma;\Delta \vdash e' : \tau' \quad \tau \# \tau'}{\Gamma;\Delta \vdash e \vee e' : \tau \vee \tau'}\ \text{KYAlt} \qquad \frac{\Gamma;\Delta, x : \tau \vdash e : \tau}{\Gamma;\Delta \vdash \mu x.e : \tau}\ \text{KYFix}$$

$$\frac{\Gamma;\Delta \vdash e : \tau \quad \Gamma, \Delta;\cdot \vdash e' : \tau' \quad \tau \circledast \tau'}{\Gamma;\Delta \vdash e \cdot e' : \tau \cdot \tau'}\ \text{KYCat}$$

Figure 2.7: The KY typing judgement.

$$\frac{\tau = S\sigma}{\Gamma, x : \sigma \vdash x : \tau}\ \text{HMVar} \qquad \frac{\Gamma \vdash e : \tau \to \tau' \quad \Gamma \vdash e' : \tau}{\Gamma \vdash ee' : \tau'}\ \text{HMApp}$$

$$\frac{\Gamma, x : \tau \vdash e : \tau'}{\Gamma \vdash \lambda x.e : \tau \to \tau'}\ \text{HMAbs} \qquad \frac{\Gamma \vdash e : \tau \quad \Gamma, x : \forall \alpha.\tau \vdash e' : \tau'}{\Gamma \vdash \text{let}\, x = e\, \text{in}\, e' : \tau'}\ \text{HMLet}$$

Figure 2.8: The Hindley-Milner typing judgement.

The KY type system uses two different variable contexts, so it can distinguish between *guarded* and *unguarded* variables. Guarded variables can only occur on the right of a non-empty string. This is to make sure recursion is deterministic in a naive parser implementation.

Figure 2.7 gives the full typing judgement of the KY type system. Of particular note, the KYFix rule assumes x is guarded in the hypothesis, the KYCat rule shifts the guarded context into the unguarded one for the argument on the right side, and the KYVar rule can only reference unguarded variables. Krishnaswami and Yallop showed [5] that if an expression has a complete typing judgement when the two variable contexts are empty, it is possible to compute a parser for the language of that expression.

2.1.3 The Hindley-Milner Type System

The simply-typed lambda calculus is possibly the simplest type system, consisting of ground terms and functions only. System F extends the lambda calculus by adding *polymorphism*, where values can have multiple types. Unfortunately, type inference, the problem of assigning types to expressions, is undecidable for System F [8].

To overcome this problem, Hindley [4] and later Milner [7] described a different type system with decidable inference. Like System F, it has polymorphic types and type variables. The difference is that only let expressions can have polymorphic types. The typing rules are detailed in figure 2.8.

A key part of the Hindley-Milner type system is *specialisation*. This is the instantiation of one or more free variables in a polymorphic type. If a type σ can be specialised to type σ' by a map S of instantiations, then $\sigma' = S\sigma$. Only the HMVar rule can specialise types.

Conversely, the HMLet rule *generalises* types. First, the bound expression is type checked. Then, all the free type variables are bound by the universal quantification, before the body is type checked with this new expression.

2.1.4 Translators

Translators are programs that transform one formal language into another whilst preserving the semantics. A familiar example are natural language translators, which map sentences from one language into another whilst preserving the meaning. An example from computer science are compilers, which translate source code into machine code, such that when they are both executed the result is the same.

Translators consist of three different phases, named "ends". The front-end parses the source language into a source derivation. The middle-end transforms the source derivation into a target derivation, preserving the semantics. The back-end generates the target language from the target derivation.

We will consider two examples: compilers and parser generators. The front-end of a compiler parses the source code into an abstract syntax tree. Functional languages typically introduce De Bruijn indices at this stage, which are introduced later in this section. The middle-end has two roles. The first is to type check the abstract syntax tree. If this fails, then the compiler cannot guarantee that language semantics are preserved, so execution stops. Otherwise, the middle-end produces intermediate code – the derivation for machine code. Using this intermediate representation, the back-end produces machine code. In some cases, the back-end is itself a translator.

Parser generators are another example of translators. The front-end of a parser generator receives a string describing a formal language. This is parsed into an abstract syntax tree for the input language. The middle-end then attempts to produce an abstract description for the parsing algorithm. Some parser generators produce action tables you would use to describe Turing machines. Others create decision trees. In any case, if the generator cannot produce an algorithm that matches the described formal language then the translator produces an error. The back-end of parser generators use the abstract algorithm description to generate source code for the target programming language.

Often, the most complex part of a translator is the middle-end. By creating a strong separation between the three phases, the front-end and back-ends can be easily modified or replaced to accept different source languages or output different target languages respectively. For example, the back-end of many compilers can produce machine code for different instruction sets and platforms. Clang and GCC, two popular C compilers, can both act as front-ends to the GCC compiler.

De Bruijn Indices

Most functional programming languages have a property called α-renaming. To summarise, given any program, if you rename all occurrences of any variable then the semantics of the program do not change. De Bruijn indices [3] exploit the lexical scoping features of functional programming languages to provide all variables with a name that is invariant under α-renaming. This can simplify many algorithms, such as substitution.

Consider the OCaml expression `(fun x y -> x y) y` . If we try to naively evaluate the expression, by substituting `y` for `x`, we get the term `(fun y -> y y)`. However, this has different semantics than the original expression. A correct substitution would be `(fun z -> y z)`. Notice how we had to rename the bound variable.

Using De Bruijn indices, there is no longer a need to rename variables. Observe how variables form a stack – first, `y` is declared at some point earlier in the program. Within the anonymous function, `x` and `y` are bound to new variables, and outside of the function

those bindings are popped off. De Bruijn indices represent variables by their position from the top of the variable stack.

The OCaml expression from earlier becomes `(fun . . -> 0 1) 0` when using De Bruijn indices, assuming that `y` was the last variable declared before this expression. Notice how the anonymous function does not need to name its parameters – the De Bruijn indices uniquely identify every variable.

After substitution, the function becomes `(fun . -> 1 0)`. First, the variable `x` eliminated, hence removed from the variable stack. That means that the `y` inside the anonymous function was promoted to the top of the stack, so its index decreased – the `1` became a `0`. Outside the function, the variable `y` was at the top of the stack, so it had index `0`. As it moved into the function, another variable was pushed onto the stack above it, so its index was changed to `1`.

By using De Bruijn indices, the originally difficult problem of renaming variables during substitution has become a simple transformation of incrementing and decrementing some integers.

2.1.5 Rust

Rust is an imperative programming language designed to eliminate a number of memory safety issues encountered in C and C++. This is achieved through its ownership system, although that is not exploited by this project. This project instead makes use of the procedural-macro system.

A procedural macro is a program that receives a stream of Rust tokens and outputs a different stream of Rust tokens. There are two primary uses for procedural macros: to extend existing Rust code; and to add new syntax to Rust.

The primary use of procedural macros is to extend existing Rust code. These add additional definitions to Rust data types. For example, if a data type is annotated with the `#[derive(Debug)]` attribute, then the Rust compiler can use a procedural macro to generate code used to print debugging information for the data type.

This project is primarily concerned with the other use of procedural macros – to add new syntax to Rust. If a formal language only uses Rust tokens, then strings from that language can be used as arguments to a procedural macro. The procedural macro can then parse and process this language however it sees fit, before outputting arbitrary Rust source code derived from the string.

One example used in this project is `quote`. This procedural macro transforms Rust source code into an abstract syntax tree representing that code. This is used in the back-end of the Chomp parser generator (section 3.6) to produce parser code.

2.2 Requirements Analysis

The primary goal of the project was to provide a parser generator for the KY type system. Achieving this goal requires three parts. First, the Nibble language has to provide a syntax for μ-regular expressions. Second, the Chomp parser generator has to implement the KY type system. Third, the Chomp parser generator has to output source code for a parser.

After completing this primary deliverable, the Nibble language and Chomp parser generators could be extended with new features, one after another. This lends itself to the

spiral development model [1], where each new feature undergoes a complete waterfall development cycle – design, implement, integrate, test.

It is useful to be able to concurrently work on many features at once during the design phase, to be able to gauge the difficulty in completing a full implementation and to see the ways in which different features can conflict with each other. This is only possible with strict version control measures, so that each feature remains separate and so that a functional deliverable is always available.

To solve these problems, I used `git` for version control. Development of the core deliverable took place on the `master` branch. Once it was complete, all new features were developed on different branches. Several branches were added for the exploratory design of each potential feature. Once I decided on a particular feature to implement, I proceeded to complete its waterfall cycle on its design branch. Then, the `master` branch was rebased onto the feature branch.

Many additional features were considered for inclusion in this project as stretch requirements. I performed a MoSCoW analysis for each potential feature, shown in table **??**. This ranked features by the impact on the design of the Nibble language and the complexity of implementation in the Chomp parser generator.

#+caption: A MoSCoW analysis of features to include in the project. [̂htbp]

Priority	Feature
Must Have	Embed μ-regular expressions; Implement KY type system; Generate Rust parser
Should Have	Let statements; Function expressions; Type inference;
Could Have	User-defined parser errors; Semantic actions
Won't Have	Lexer

One important stage of each waterfall development cycle was testing. Most tests for the Chomp parser generator were end-to-end tests. By using the visitor pattern, described in section 3.5.1, the implementation of an operation is broken down into a different operation for each syntax node in the Nibble language. Each of these sub-operations were small enough to verify by inspection. Therefore, only large inputs needed testing, and the easiest way to provide large test inputs is with end-to-end tests.

A key part of the test suite was AutoNibble, the Chomp-generated parser for the Nibble language. If the outputs of AutoNibble and the handwritten parser for Nibble used by Chomp's front-end were equal, then it was highly likely that Chomp worked correctly.

When a new feature was added to the project, it could introduce regressions in the behaviour of existing end-to-end tests. Once the source of the problem was identified, I added a regression unit test to exercise the issue, to save time if a future extension reintroduced the problem.

This project was intended to be a proof-of-concept for parser generators based on the KY type system. Therefore, the code was made publicly available on both my personal website and on GitHub. It is dual-licensed under the MIT and Apache 2.0 licenses, like many Rust projects, such that other people can reuse code as part of any project or in any form, as long as they include the licenses.

2.3 Starting Point

I closely studied the KY type system before beginning the project.

I had previous experience with using the Rust language for personal projects.

The project builds on ideas about formal languages. These have been studied in the *Part IA Discrete Maths* and *Part IB Compiler Construction* courses. I also completed a small personal project on them previously.

Additionally, the project uses concepts from type systems, covered in the *Part IB Semantics of Programming Languages*, *Part II Types* and *Part II Denotational Semantics* courses.

Chapter 3

Implementation

There are two areas of implementation for this project. The first is the design of the Nibble language, which describes formal languages, along with the description of the theoretical implementation of two type systems for Nibble (sections 3.1 and 3.2). The second explores the implementation of the Chomp parser generator, which transforms a Nibble expression describing a formal language into Rust source code for a parser of that language (sections 3.3 through 3.6).

Section 3.1 starts by introducing the syntax and semantics of the Nibble language, explaining how it solves the repetition problem of μ-regular expressions. Then in section 3.2, the design of two type systems for Nibble are described: KY-λ in section 3.2.1 and LM in section 3.2.2.

Next this chapter moves on to describing the Chomp parser generator, starting with the structure of its code repository and overall architecture in section 3.3. Chomp has an architecture similar to other compilers and translators (section 2.1.4). The front-, middle- and back-ends of Chomp are described in sections 3.4, 3.5, and 3.6 respectively.

3.1 The Nibble Language

Nibble is a formal language for describing formal languages – semantically, a Nibble expression represents a formal language. Nibble is designed to be a user-friendly alternative to μ-regular expressions. In section 2.1.2, the problem of repetition in μ-regular expressions was introduced. Nibble solves this problem by introducing let statements and lambda abstractions. The syntax of the Nibble language is given using BNF in listing 3.1.

For Nibble expressions to be a suitable replacement for μ-regular expressions, Nibble must be able to describe the same set of languages. Nibble achieves this by directly embedding μ-regular expressions. Listing 3.2 shows how Nibble embeds the μ-regular expression from figure 2.3, describing the sheep language.

You may notice that Nibble does not embed the $\perp$ μ-regular expression. By itself, $\perp$ has no practical use – there is no need to parse the empty language. When combined with other combinators, $\perp$ is either an annihilator or the identity, demonstrated in figure 3.1. This means any μ-regular expression containing $\perp$ is either semantically equivalent to $\perp$, or semantically equivalent to an μ-regular expression without $\perp$.

```
<expression> ::= <let-stmt> <expression> | <match-stmt>
<let-stmt>   ::= "let" <ident> <arg-list> "=" <expr> ";"
<match-stmt> ::= "match" <expr> ";"

<expr>     ::= <lambda> | <alt>
<lambda>   ::= "/" <arg-list> "/" <alt>
<alt>      ::= <named> | <named> "|" <alt>
<named>    ::= <cat>   | <cat>   ":" <ident>
<cat>      ::= <call>  | <call>  "." <cat>
<call>     ::= <term>  | <term>      <call>
<term>     ::= <epsilon>
             | <literal>
             | <ident>
             | <fix>
             | "(" <expr> ")"
<fix>      ::= "!" <term>
<epsilon>  ::= "_"

<arg-list> ::= "" | <ident> <arg-list>
<ident>    ::= <letter> | <letter> <ident>
```

Listing 3.1: The syntax of the Nibble language.

```
match "baa".!(/x/ (_|"a" . x));
```

Listing 3.2: Nibble expressions can embed μ-regular expressions.

$$\begin{aligned} \bot \cdot e &= \bot \\ e \cdot \bot &= \bot \\ \bot \vee e &= e \\ e \vee \bot &= e \\ \mu x.\bot &= \bot \end{aligned}$$

Figure 3.1: Equalities to eliminate $\bot$ from μ-regular expressions.

The Nibble language fixes the repetition problem of μ-regular expressions, by introducing let statements. Listing 3.3 demonstrates how Nibble can eliminate the simple repetition from figure 2.4.

```
let hex = "0"|"1"|"2"|"3"|"4"|"5"|"6"|"7"|"8"|"9"
        | "a"|"b"|"c"|"d"|"e"|"f"
        | "A"|"B"|"C"|"D"|"E"|"F";
match "#" . hex . hex . hex . hex . hex . hex . (_ | hex . hex);
```

Listing 3.3: Nibble expressions can introduce variables with let statements.

A let statement introduces a new variable name, the binding variable, and assigns it a Nibble expression, the bound expression. In this case, the variable `hex` is assigned to a hexadecimal character. The variable can be used repeatedly in following statements.

Whilst let statements can eliminate verbatim repetition, they do not help with repetitive patterns, where there are only minor differences between different instances of a pattern. The Nibble language handles this with lambda abstractions, which are demonstrated in 3.4.

```
let opt = /x/ _ | x;
let plus x = !(/plus/ x . opt plus);
match plus "a" . plus "b" . plus "c";
```

Listing 3.4: Lambda and application expressions add functions to the Nibble language.

There are two ways to introduce a lambda abstraction: either through a lambda expression `/x/ e`, or through a let-lambda statement `let x y = e`. The let-lambda statements are *syntactic sugar* for a let-statement binding a lambda-expression. They are indistinguishable in their semantics and how they are type checked: the Nibble expression `let x y = e` is equivalent to `let x = /y/ e`.

Notice how in the second line of the listing, the fixed-point operator `!` takes the lambda expression as an argument. In general, the fixed-point operator accepts any expression that is a one-argument first-order function. Whilst not demonstrated here, functions can take more than one argument.

3.2 Type Systems for Nibble

I have designed two type systems for Nibble: KY-λ and LM. The KY-λ type system is a minimal departure from the KY type system for μ-regular expressions, which was presented in section 2.1.2, by treating lambda abstractions as μ-regular expression macros. The LM type system is a theoretical system for Nibble, incorporating ideas from the Hindley-Milner type system, introduced in section 2.1.3. This adds polymorphic function types on top of the KY type system.

3.2.1 The KY-λ Type System

The KY-λ type system is a type system for Nibble using the KY type system, presented in section 2.1.2, as the core. By treating lambda abstractions as macros for μ-regular expressions, Nibble expressions are translated into embedded μ-regular expressions. This μ-regular expression is type checked using the KY type system.

$$\frac{}{[\![\texttt{\_}]\!] = \texttt{\_}} \qquad \frac{}{[\![\texttt{"w''}]\!] = \texttt{"w''}}$$

$$\frac{}{[\![\texttt{x}]\!] = \texttt{x}}$$

$$\frac{[\![\texttt{e}]\!] = \texttt{f} \quad [\![\texttt{e'}]\!] = \texttt{f'}}{[\![\texttt{e . e'}]\!] = \texttt{f . f'}} \qquad \frac{[\![\texttt{e}]\!] = \texttt{f} \quad [\![\texttt{e'}]\!] = \texttt{f'}}{[\![\texttt{e | e'}]\!] = \texttt{f | f'}}$$

$$\frac{}{[\![\texttt{/x/ e}]\!] = \texttt{/x/ e}} \qquad \frac{[\![\texttt{e}]\!] = \texttt{/x/ f} \quad [\![\texttt{f [e'/x]}]\!] = \texttt{f'}}{[\![\texttt{e e'}]\!] = \texttt{f'}}$$

$$\frac{[\![\texttt{e}]\!] = \texttt{/x/ f} \quad [\![\texttt{f}]\!] = \texttt{f'}}{[\![\texttt{!e}]\!] = \texttt{!(/x/ f')}} \qquad \frac{[\![\texttt{e' [e/x]}]\!] = \texttt{r}}{[\![\texttt{let x = e; e'}]\!] = \texttt{r}}$$

Figure 3.2: Translation of Nibble expressions

For a Nibble expression `e`, the translation of `e` is denoted as $[\![\texttt{e}]\!]$. This is detailed in figure 3.2 To summarise, translation performs call-by-name evaluation of Nibble expressions. The only exception is the fixed-point operator, `!e`. This first translates the argument `e`. If `e` translates to `/x/ f`, then `f` is translated, keeping `x` free. Otherwise, translation fails.

There are some problems with this approach. Firstly, call-by-name evaluation of untyped terms is non-terminating. Consider listing 3.5. The expression `omega omega` evaluates to `omega omega`. Users might be confused by the parser generator hanging instead of producing an error. In any case, the type system ensures parsers terminate in linear time, which is more important in most cases.

```
let omega x = x x;
match omega omega;
```

Listing 3.5: An example Nibble expression that does not translate. Evaluating `omega omega` produces `omega omega`.

Another issue is that unused expressions are completely ignored. Whilst this has some computational benefits, it could cause confusion for users. An example is in listing 3.6. Due to the evaluations strategy, because the ill-typed expression `"a" | "a"` (it fails the # constraint) is bound to the unused variable `unused`, the overall expression is well-typed. If someone referenced `unused`, type checking would unexpectedly fail.

```
let unused = "a" | "a";
match "baa";
```

Listing 3.6: The KY-λ type system ignores expressions bound by unused variables.

3.2.2 The LM Type System

The LM type system is an alternative type system for Nibble. It combines features of the Hindley-Milner type system discussed in section 2.1.3 with the core of the KY type system, presented in section 2.1.2. This allows for expressions bound by let statements to have polymorphic types, and removes the need to translate expressions before type checking.

$$\frac{}{\Gamma;\Delta \vdash \texttt{\_} : K(\tau_\epsilon);\emptyset}\ \text{LMEps} \qquad \frac{}{\Gamma;\Delta \vdash \texttt{"cw"} : K(\tau_c);\emptyset}\ \text{LMLit}$$

$$\frac{\sigma = S\rho \qquad C' = SC}{\Gamma, x : (\rho, C);\Delta \vdash \texttt{x} : \sigma; C'}\ \text{LMVar}$$

$$\frac{\Gamma;\Delta \vdash \texttt{e} : K(\tau);C \qquad \Gamma,\Delta;\cdot \vdash \texttt{e'} : K(\tau');C'}{\Gamma;\Delta \vdash \texttt{e . e'} : K(\tau\cdot\tau');C \cup C' \cup \{\tau \circledast \tau'\}}\ \text{LMCat}$$

$$\frac{\Gamma;\Delta \vdash \texttt{e} : K(\tau);C \qquad \Gamma;\Delta \vdash \texttt{e'} : K(\tau');C'}{\Gamma;\Delta \vdash \texttt{e | e'} : K(\tau\vee\tau');C \cup C' \cup \{\tau\#\tau'\}}\ \text{LMAlt}$$

$$\frac{\Gamma, x : (\sigma,\emptyset);\Delta \vdash \texttt{e} : \sigma';C}{\Gamma;\Delta \vdash \texttt{/x/ e} : \sigma \to \sigma';C}\ \text{LMAbs1} \qquad \frac{\Gamma;\Delta, x : (\sigma,\emptyset) \vdash \texttt{e} : \sigma';C}{\Gamma;\Delta \vdash \texttt{/x/ e} : \sigma \rightsquigarrow \sigma';C}\ \text{LMAbs2}$$

$$\frac{\Gamma;\Delta \vdash \texttt{e} : \sigma \to \sigma';C \qquad \Gamma;\Delta \vdash \texttt{e'} : \sigma;C'}{\Gamma;\Delta \vdash \texttt{e e'} : \sigma';C \cup C'}\ \text{LMApp1}$$

$$\frac{\Gamma;\Delta \vdash \texttt{e} : \sigma \rightsquigarrow \sigma';C \qquad \Gamma,\Delta;\cdot \vdash \texttt{e'} : \sigma;C'}{\Gamma;\Delta \vdash \texttt{e e'} : \sigma';C \cup C'}\ \text{LMApp2}$$

$$\frac{\Gamma;\Delta \vdash \texttt{e} : K(\tau) \rightsquigarrow K(\tau);C}{\Gamma;\Delta \vdash \texttt{!e} : K(\tau);C}\ \text{LMFix} \qquad \frac{\Gamma;\Delta \vdash \texttt{e} : \sigma \rightsquigarrow \sigma';C}{\Gamma;\Delta \vdash \texttt{e} : \sigma \to \sigma';C}\ \text{LMSub}$$

$$\frac{\Gamma;\Delta \vdash \texttt{e} : \sigma;C \qquad \Gamma;\Delta, x : (\bar\sigma,\bar C) \vdash \texttt{e'} : \sigma';C'}{\Gamma;\Delta \vdash \texttt{let x = e; e'} : \sigma';C'}\ \text{LMLet}$$

Figure 3.3: The LM type rules

Figure 3.3 shows the typing rules for the LM type system. We first talk through the various rules, then show some example inferences. Looking at those examples may help with understanding these rules.

The variable contexts store both types and constraints. Constraints are relations between types in the KY type system. They need to be stored in the variable context because these relations are not always decidable for type variables. For example, whether $\alpha\#\tau_c$ holds depends on what α is.

Another oddity in the LM typing system is the form of the judgement. We extend the KY typing judgement by adding a set of constraints C to the conclusion. The judgement has the form $\Gamma;\Delta \vdash \texttt{e} : \tau;C$, which can be read: under unguarded variable context Γ and guarded context Δ, the Nibble expression `e` has the type τ given the constraints in C hold. Thus, type checking an expression becomes a two-step process: infer a type; then check the constraints hold.

The LMEps and LMLit rules are the simplest LM typing rules, being largely unchanged from the KY type system. The only differences are that the type is wrapped in a K constructor, to distinguish KY types from function types, and they both return an empty set of constraints.

The LMVar rule is a combination of the variable rules from the Hindley-Milner and KY type systems. First, the variable must be unguarded. This is to prevent infinite recursion, like in the KY type system. Second, the output type and constraints are a specialisation of the type and constraints from the context. This is like the Hindley-Milner variable typing rule. We specialise the type constraints because they can also include type variables.

LMCat and LMAlt remain similar to the corresponding rules in the KY type system. Like the LMEps and LMLit rules, all the types are wrapped in a K constructor. Instead of the constraints appearing in the premise, as they do in the KY type system, they are moved to the conclusion. This is so they can be checked later when all type variables are instantiated.

Notice how there are two different typing rules for lambda expressions. This is due to the two variable contexts from the KY type system. One function type, used by LMAbs1, is for unguarded functions, where the formal parameter can be used in an unguarded context. The other function type is used by LMAbs2 for guarded functions, where the formal parameter can only be used in guarded contexts. Because lambda expressions are monomorphic, type constraints pass straight through.

Again due to the presence of two function types, there are two typing rules for application. If the called function is an unguarded function type, then the argument is evaluated in the same context, described by the LMApp1 rule. If the function is a guarded function, the LMApp2 rule applies and the argument is evaluated in an unguarded context – the function body ensures the parameter only appears on the right side of a concatenation, so all variables are accessible.

The LMSub rule allows guarded functions to be used when an unguarded function was expected. Krishnaswami and Yallop [5] proved the transfer property that makes this safe. This rule is likely only useful for type inference.

The LMFix rule only accepts first-order guarded functions as the argument. Whilst fixed-points could accept higher-order functions, doing so would allow Nibble expressions that do not translate to μ-regular expressions. To prevent unguarded recursion in parsers, the formal parameter must be guarded.

The LMLet rule is taken from the corresponding Hindley-Milner typing rule almost directly. Note that bound variables are always unguarded. This is also the only typing rule that adds constraints to variables in the context. A consequence of this typing rule is that constraints C on the bound expression are only checked if `x` is used in the body.

Examples

I will now justify the need for two different function contexts. Consider the expression `/x/ _ | x`, corresponding to an optional `x`. This expression is an essential combinator for real-world languages. `x` appears unguarded in the expression `_ | x`, so this lambda expression can have the type $\forall\alpha.\tau_\epsilon \vee \alpha$.

Now consider the expression `!(/x/ "a" | "<".x.">")`. As a μ-regular expression, this would be represented as $\mu x.a \vee <x>$. Recalling the rules from the KY typing judgement,

x would be introduced to the guarded context. Hence, `x` should be introduced to the guarded context too. Therefore, this expression has the type $\mu\alpha.\tau_a \vee \tau_< \cdot \alpha \cdot \tau_>$.

3.3 Chomp Repository Overview

Chomp is a parser generator from the Nibble language to Rust. Chomp is implemented in Rust. Table **??** gives a brief description of the repository structure for Chomp.

#+caption: Brief outline of the code repository structure. All code is

[ĥtbp]

Directory	Description	Lines of Code
`src/nibble`	Nibble parser and normalisation	738
`src/chomp`	Chomp type inference algorithm	1786
`src/lower`	Parser code generation	403
`tests`	Minimal end-to-end tests of Chomp	57
`chewed`	Shared library for all Chomp-generated parsers	270
`chomp-macro`	Procedural macro interface	42
`autonibble/src`	AutoNibble implementation	489
`autonibble/tests`	Tests for correctness of AutoNibble	35
`autonibble/benches`	Benchmarks of AutoNibble	57
`chomp-bench/**/json`	Benchmarks of various parsers for JSON	489
`chomp-bench/**/arith`	Benchmarks of various parsers for arithmetic	274

Overall, the project consists of a main Chomp generator and four smaller supporting libraries. The top-level directory consists of the source code and test directory for the main Chomp generator. The `chewed`, `chomp-macro`, `autonibble` and `chomp-bench` directories contain code for the supporting libraries.

The `src` directory contains the code for the main Chomp generator. It is split into three parts. The front-end is in the `src/nibble` directory. This parses an input stream of Nibble expressions and produces an abstract syntax tree for it. The middle-end is in the `src/chomp` directory. This performs type inference using the KY-λ type system. It outputs a typed μ-regular expression. The back-end is in the `src/lower` directory. It is responsible for converting the typed μ-regular expression into Rust source code for a parser.

The `tests` directory contains the code to run a number of end-to-end tests for the Chomp generator, making sure certain examples type check correctly. Correctness of the generated parsers is left to benchmarking and other test code.

All Chomp-generated parsers share some common code, such as the trait definition (a type definition like a Java interface) for a parser. This common code is kept in the `chewed` library. The reason it forms a separate library is so that consumers of Chomp only need to include the small `chewed` library with their final binary, instead of the relatively large Chomp library.

To help make Chomp easier for developers to include in their projects, a procedural macro interface was created. Due to current limitations in Rust, this interface has its own, small library in the `chomp-macro` directory.

The success criterion for this project required bootstrapping the Nibble language – using a Chomp-generated parser to parse Nibble expressions. This parser, dubbed *AutoNibble*,

is in the `autonibble` directory. This directory includes some simple tests of the correctness of AutoNibble, as well as some benchmarks to compare its performance against the handwritten parser used by Chomp's front-end.

Finally, there is the `chomp-bench` directory. This is a small library for comparing the performance of Chomp-generated parsers against handwritten parsers and `lalrpop`-generated parsers, which is an external parser generator for Rust. Ideally, this library would be part of `chomp-macro`. However, limitations in the build system for `lalrpop` makes this impossible.

3.4 The Front-End: Parsing and Normalisation

The front-end of Chomp is responsible for converting the input stream of characters representing a Nibble expression into an abstract syntax tree. This occurs in three stages: lexing, which splits the characters in the input stream into different tokens; parsing, which transforms the token stream into a concrete syntax tree; and normalisation, which converts this concrete syntax tree into an abstract syntax tree.

In Chomp, the lexing is performed by the external `syn` library. This can convert streams of characters into tokens from the Rust language. Nibble uses a subset of the tokens found in Rust, so lexing into Rust tokens makes the parser simpler. It also makes integration with procedural macros significantly easier, because procedural macros receive a stream of Rust tokens as input.

The parser in Chomp also uses the `syn` library. It provides lightweight interface to parse some often-used data structures. For example, it provides the `Punctuated<T, P>` type, which represents a list of values of type `T`, separated by values of type `P`.

Listing 3.7 shows how Chomp parses a Nibble term. This function makes use of Rust's type inference and trait systems to call one of four different functions, all written as `input.parse()`. By checking what the next input token is, it is possible to determine exactly what type of term can appear.

The front-end finishes with normalisation, which converts the concrete syntax tree into an abstract syntax tree. This occurs in two stages that occur simultaneously. First, syntactic sugar is expanded. Second, variable names are converted to De Bruijn indices (introduced in section 2.1.4).

In section 3.1, let-lambda statements were introduced as syntactic sugar for a let statement binding a lambda expression. During normalisation, let-lambda statements are converted into this expanded form, instead of keeping them around as another node type in the abstract syntax tree. This reduces complexity in later stages of Chomp.

The other part of normalisation is conversion to De Bruijn indices. Most of this work is achieved by the `Context` struct, shown in listing 3.8. As discussed in section 2.1.4, when De Bruijn indices were introduced, variables in lexically-scoped languages form a stack. The most-recently declared variable is at the top of the stack. New variables are pushed on top of the stack, and popped off when they leave scope.

There are two ways to introduce variables in Nibble. The binding variables from let statements are in scope for the rest of the Nibble expression. The formal parameters of lambda expressions are in scope only for the body of the lambda expression. This is reflected by the two different ways to push variables to the stack.

```rust
pub enum Term {
    Epsilon(Epsilon),
    Ident(Ident),
    Literal(Literal),
    Fix(Fix),
    Parens(ParenExpression),
}

impl Parse for Term {
    fn parse(input: ParseStream<'_>) -> syn::Result<Self> {
        let lookahead = input.lookahead1();

        if lookahead.peek(Token![_]) {
            input.parse().map(Self::Epsilon)
        } else if lookahead.peek(LitStr) {
            input.parse().map(Self::Literal)
        } else if lookahead.peek(Token![!]) {
            input.parse().map(Self::Fix)
        } else if lookahead.peek(Paren) {
            input.parse().map(Self::Parens)
        } else if lookahead.peek(Ident::peek_any) {
            input.call(Ident::parse_any).map(Self::Ident)
        } else {
            Err(lookahead.error())
        }
    }
}
```

Listing 3.7: Rust code snippet that parses a Nibble term. An enum is like an OCaml datatype.

```rust
pub struct Context {
    bindings: Vec<Name>,
}

impl Context {
    /// Get the De Bruijn index of `name`, if it is defined.
    pub fn lookup(&self, name: &Name) -> Option<usize> {
        self.bindings
            .iter()
            .rev()
            .enumerate()
            .find(|(_, n)| *n == name)
            .map(|(idx, _)| idx)
    }

    /// Permanently add the variable `name` to the top of the stack.
    pub fn push_variable(&mut self, name: Name) {
        self.bindings.push(name);
    }

    /// Call `f` with the variable `name` pushed to the top of the stack.
    pub fn with_variable<F: FnOnce(&mut Self) -> R, R>(
        &mut self,
        name: Name,
        f: F,
    ) -> R {
        self.bindings.push(name);
        let res = f(self);
        self.bindings.pop();
        res
    }
}
```

Listing 3.8: Rust code snippet for conversion to De Bruijn indices.

`push_variable` adds a variable onto the stack of `bindings` in a `Context` permanently. The API provides no way to remove variables. This is called by let statements, after converting the bound expression but before converting their body. The `with_variable` method pushes a variable onto the stack only for the duration of a call to the function `f`. This is used by lambda expressions, where `f` will convert the lambda body.

The `lookup` method does all of the heavy-lifting. It numbers each member of the stack starting from the top. Then still from the top, it returns the index of the first item with a matching name. If no such item exists, then an error is returned.

3.5 The Middle-End: Type Inference

The middle-end of Chomp performs type inference using the KY-λ type system, to produce a typed μ-regular expression. First, the abstract syntax tree computed by the front-end is translated. Next, its type is inferred using the KY type system, and the output expression is built. Before going into detail over how these two stages work, the visitor design pattern is described, as it is used to implement both of them.

3.5.1 The Visitor Design Pattern

Both translation and the type inference algorithm use the visitor design pattern. This design pattern separates algorithms from the data structures they operate on. The visitor design pattern follows the open/closed principle – the data structure is closed for modification, but the design pattern makes it open for new algorithms.

Figure 3.4 shows a UML diagram for the visitor design pattern. The `Visitor` interface requires implementors to handle each type of object in the `Visitable` data type. The `Visitable` data type then only needs one generic dispatch method to implement all the algorithms that use this pattern.

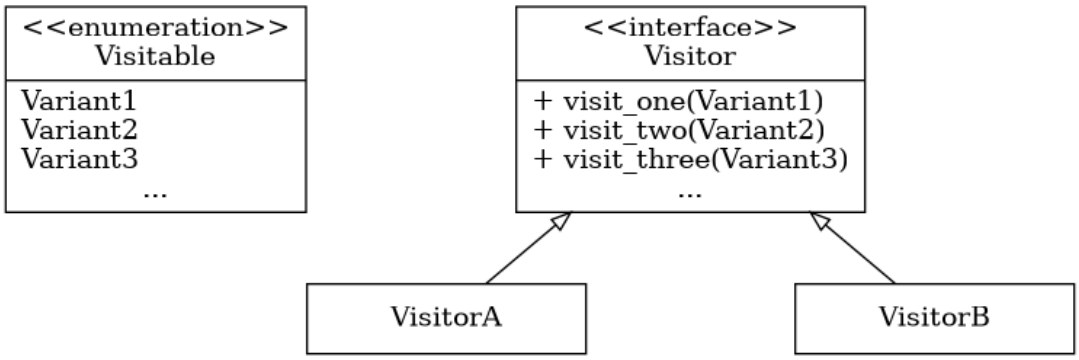

Figure 3.4: A UML diagram depicting the visitor design pattern.

Listing 3.9 show how this pattern is implemented in Chomp. The `Visitor` trait includes a function signature for each type of abstract-syntax-tree node. `NamedExpression` then uses pattern matching to dispatch the appropriate method.

An alternative to the visitor pattern in Rust is to define algorithms as traits directly on `NamedExpression`. However, this would require making the representation of `NamedExpression` public, so that it can be fully unwrapped. Visitors would also be more tightly coupled to the implementation of `NamedExpression`. Changing the representation of `NamedExpression` would then be more difficult in future.

```rust
pub type NameSpan<'a> = (Option<&'a Name>, Option<Span>);
pub trait Visitor {
    type Out;
    fn visit_epsilon(&mut self, namespan: NameSpan, eps: &Epsilon) -> Self::Out;
    fn visit_literal(&mut self, namespan: NameSpan, lit: &Literal) -> Self::Out;
    fn visit_cat(&mut self, namespan: NameSpan, cat: &Cat) -> Self::Out;
    fn visit_alt(&mut self, namespan: NameSpan, alt: &Alt) -> Self::Out;
    fn visit_fix(&mut self, namespan: NameSpan, fix: &Fix) -> Self::Out;
    fn visit_var(&mut self, namespan: NameSpan, var: &Variable) -> Self::Out;
    fn visit_call(&mut self, namespan: NameSpan, call: &Call) -> Self::Out;
    fn visit_lambda(&mut self, namespan: NameSpan, lambda: &Lambda) -> Self::Out;
    fn visit_let(&mut self, namespan: NameSpan, stmt: &Let) -> Self::Out;
}

impl NamedExpression {
    pub fn visit<V : Visitor>(&self, visitor: &mut V) -> <V as Visitor>::Out {
        let namespan = (self.name.as_ref(), self.span);
        match &self.expr {
            Self::Epsilon(e) => visitor.visit_epsilon(namespan, e),
            Self::Literal(l) => visitor.visit_literal(namespan, l),
            Self::Cat(c) => visitor.visit_cat(namespan, c),
            Self::Alt(a) => visitor.visit_alt(namespan, a),
            Self::Fix(f) => visitor.visit_fix(namespan, f),
            Self::Variable(v) => visitor.visit_variable(namespan, v),
            Self::Call(c) => visitor.visit_call(namespan, c),
            Self::Lambda(l) => visitor.visit_lambda(namespan, l),
            Self::Let(l) => visitor.visit_let(namespan, l),
        }
    }
}
```

Listing 3.9: Rust code snippet showing the implementation of the visitor design pattern.

3.5.2 Translation

The first step in using the KY-λ type system is translation of the expression. Recall from section 3.2.1 how translation of Nibble is essentially call-by-name evaluation. This is implemented using a number of visitors.

The outer-most visitor is the `Reduce` visitor (a misnomer). This is what drives the translation. It searches for let statements and application expressions to translate, without stepping into lambda expressions. For the let and application expressions, it performs the appropriate substitution and then translates the result.

The substitution is performed by another visitor, `Substitute`. Given an expression and a De Bruijn index, it searches for uses of that index and replaces them with the expression. In section 2.1.4, we noted that indices of free variables change inside of lambda bodies. The target index is shifted inside of lambda bodies, and free variables in the substitute need to be renamed.

The final visitor for translation changes the De Bruijn indices of free variables in an expression.

3.5.3 Inference

Type inference is the second part of the KY-λ type system. This uses the typing rules of the KY type system to infer the type of expressions. It returns a μ-regular expression annotated with types.

Recall how Nibble has some constructs that are not in μ-regular expressions, namely let statements, application expressions and lambda expressions. The translation in the previous section eliminates all let statements and application expressions. However, it does not eliminate lone lambda expressions. Therefore, if the type inference algorithm reaches a lambda expression, it fails.

By looking back to the KY typing rules (figure 2.7), the type inference rules for other Nibble expressions can be derived. Inference for epsilon expressions is trivial – always the type τ_ϵ. By combining the KYCat rule with the KYChar rule, a literal expression `"cw"` can be shown to always has type τ_{c}.

By implementing types using `BTreeSet` from the Rust standard library to store the first and flast sets, alternation becomes simple. First, the type of each sub-expression is inferred recursively. Next, to check that the # constraint holds, the `intersection` function from the Rust standard library is used. Finally, the `append` function is used to compute the new type.

Concatenation is nearly identical to alternation. There are two differences. First, the constraint is slightly different, although the checks are almost identical. The bigger difference is that the suffix of a concatenation needs to use an unguarded context for type inference.

The last Nibble expressions to consider are fixed-point expressions and variables. Variables are a simple lookup in the type context. The type for fixed-point expressions are found using iteration. Initially, the type is assumed to be $\tau_\perp$. This guess is then added to the variable context and used to infer the type of the fixed-point body. This is repeated until the initial guess and next inferred types are equal, or there is a type error. Listing 3.10 shows the type inference process.

```rust
fn fold_fix(&mut self, fix: Fix) -> Result<_, _> {
    let mut ty = Type::default();
    let inner = fix.inner.try_into_lambda()?.get_body();

    loop {
        // ? at end exits function if there is a type error.
        let res = self.context.with_variable_type(ty.clone(), |context| {
            inner.clone().fold(&mut TypeInfer { context })
        })?;

        let next = res.get_type();

        if next == ty {
            return Ok(/* ... */);
        }

        ty = next.clone();
    }
}
```

Listing 3.10: Rust code snippet for type inference for fixed-point expressions.

Listing 3.11 shows the type context used during type inference. Its design is similar to the naming context used in the front-end, shown in listing 3.8. There are two `with` methods: `with_variable_type` for introducing new variables; and `with_unguard` for moving the guarded context into the unguarded one. These call their function arguments after pushing some data on a stack, and then pop the data off before returning. Like the front-end context, most of the work is performed by the `get_variable_type` method.

Because the abstract syntax tree of Nibble uses De Bruijn indices to represent variables, the variable context can be represented by a pair of stacks. The `vars` field stores the stack of types in the context. The type of a variable is then the item in `vars` indexed from the top of the stack. For most type systems, this completes the lookup function. However, the KY type system has both a guarded and unguarded context. Which variables are unguarded is kept track of by `unguard_points`, which stores the total number of variables that are unguarded. A simple arithmetic check then determines whether a variable is unguarded.

3.6 The Back-End: Code Generation

Code generation is the final step in Chomp. The typed μ-regular expression computed by the middle-end is converted into Rust code for both data types and parser implementations. A separate library called `chewed` contains the definitions of some data types and the parsing trait used by all Chomp-generated parsers.

The substitutions performed while translating the original Nibble expression results in a large amount of duplication of sub-expressions. In an attempt to reduce the amount of generated code, and to make integrating a Chomp-generated parser into a project easier, the back-end attempts to eliminate work generating code for duplicate expressions using interning.

This interning occurs in three phases. First, each μ-regular expression is mapped to a

```rust
pub struct Context {
    vars: Vec<Type>,
    unguard_points: Vec<usize>,
}

impl Context {
    pub fn with_unguard<F: FnOnce(&mut Self) -> R, R>(&mut self, f: F) -> R {
        self.unguard_points.push(self.vars.len());
        let res = f(self);
        self.unguard_points.pop();
        res
    }

    pub fn get_variable_type(
        &self,
        var: Variable,
    ) -> Result<&Type, GetVariableError> {
        self.vars
            .iter()
            .nth_back(var.index)
            .ok_or(GetVariableError::FreeVariable)
            .and_then(|ty| {
                if self.unguard_points.last().unwrap_or(&0) + var.index
                    >= self.vars.len()
                {
                    Ok(ty)
                } else {
                    Err(GetVariableError::GuardedVariable)
                }
            })
    }

    pub fn with_variable_type<F: FnOnce(&mut Self) -> R, R>(
        &mut self,
        ty: Type,
        f: F,
    ) -> R {
        self.vars.push(ty);
        let res = f(self);
        self.vars.pop();
        res
    }
}
```

Listing 3.11: Rust code snippet showing the type context used by type inference.

handle. This is done structurally, such that two identical μ-regular expressions receive the same handle. Next, the set of all handles that can be reached from the top-level μ-regular expression handle are computed. After this, code is generated only for the μ-regular expressions that were reached. In practice, the first and last phases are computed in one pass, and the second step collates the necessary generated code.

Mapping expressions to handles uses many caches, one for each type of expression. First, the handles of all sub-expressions are found. Next, the cache corresponding to the expression type is checked, using the sub-expression handles as keys. If the cache does not contain a handle for a key, a new one is generated.

Finding the set of reachable handles uses well-known graph algorithms.

The generated code takes the form of a stream of Rust source tokens. Creating such streams by hand would be tedious and error prone, given the complexity of parts of the generated code. Instead, a procedural macro from the `quote` library is used to allow the output token stream to be written literally. The procedural macro converts the literal Rust code into a token stream describing it.

The form of generated code depends on the μ-regular expression. ϵ is translated to the `Epsilon` type in the `chewed` library, on the assumption that almost all Nibble expressions will include an epsilon expression.

Literals are each translated to a unique unit struct. By using unique types, no data needs to be stored with the type to determine what literal it is. This means that the Rust compiler can eliminate these literal types, keeping only the side effects of their computation.

Concatenations are quite simple in terms of code generation. Each concatenation is translated into a struct, where each sub-expression is a different field. To parse the concatenation, each field is parsed in turn.

The most challenging expression form to translate into Rust code is alternation. Alternations are represented by enumerations, where each constructor corresponds to a different alternative. When parsing an alternation, which alternative to try and parse depends on the next character of input and the first sets of each alternative. Example output code is shown in listing 3.12.

Fixed-point expressions are implemented as type aliases. Variables then refer to the type of the declaring fixed point.

3.7 Summary

This chapter began by introducing the Nibble language for describing formal languages. Two type systems for Nibble were introduced – the KY-λ type system and the LM type system.

The KY-λ type system treats Nibble constructs as macros for μ-regular expressions, before deferring to the KY type system. This causes type checking to be non-terminating, and well-typed Nibble to contain some ill-typed fragments.

To address these issues, I detailed the LM type system. By combining the Hindley-Milner type system with the KY type system, it gives Nibble expressions polymorphic types. To account for type variables, the constraints from the KY type system were moved from the premise to the conclusion of the typing rules, to be solved separately.

```rust
// Parser for `match (_|"a")|("b"|"B")|"c";`
enum Ast {
    Branch1(/*...*/); //   _|"a"
    Branch2(/*...*/); // "b"|"B"
    Branch3(/*...*/); // "c"
}
impl Parse for Ast {
    fn take<P: Parser + ?Sized>(input: &mut P) -> Result<Self, TakeError> {
        match input.peek() {
            Some('a')   => Ok(Self::Branch1(input.take()?)),
            Some('b')
            | Some('B') => Ok(Self::Branch2(input.take()?)),
            Some('c')   => Ok(Self::Branch3(input.take()?)),
            // Because `_|"a"` contains the empty string,
            // we can always take a member of `_|"a"`.
            _           => Ok(Self::Branch1(input.take()?))
        }
    }
}
```

Listing 3.12: Chomp-generated Rust code for parsing an alternation.

The chapter started discussing the Chomp parser generator, which uses the KY-λ type system. Chomp is both implemented in and targets Rust to exploit the procedural-macro system. Like most translators, Chomp is split into three "ends".

The front-end of the Chomp generator takes a concrete stream of characters and produces an abstract syntax tree of a Nibble expression. An external lexer converts the character stream to a sequence of Rust tokens. A parser then converts the Rust tokens into a concrete syntax tree. A normalisation step then expands syntactic sugar and introduces De Bruijn indices.

The middle-end is responsible for type inference using the KY-λ type system. First, the Nibble expression represented by the abstract syntax tree is translated to a μ-regular expression with call-by-name evaluation. Next, the KY type system is used to infer the type of this μ-regular expression.

Finally, the back-end performs code generation. The type-annotated μ-regular expression is expanded into a sequence of Rust tokens describing a parser for its language. This process utilises interning to eliminate the replication of expressions introduced by the earlier translation.

Chapter 4

Evaluation

I now evaluate whether the implementation has fulfilled the goals of the project. In section 4.1 I demonstrate the success criterion was satisfied. Following this, I compare the benefits of using Nibble to BNF in section 4.2. Section 4.3 contrasts integrating Chomp into a project with integrating `lalrpop`, a parser generator for Rust using BNF-inspired syntax. I pay particular attention to the impact of the KY type system on describing formal languages in Nibble. Finally, in section 4.4 I analyse the results of benchmarks to determine the run-time cost of using Chomp-generated parsers.

4.1 Meeting the Success Criterion

The success criterion required that the Chomp parser generator: "can generate a parser implementation for [the Nibble language] that produces identical output to the hand-written Nibble parser". I start by considering whether I have produced the requisite deliverables, then move on to checking I have fulfilled the success criterion.

I described the Nibble language in section 3.1, which is a new language for describing formal languages. By embedding μ-regular expressions in Nibble, Nibble provides a new way of describing μ-regular expressions. The introduction of let statements and lambda expressions shifts Nibble away from its mathematical roots in μ-regular expressions towards a more programmatic style.

Whilst not explicitly stated in the proposal, it was required to design an appropriate type system for Nibble that was compatible with the KY type system. I have achieved this in two different ways. The KY-λ type system described in section 3.2.1 describes the minimal changes to the KY type system necessary for use with Nibble. Section 3.2.2 describes the LM type system, which is a fresh type system directly on Nibble compatible with the KY type system.

Another core requirement was the creation of Chomp – a parser generator taking Nibble as input. Chapter 3 discusses the implementation of Chomp, including the three parts described in the original implementation.

The rest of this section is concerned with the bootstrapping of Chomp required to satisfy the success criterion. All of this code is in the `autonibble` directory of the repository.

First, Nibble is sufficiently expressive to be able to describe itself. This is demonstrated in the `autonibble/src/lib.rs` file. This is a significant result – Nibble syntax is similar

```
// BNF
<xs>    ::= "x" | "x" <xs>
<ys>    ::= "y" | "y" <ys>
<start> ::= <xs> <ys>

// Nibble
let list inner = !(/list/ inner . (_ | list));
match list "x" . list "y";
```

Figure 4.1: Comparison of BNF and the Nibble language, demonstrating that Nibble can abstract away patterns such as lists.

to that of many programming languages used in industry. Further, the Chomp generator can process this self-description to produce the AutoNibble parser.

The rest of the code in `autonibble/src/lib.rs` is there to make AutoNibble a complete front-end for Chomp. It performs the same normalisation process on AutoNibble's concrete syntax tree as Chomp does on its own concrete syntax tree.

Finally, I directly compare the outputs of the full AutoNibble front-end and Chomp's front-end. This is performed on the test cases present in the `autonibble/tests/compare/` directory.

4.2 Comparing Nibble and BNF

Untyped Nibble and BNF, introduced in section 2.1.1, are both languages for describing formal languages. By comparing the features and ergonomics of Nibble and BNF, I can conclude whether the Nibble language has the potential to join BNF as an effective way to describe formal languages.

First, I explore the classes of formal languages that BNF and Nibble can describe. As stated in section 2.1.1, BNF can describe all context-free languages. Nibble likely also only accepts context-free languages, despite translation being able to perform arbitrary computation.

A fundamental principle of programming is DRY – don't repeat yourself. Duplicating code makes maintenance more difficult. A conscious effort has to be made to keep all the duplicates in sync, and any bug found in one duplicate will be present in all others. Nibble avoids replication using let statements for verbatim copies, and lambda expressions for parametric copies.

In contrast, whilst a new top-level form in BNF can eliminate verbatim and some cases of parametric repetition, other forms of parametric replication cannot be removed. Figure 4.1 shows how Nibble can remove replication of list definitions which cannot be removed from BNF. Whilst the BNF forms `<xs>` and `<ys>` have the same shape, there is no way to abstract that out. On the other hand, the Nibble variable `list` provides a parametric list definition.

Another difference between BNF and Nibble is the variable scope rules. The mutually-recursive global scope of BNF allows other forms to be referenced without qualification from anywhere. Nibble uses much more restrictive non-recursive lexical scoping rules – only variables defined previously can be used. This is shown in figure 4.2. In BNF, the `<term>` form can directly refer to the `<expr>` form, even though it is defined later.

```
// BNF
<term> ::= <number> | "(" <expr> ")"
<expr> ::= <term>   | <term> + <expr>

// Nibble
let term expr = number | "(" . expr . ")";
let expr      = !(/expr/ term expr . (_ | ("+" | "-") . expr));
```

Figure 4.2: Comparison of BNF and the Nibble language demonstrating the difference in variable scope rules.

```
// lalrpop
Sign : bool = {
    "+" => true,
    "-" => false,
}
SignedNumber = <s : Sign?> <n : Number>;

// Chomp
let signed_number = number | ("+"|"-") . number;
```

Figure 4.3: Comparison of `lalrpop` and the Chomp parser generator demonstrating that `lalrpop` can accept optional prefixes before a string.

Conversely, Nibble requires `term` to be a function with `expr` as a parameter. Also note that Nibble makes recursion explicit using the fixed point, compared to BNF's implicit recursion.

4.3 Integrating Chomp

Whilst Nibble is interesting in its own right, most of its use in practice depends on Chomp. Here, I compare Chomp and KY-λ-typed Nibble against `lalrpop`, a parser generator for Rust using BNF-inspired syntax. We first discuss the language classes accepted by both KY-typed Nibble and `lalrpop`, and the consequences on the form of descriptions. Then I look at how to integrate both Chomp and `lalrpop` into a project build system.

Due to limitations of the KY type system, KY-λ-typed Nibble can only describe LL languages. Meanwhile, `lalrpop` can describe the wider class of LR languages.

One practical example of these differences in allowed language descriptions is shown in figure 4.3. This shows how to parse an optionally-signed number in both `lalrpop` and KY-λ-typed Nibble. `lalrpop` has no problems with having the optional sign as a prefix to the number. On the other hand, the KY-λ typing rules use the $\circledast$ constraint to prevent the prefix of a concatenation from containing the empty string. This means `number` must be used twice – once for unsigned numbers, and another for signed numbers.

Another major difference between `lalrpop` and KY-λ-typed Nibble is that the KY-λ type system only accepts left-factored expressions. Figure 4.4 shows how `lalrpop` and KY-λ-typed Nibble would describe the small language consisting of the three strings "bat", "band" and "brook". In `lalrpop`, all three strings appear as literals. However, KY-λ-typed Nibble factors out the common prefixes.

The Nibble has to be left-factored to satisfy the # constraint – each alternative must have a disjoint First set. Consider listing 4.1, which shows the output parser implementation

```
// lalrpop
Language = {
    "bat",
    "band",
    "brook",
}

// Chomp
match "b" . ("a" . ("t" | "nd") | "rook");
```

Figure 4.4: Comparison of `lalrpop` and the Chomp parser generator that demonstrates that `lalrpop` does not need left-factoring.

for the unfactored expression. Each alternative has the same condition of `Some('b')`. As there is no backtracking, parsing this unfactored expression would not perform as expected.

```
impl Parse for Ast {
    fn take<P: Parser + ?Sized>(input: &mut P) -> Result<Self, TakeError> {
        match input.peek() {
            // "bat"
            Some('b') => Ok(Self::Bat(input.take()?),
            // "band"
            Some('b') => Ok(Self::Band(input.take()?),
            // "brook"
            Some('b') => Ok(Self::Brook(input.take()?),
            // Errors
            None      => Err(TakeError::EndOfInput(/*...*/)
            Some(c)   => Err(TakeError::BadBranch(/*...*/)),
        }
    }
}
```

Listing 4.1: Chomp-generated Rust code assuming left-factoring was unnecessary. Notice how the conditions for three of the branches are all `Some('b')`.

Left factoring causes many problems. Firstly, the resulting expression is more difficult to read. This hinders developers trying to understand what language the expression is describing. Secondly, searching for a literal can now fail. When searching for a word, users start at the left and work towards the right. However, left-factoring separates the left-most characters from the rest of a literal. More generally, left factoring obfuscates the semantics of an expression.

Additionally, left factoring introduces replication. Consider a programming language like Java. Two common statements are variable assignments and if statements. Variables can be any sequence of lowercase characters that is not a keyword. Figure 4.5 shows fragments of `lalrpop` and KY-λ-typed Nibble that match these two types of statements. The `lalrpop` fragment is as expected – `"if"` does one thing and variables do another. The Nibble fragment introduces a lot of repetition to avoid left-factoring.

`lalrpop` avoids the left factoring problem via two mechanisms. The first is that it accepts a larger range of language descriptions, as described above. Secondly, it has a lexing stage. A lexer splits the input character stream into different tokens, which are then given to

```
// lalrpop
Statement = {
    "if" "(" <e: Expression> ")" "{" <then: Statements> "}",
    <var: r"[a-z]+"> "=" <e : Expression> ";" => {
        if !var.is_keyword() {
            // all good
        } else {
            // error!
        }
}

// Chomp
let if_body = "(" . expr . ")" . opt ws . "{" . stmts . "}";
let assign_body = opt ws . "=" . expr . ";";
let stmt =
  "i" . ( "f" . ( if_body
                | plus a_to_z . assign_body)
        | assign_body
        | ("a"|/* ... */|"e"|"g"|/* ... */|"z") . star a_to_z . assign_body
        )
| ("a"|/* ... */|"h"|"j"|/* ... */|"z") . star a_to_z . assign_body
;
```

Figure 4.5: Comparison of `lalrpop` and the Chomp parser generator showing Chomp's left-factoring for a Java-like language.

the parser. The lexer decides whether a particular sequence of characters is a keyword or just a regular name.

Chomp does not use a lexer. Whilst lexers are useful to avoid left factoring, they are not necessary. Throughout the project, a more powerful type system was judged to be a more useful addition to Chomp. Given a sufficiently smart type system, the left factoring problem can be eliminated entirely.

Another feature of `lalrpop` not present in Nibble is the use of semantic actions. These are small computations performed by the parser during parsing, often used to build the abstract syntax tree.

Chomp can bypass semantic actions using Rust's trait system. All data types used by Chomp-generated parsers are made publicly accessible. Developers using Chomp can define trait implementations for these data types, giving them arbitrary properties. This is how AutoNibble converts its concrete syntax tree into an abstract syntax tree for the Nibble language – it uses the same `Convert` trait used in the front-end of Chomp.

Chomp is easier to integrate into projects than `lalrpop`. Chomp uses Rust's procedural-macro system to transform a Nibble expression embedded in a Rust file into a Chomp-generated parser. `lalrpop` instead requires an external build script to first convert `lalrpop` descriptions into Rust source code, which has to be explicitly included in the project.

Finally, Chomp computes explicit types for Nibble expressions. This has benefits in terms of error reporting over `lalrpop`. Under the KY type system, an μ-regular expression is only rejected if one of the $\circledast$ or # constraints does not hold. The types of the relevant

sub expressions then trace the origin of the problem. Using `lalrpop`, parser generation only fails if there is a problem generating the output parser.

4.4 Performance of Chomp-generated Parsers

One claim made by Krishnaswami and Yallop [5] was that the parsing algorithm they presented produced linear parsers. After a discussion on the benchmark methodology, I test whether this claim extends to Chomp and Chomp-generated parsers and compare the performance of Chomp-generated parsers to `lalrpop` and handwritten parsers.

4.4.1 Methodology

Benchmarks were performed using the `criterion` library for Rust. By analysing the timing data produced by `criterion`, it is possible to estimate how long it takes a function to run and the variance of that measurement. To estimate the asymptotic behaviour of a parser on the size of the input, benchmarks are performed for many different input lengths.

First, a description of how `criterion` measures performance. There are two stages it takes to benchmark every function: warm-up and measurement. The warm-up stage prepares the system for executing the desired function. The function is executed once, then twice, then four times and so on, until the total warm-up time exceeds three seconds.

Measurements are split into samples. Each sample consists of a number of iterations, or executions, of the function. `criterion` measures the duration of each of one hundred samples. For the nth sample, the number of iterations $I_n = nd$, where d is a scaling factor chosen so that all one hundred samples are collected in approximately thirty seconds. d is calculated by the number of iterations and duration of the warm-up period.

Assume that a sample measurement T_n is given by the equation $T_n = M + nt + \epsilon$, where M is a constant measurement error, t is the wall-time of one function iteration, and ϵ is a normally distributed error. By performing linear regression on the sample measurements with respect to the sample size, you can compute $\bar{t}$ and $\bar{\sigma^2}$, an estimate for t and the variance of the estimate.

Due to the deterministic nature of parsers, and the number of iterations performed during benchmarking, wall-time is a suitable proxy measurement for processor time. The primary source of non-determinism in measurements will be hardware interrupts, which were minimised by running benchmarks on an idle system.

Only one benchmark was performed at a time. Every benchmark used a single thread of execution. The measurements were collected on a desktop computer with the following specifications:

- Processor: AMD Ryzen 5 3600 @ 4.2 GHz
- Memory: 16 GiB DIMM DDR4 2400 MHz RAM
- OS: Arch Linux 5.11
- Rust Version: 1.51.0

4.4.2 Performance of AutoNibble

AutoNibble is the most complex example of Nibble that exists, and hence produces the most complex Chomp-generated parser. Roughly eighty lines of Nibble are translated into over 4800 lines of Rust source code. I compare the performance of AutoNibble to the Nibble parser in Chomp's front-end.

Figure 4.6 shows the benchmark results for AutoNibble and the handwritten Nibble parser in Chomp. The benchmarks were computed on Nibble expressions of various lengths, ranging from 12 to 3096 bytes.

Because AutoNibble and Chomp's parser produce different concrete syntax trees, the benchmarks include normalisation to the Nibble abstract syntax tree, as detailed in section 3.4.

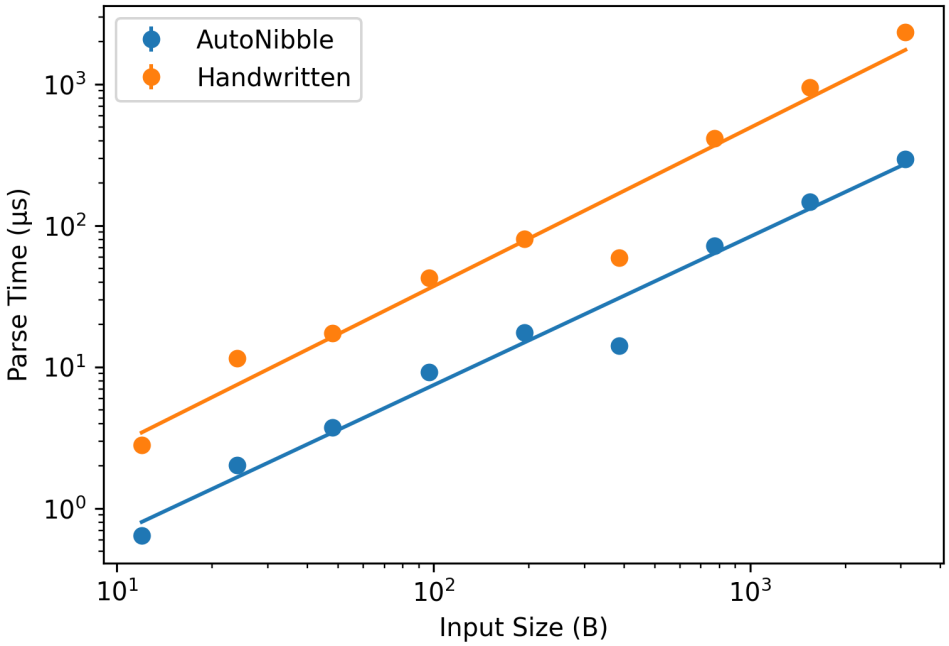

Figure 4.6: Graph of wall-time to parse a Nibble expression against the length of the Nibble expression in bytes. This compares the handwritten Nibble parser used by the Chomp parser generator against AutoNibble.

The trend in the data for both AutoNibble and the handwritten Nibble parser suggests that both parsers operate in linear time. Note that even the largest input is quite small, so behaviour might change for bigger inputs.

This graph also suggests that AutoNibble outperforms the handwritten parser. This is an encouraging result, because it suggests Chomp-generated parsers can be used in some practical applications to improve performance.

4.4.3 Performance against `lalrpop`

Whilst AutoNibble is the most complex example of Nibble, using Nibble as the benchmark language has some issues. There are very few examples of Nibble, and the complexity of writing descriptions of the Nibble language for other parser generators seems daunting, especially given how the language was continuously evolving. Instead, Chomp-generated,

handwritten and `lalrpop` parsers were created for two widely-used languages: JSON and arithmetic.

Figures 4.7 and 4.8 show the benchmark results for JSON and arithmetic respectively. For JSON, each parser produced a Rust datatype corresponding to the JSON value. The input data ranges from roughly one hundred bytes in length to 28000 bytes, and was taken from a weather API. For arithmetic, the result of evaluating the expression was computed. The length of data goes from 100 bytes up to 32000 and was randomly generated in advance of writing the benchmark.

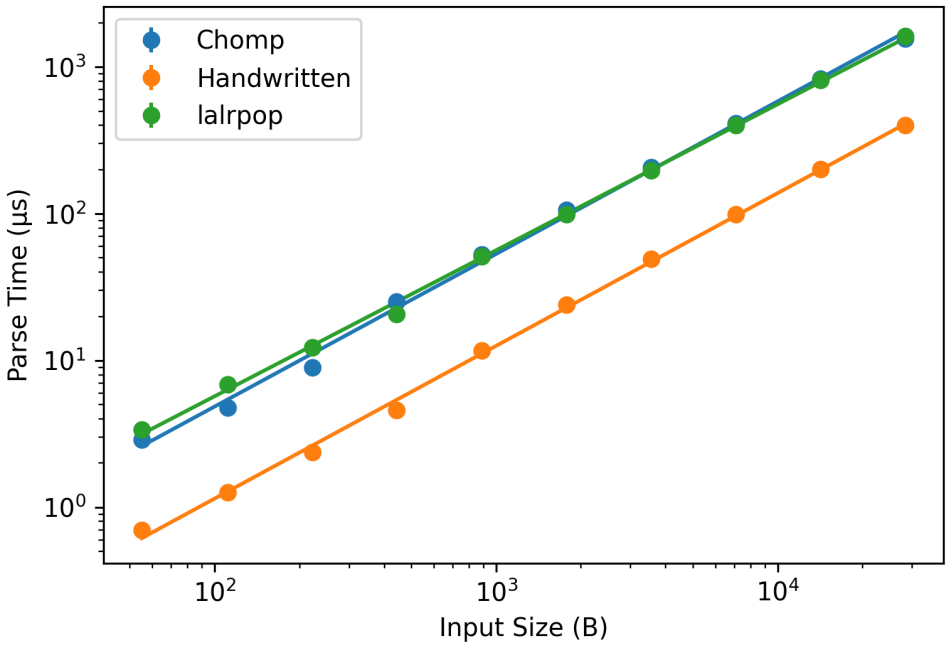

Figure 4.7: Graph of wall-time to parse a JSON value string against the length of the string in bytes. This compares a Chomp-generated parser, a `lalrpop`-generated parser and a handwritten parser.

In both cases, the Chomp-generated parsers appear to run in linear time. This gives further evidence that the parsers truly do have linear performance.

In these two tests, the handwritten parser outperforms both generated parsers. The handwritten parser can immediately produce values of the appropriate type. On the other hand, the generated parsers first produce a concrete syntax tree before converting it to the correct type. Chomp-generated parsers create the full tree and are then converted. `lalrpop` parsers create fragments of concrete syntax trees before performing conversion.

The performance of `lalrpop`- and Chomp-generated parsers is comparable, with the Chomp-generated parser outperforming the `lalrpop` parser for arithmetic expressions. Therefore, if the usability issues described in section 4.3 are resolved, the Chomp generator could compete with traditional parser generators in general use.

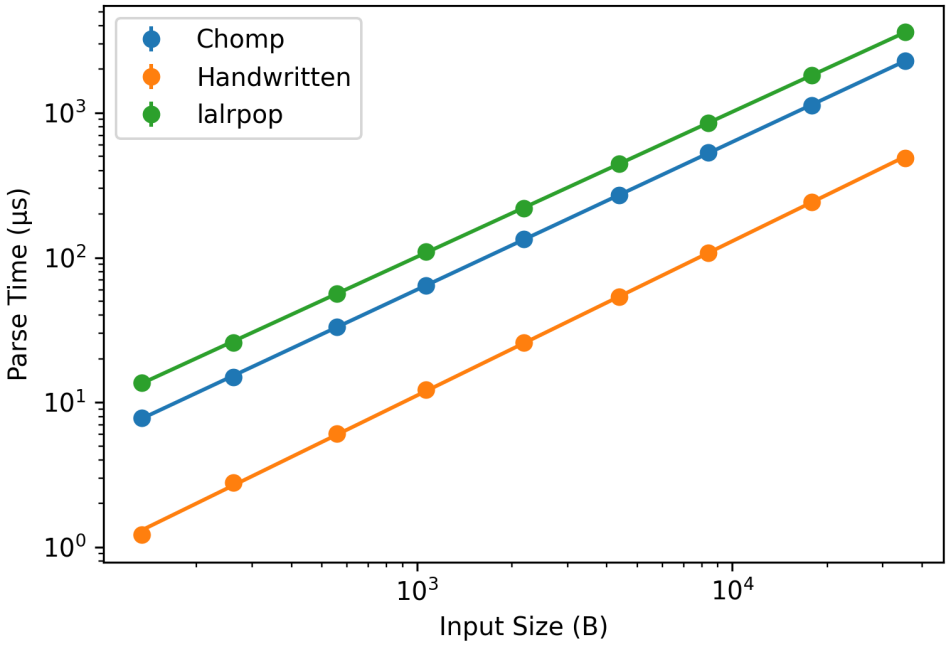

Figure 4.8: Graph of wall-time to parse and evaluate an arithmetic expression against the length of the arithmetic expression in bytes. This compares a Chomp-generated parser, a `lalrpop`-generated parser and a handwritten parser.

Chapter 5

Conclusion

This project was a success – I completed the success criterion of implementing AutoNibble and completed an extension of adding functions to the Nibble language.

My project set out to develop a parser generator based on μ-regular expressions. By creating the Nibble language, I designed an ergonomic way to describe μ-regular expressions (section 3.1). I then designed two type systems for the Nibble language, which extend the KY type system to the syntax of Nibble (section 3.2). One of these type systems was implemented in the Chomp parser generator (section 3.3), to produce parsers implemented in Rust.

I proved that the Nibble language is capable of being able to describe itself. Further, using the Chomp parser generator to create the AutoNibble parser, I found that AutoNibble outperforms a hand-written parser. I also found that Chomp-generated parsers have comparable performance to traditionally-generated parsers.

5.1 Lessons Learned

The time table proposed in my original project proposal was quite optimistic. I underestimated the effect that external time pressures – such as supervisions during Lent term – would have on the time I could spend on this project. Fortunately I had allocated plenty of slack time in the proposed schedule. However, a more balanced schedule would have been the better option.

The code structure used by the Chomp parser generator has three separate modules for the front-, middle-, and back-ends. As described in section 2.1.4, this is the architecture typically used by translators in industry. When the interface to each end is stable, there is little coupling between the three ends. However, because the Nibble language and the Chomp parser generator are both brand-new, the interface for each end was constantly evolving. Every new feature changed one of these critical interfaces, so almost every file had to be modified. More research should have been spent trying to find an architecture that could withstand the fast development cycle used by the project.

The end-to-end tests for this project were written relatively early on in the evolution of the Nibble language and the Chomp parser generator. Several times, these tests and benchmarks had to be almost completely rewritten to account for changes in the syntax of Nibble or the structure of Chomp-generated parsers. Writing more unit tests could have reduced the number of necessary end-to-end tests, and, since unit tests are faster

to change than end-to-end tests, more time could have been spent adding new features instead of maintaining tests.

5.2 Future Work

The LM Type system generates the potential for lots of future work. First, there is no formal proof that it is a correct type system. Whilst the modifications made to the Hindley-Milner and KY type systems are small, these changes could require imaginative solutions to prove soundness and completeness properties. Formal proofs of these properties are necessary to make sure the LM type system achieves what it sets out to complete.

Secondly, the Chomp parser generator could be modified to use the LM type system. Even without proof, a practical implementation can provide evidence for the claims made by the LM type system. It can also help to justify some design decisions of the LM type system, such as the choice to use structural unification for types. Work on this implementation can proceed alongside a proof for the type system. Using a dependant-type system such as Agda could allow for the proof and implementation to be tightly coupled, in the sense that changes to one necessitates changes to the other.

I conjectured that the LM type system does not accept expressions for a larger class of languages than the KY type system does. Unfortunately both type systems reject some simple expressions that can be parsed in linear time, for example the μ-regular expression $(\epsilon \vee a) \cdot b$, representing an optional prefix symbol a before the symbol b. Searching for a type system for μ-regular expressions that permits accepts a wider range of expressions would make writing well-typed Nibble easier.

In section 3.2.1, I discussed how translation of Nibble expressions can perform arbitrary computations. In contrast, the LM type system performs no such computation, because of the syntactic nature of the type system. The Nibble language and the LM type system could be extended to support additional data types, such as natural numbers. This would make it possible to express a richer set of parser combinators in Nibble.

Chapter 6

References

[1] B. W. Boehm. "A Spiral Model of Software Development and Enhancement". In: *Computer* 21.5 (1988), pp. 61–72. DOI: 10.1109/2.59.

[2] Noam Chomsky. "On Certain Formal Properties of Grammars". In: *Information and Control* 2.2 (1959), pp. 137–167. ISSN: 0019-9958. DOI: 10.1016/S0019-9958(59)90362-6.

[3] N. G. de Bruijn. "Lambda Calculus Notation with Nameless Dummies, a Tool for Automatic Formula Manipulation, with Application to the Church-Rosser Theorem". In: *Indagationes Mathematicae (Proceedings)* 75.5 (1972), pp. 381–392. ISSN: 1385-7258. DOI: 10.1016/1385-7258(72)90034-0.

[4] R. Hindley. "The Principal Type-Scheme of an Object in Combinatory Logic". In: *Transactions of the American Mathematical Society* 146 (1969), pp. 29–60. ISSN: 00029947. DOI: 10.2307/1995158.

[5] Neelakantan R. Krishnaswami and Jeremy Yallop. "A Typed, Algebraic Approach to Parsing". In: *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI 2019. Phoenix, AZ, USA: Association for Computing Machinery, 2019, pp. 379–393. ISBN: 9781450367127. DOI: 10.1145/3314221.3314625.

[6] Haas Leiß. "Towards Kleene Algebra with Recursion". In: *Computer Science Logic*. Ed. by Egon Börger et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 1992, pp. 242–256. ISBN: 9783540472858. DOI: 10.1007/BFb0023771.

[7] Robin Milner. "A Theory of Type Polymorphism in Programming". In: *Journal of Computer and System Sciences* 17.3 (1978), pp. 348–375. ISSN: 0022-0000. DOI: 10.1016/0022-0000(78)90014-4.

[8] J.B. Wells. "Typability and Type Checking in the Second-Order λ-Calculus are Equivalent and Undecidable". In: *Proceedings Ninth Annual IEEE Symposium on Logic in Computer Science*. July 1994, pp. 176–185. DOI: 10.1109/LICS.1994.316068.

Appendix A

Project Proposal

Introduction and Description

A parser is a function that takes a string and produces a structured output, often a tree. Parsers can also return errors, if the string is not in the language.

There are two approaches to writing parsers for context-free languages. The traditional approach frames a grammar as a set of production rules. A separate tool, such as Yacc or Antlr, generates code from these rules, implementing a parser for the grammar. These tools enforce restrictions on the production rules to make sure the parser implementation can operate in linear time.

Some of these generation tools can accept ambiguous production rules: the same string can be produced by many derivations. The tools will pick an arbitrary derivation to produce for a string. This could hide a problem in the grammar that the user is unaware of.

The traditional approach can be seen as a bottom-up parser. They take a single character at a time and then try to build the derivation tree from it and previous steps. *Parser combinator* libraries take an entirely different approach.

A parser combinator is a function that takes some number of parsers and produces a new parser. In a parser combinator library, there are a small number of primitive parsers, which match a single character or the empty string, as well as a few basic parser combinators. The user is typically able to define more parser combinators to help express more complex parsers.

These parsers operate by starting at the root of the derivation tree and walking down it, with different combinators taking branches depending on what characters they encounter. Most implementations include some form of backtracking, in case the parser went down the wrong route. This makes the execution time unclear.

In a recent result, Krishnaswami and Yallop [1] defined a calculus, *context-free expressions*, for defining a parser. This is similar in style to parser combinators. There are some primitive expressions, and operators that produce expressions from others.

One major benefit of context-free expressions over parser combinator libraries comes from a type system. A well typed expression describes a parser that is both unambiguous and that runs in linear time. Additionally, context-free expressions are built from a small

number of primitives. This makes it easy to convert a context-free expression into a parser implementation.

Starting Point

Context-Free Expressions

Context-free expressions as presented by Krishnaswami and Yallop [1] are not part of the tripos. Over the summer, I devoted some time to studying them and reading literature around related concepts.

Rust

I plan to implement the project using Rust. It is a popular modern language, with a good history for being used to write compilers despite being relatively young. I have used Rust in hobby projects for several years and am familiar with the contents of *The Rust Book*.

Another advantage of Rust is that it has built-in benchmarking and staged compilation. This should lead to much more interesting evaluation methods than other languages.

Substance and Structure of the Project

The goal of the project is to design a new syntax for context-free expressions, named Nibbler, and write a parser generator using it, named Chomp.

The syntax for context-free expressions is oriented towards mathematical applications. Nibbler will be in a more programmatic style.

The project will consist of three stages, implemented in this order:

- Lexer and parser for Nibbler
- Type system
- Output Rust code generation

Generating a correct parser depends on the correctness of each individual component. Therefore each part will need extensive unit tests, as well as full integration and end-to-end tests.

I expect most development time will go into the type system. The lexer and parser should be relatively simple to implement. Context-free expressions also have only a small number of primitives and operations, significantly reducing the complexity of code generation.

The target language will be Rust to allow for more interesting evaluation. Additionally, features of Rusts type system—in this case ownership—will make it easier to determine if the parser will run in linear time.

Success Criterion

The project will be a success if Chomp can generate a parser implementation for Nibble that produces identical output to the hand-written Nibble parser. We will call the Chomp-generated implementation of the Nibble parser *AutoNibble*.

To determine whether AutoNibble is identical to the hand-written parser, we will modify Chomp to use AutoNibble. Lets call this modified version of Chomp *AutoChomp*. If Chomp and AutoChomp produce identical outputs, then Nibble and AutoNibble will produce identical outputs, thus meaning that the original Chomp implementation is correct.

This approach is very similar to the method used to check the validity of a bootstrapped compiler. Instead of replacing the whole compiler, we only change the parser implementation.

Quantitative evaluation can involve comparing the execution performance of a Chomp-generated parser against both hand-written and traditionally-generated parsers for several languages. This can include simple languages like JSON and Nibble, to large languages like C. The correctness can also be evaluated by comparing the output of a Chomp-generated parser against traditional parser generators.

Possible Extensions

User-Defined Parse Errors

When given an invalid string, a parser produces an error. This error is shown to the user so the can correct the issue. This extension will allow defining custom error messages within the Nibbler language. This allows for more descriptive error messages that what Chomp could automatically generate.

Polymorphic Operators

The type checking for context-free expressions [1] requires that any operators are expanded in-line. This results in a potentially exponential increase in expression size. This extension will devise a way to type check operators in a polymorphic way. Performance of the two different approaches can be compared quantitatively.

Better Chomp Errors

There are some inputs that will fail to type check. In some of these cases, the modifications a user has to make are small. Chomp should be able to suggest to the user how to correct the problem.

Semantic Expressions

A common step in many compilers is constant evaluation. Another is restructuring the concrete syntax tree into an abstract one. Both of these processes can be achieved by semantic expressions: simple annotations to Nibbler. This can range from basic arithmetic and pattern matching to higher order functions acting on lists.

Work Plan

Work will be carried out in ten two-week work packages. The first half will be completed before the progress report deadline and will focus on fulfilling the success criterion. The second half will focus on extensions and production of the final report.

This allows for some slack times and catch-up times. Slack times are periods where work might progress slowly. Catch-up times are allocated to allow for delays.

Weeks 1 and 2: October 22 — November 4

Lexer and Parser

Create the project skeleton, including version control and backup systems. Create a few small test Nibbler files. Create unit tests for each language component. Implement lexer and parser.

Goal: project skeleton

Goal: lexer

Goal: parser

Weeks 3, 4 and 5: November 5 — November 25

Type Checking

Create unit tests and implement type checking for primitive expressions and operators. Create integration test for parsing and type-checking simple and complex Nibbler files.

Goal: type checker

Week 6: November 26 — December 2

End of Term

Weeks 7 and 8: December 3 — December 16

Code Generation

Create unit tests and implement code generation. Create integration tests for type-checking and generating code. Create end-to-end tests for Nibbler files.

Goal: code generation

Weeks 9 and 10: December 17 — December 30

Christmas Break

Weeks 11 and 12: December 31 — January 13

Initial Evaluation

Define Nibbler using Nibbler. Use Chomp to create AutoNibble. Create AutoChomp using AutoNibble. Evaluate performance and correctness of AutoChomp against Chomp.

Goal: meet success criterion

Weeks 13, 14 and 15: January 14 — February 3

Further Evaluation

Create a conversion tool from BNF to Nibbler. Use Chomp to generate parsers for several real-world languages. Compare performance and correctness of Chomp to traditional parser generators. Create progress report.

Goal: complete evaluation

Goal: complete progress report

Weeks 16 and 17: February 4 — February 17

Research and Start Extensions

Research how to proceed with extensions. Make an implementation and evaluation plan. Send plans to supervisor.

Goal: plan extensions

Weeks 18 and 19: February 18 — March 3

Work on Extensions

Weeks 20 and 21: March 4 — March 17

Start Draft Report

Draft Introduction and Preparation chapters of dissertation. Share draft chapters with supervisor. Continue work on extensions.

Goal: draft introduction

Goal: draft preparation

Weeks 22 and 23: March 18 — March 31

Continue Draft Report

Draft Implementation and evaluation chapters. Share draft chapters with supervisor. Continue work on extensions. Address feedback about Introduction and Preparation chapters.

Goal: draft implementation

Goal: draft evaluation

Goal: finish implementing extensions

Weeks 24, 25 and 26: April 1 — April 21

Finish Draft Report and Extensions

Evaluate extensions. Draft Conclusion chapter. Share full draft with supervisor and DoS.

Goal: draft conclusion

Goal: evaluate extensions

Weeks 27, 28 and 29: April 22 — May 12

Final Touches

Follow feedback given by DoS and supervisor.

Goal: submit final report

Resources

I plan to use my own computer. It has a 1.8 GHz CPU with 4 GB of RAM and 128 GB of storage. It runs Arch Linux as an operating system.

To mitigate data loss, local data backups are taken every 15 minutes. Data is also backed up remotely every hour. All code will also be controlled in a git repository, and will be pushed to two different git servers.

All data backups can be accessed from many alternate sources. In the event of a software or hardware failure, work can proceed via the MCS until the system is restored.

References

[1] Neelakantan R. Krishnaswami and Jeremy Yallop. "A Typed, Algebraic Approach to Parsing". In: *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI 2019. Phoenix, AZ, USA: Association for Computing Machinery, 2019, pp. 379–393. ISBN: 9781450367127. DOI: 10.1145/3314221.3314625.