An Extensible Framework for Synthesizing Efficient, Verified Parsers

by

Jason S. Gross

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Science in Computer Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2015

Author ....
Department of Electrical Engineering and Computer Science
August 19, 2015

Certified by ....
Adam Chlipala
Associate Professor without Tenure of Computer Science
Thesis Supervisor

Accepted by ....
Leslie A. Kolodziejski
Chair, Department Committee on Graduate Students

An Extensible Framework for Synthesizing Efficient, Verified Parsers

by

Jason S. Gross

Submitted to the Department of Electrical Engineering and Computer Science
on August 19, 2015, in partial fulfillment of the
requirements for the degree of
Master of Science in Computer Science and Engineering

Abstract

Parsers have a long history in computer science. This thesis proposes a novel approach to synthesizing efficient, verified parsers by refinement, and presents a demonstration of this approach in the Fiat framework by synthesizing a parser for arithmetic expressions. The benefits of this framework may include more flexibility in the parsers that can be described, more control over the low-level details when necessary for performance, and automatic or mostly automatic correctness proofs.

Thesis Supervisor: Adam Chlipala
Title: Associate Professor without Tenure of Computer Science

Acknowledgments

Thank you, Mom, for encouraging me from my youth and supporting me in all that I do. Thank you, Allison Schneider, for providing emotional support and encouragement when I was discouraged about the state of my thesis. Thank you, officemates, Benjamin Delaware, CJ Bell, Peng Wang, Clément Pit–Claudel, for your work with me on Fiat, and for bouncing ideas back and forth with me about parsers. Thank you, Natasha Plotkin, for your invaluable role as my human compiler, when I was having trouble with the duration of my write–compile–revise loops. Last, and most of all, thank you, Adam Chlipala, for your patience, guidance, advice, and wisdom, during the writing of this thesis, and through my research career.

Contents

Chapter 1

Parsing Context-Free Grammars

We begin with an overview of the general setting and a description of our approach to parsing. Our parser can be found on GitHub, in the folder `src/Parsers` of `https://github.com/JasonGross/fiat`.[1]

Why parsing? Parsing, a well-studied algorithmic problem, is the first step for a variety of applications. To perform meaningful analysis on text of any written language, the first step is generally to break the text up into words, sentences, and paragraphs, and impose some sort of structure on the words in each sentence; this requires parsing. To compile, interpret, or execute a program, a computer first needs to read its code from the disk and turn the resulting stream of bytes into a structured representation that it can manipulate and run; this requires parsing. Parsing JavaScript, in particular, is a useful application; JavaScript has become the de facto language of the web. Unlike machine code, which was designed to be easy for computers to manipulate quickly, JavaScript was designed to be relatively easy to read by a person. Having responsive dynamic webpages requires downloading and interpreting JavaScript quickly; if the JavaScript parser being used is slow, there's no hope of loading content without frustrating delays for the user.

1.1 Parsing

The job of a parser is to decompose a flat list of characters, called a *string*, into a structured tree, called a *parse tree*, on which further operations can be performed. As a simple example, we can parse `"ab"` as an instance of the regular expression `(ab)`*, giving this parse tree, where we write $\cdot$ for string concatenation.

[1] The version, as of the writing of this thesis, is 2c1aa766b9923ce75f26d6477f9fd5d8b6d3f9c1. The Fiat homepage is `http://plv.csail.mit.edu/fiat/`. The more general, dependently typed version of the parser is at `https://github.com/JasonGross/parsing-parses`.

$$\dfrac{\dfrac{\dfrac{}{\texttt{"a"} \in \texttt{'a'}} \qquad \dfrac{}{\texttt{"b"} \in \texttt{'b'}} \qquad \dfrac{\dfrac{}{\texttt{""} \in \epsilon}}{\texttt{""} \in \texttt{(ab)}^*}}{\texttt{"a"} \cdot \texttt{"b"} \cdot \texttt{""} \in \texttt{ab(ab)}^*}}{\texttt{"ab"} \in \texttt{(ab)}^*}$$

Our parse tree is implicitly constructed from a set of general inference rules for parsing. There is a naive approach to parsing a string s: run the inference rules as a logic program. Several execution orders work: we may proceed bottom-up, by generating all of the strings that are in the language and not longer than s, checking each one for equality with s; or top-down, by splitting s into smaller parts in a way that mirrors the inference rules. In this thesis, we present an implementation based on the second strategy, parameterizing over a "splitting oracle" that provides a list of candidate locations at which to split the string, based on the available inference rules. Soundness of the algorithm is independent of the splitting oracle; each location in the list is attempted. To be complete, if any split of the string yields a valid parse, the oracle must give at least one splitting location that also yields a valid parse. Different splitters yield different simple recursive-descent parsers.

There is a trivial, brute-force splitter that suffices for proving correctness: simply return the list of all locations in the string, the list of all numbers between 0 and the length of the string. Because we construct a parser that terminates no matter what list it is given, and all valid splits are trivially in this list, this splitting "oracle" is enough to fill the oracle-shaped-hole in the correctness proofs. Thus, we can largely separate concerns about correctness and concerns about efficiency. In Chapter 3, we focus only on correctness; we set up the framework we use to achieve efficiency in Chapter 4, and we demonstrate the use of the framework in Chapters 5, 6 and 7.

Although this simple splitter is sufficient for proving the algorithm correct, it is horribly inefficient, running in time $\mathcal{O}(n!)$, where n is the length of the string. We synthesize more efficient splitters in later chapters; we believe that parameterizing the parser over a splitter gives us enough expressiveness to implement essentially all optimizations of interest, while yielding a sufficiently constrained design space to make proofs relatively straightforward. For example, to achieve linear parse time on the $\texttt{(ab)}^*$ grammar, we could have a splitter that, when trying to parse $\texttt{'c}_1\texttt{'} \cdot \texttt{'c}_2\texttt{'} \cdot \texttt{s}$ as $\texttt{ab(ab)}^*$, splits the string into $(\texttt{'c}_1\texttt{'}, \texttt{'c}_2\texttt{'}, \texttt{s})$; and when trying to parse `s` as ϵ, does not split the string at all.

Parameterizing over a splitting oracle allows us to largely separate correctness concerns from efficiency concerns.

Proving completeness—that our parser succeeds whenever there is a valid parse tree—is conceptually straightforward: trace the algorithm, showing that if the parser returns `false` at a given point, then assuming a corresponding parse tree exists yields a contradiction. The one wrinkle in this approach is that the procedure, the logic program, is not guaranteed to terminate.

1.1.1 Infinite Regress

Nontermination is a particularly pressing problem for us; we have programmed our parser in the proof assistant Coq [8], which only permits terminating programs. Coq is an interactive proof assistant; it includes a strongly typed functional programming language, called Gallina, in the tradition of OCaml and Haskell. Because Gallina programs do double duty as both functional programs and proofs, via the Curry-Howard isomorphism [9, 13], all programs are required to be provably terminating. However, naive recursive-descent parsers do not always terminate!

To see how such parsers can diverge, consider the following example. When defining the grammar $\texttt{(ab)}^*$, perhaps we give the following production rules:

$$\frac{s \in \epsilon}{s \in \texttt{(ab)}^*}\,(\epsilon) \qquad\qquad \frac{s_0 \in \texttt{'a'} \qquad s_1 \in \texttt{'b'}}{s_0 s_1 \in \texttt{(ab)}^*}\,(\texttt{"ab"})$$

$$\frac{s_0 \in \texttt{(ab)}^* \qquad s_1 \in \texttt{(ab)}^*}{s_0 s_1 \in \texttt{(ab)}^*}\,(\texttt{(ab)}^*\texttt{(ab)}^*)$$

Now, let us try to parse the string $\texttt{"ab"}$ as $\texttt{(ab)}^*$:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\texttt{""} \in \epsilon}}{\texttt{""} \in \texttt{(ab)}^*} \qquad \dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\texttt{""} \in \epsilon}}{\texttt{""} \in \texttt{(ab)}^*} \qquad \dfrac{\iddots}{\texttt{"ab"} \in \texttt{(ab)}^*}}{\texttt{""} \cdot \texttt{"ab"} \in \texttt{(ab)}^*}}{\texttt{"ab"} \in \texttt{(ab)}^*}}{}}{\texttt{""} \cdot \texttt{"ab"} \in \texttt{(ab)}^*}}{\texttt{"ab"} \in \texttt{(ab)}^*}$$

Thus, by making a poor choice in how we split strings and choose productions, we can quickly hit an infinite regress.

Assuming we have a function $\texttt{split} : \texttt{String} \to \texttt{[String} \times \texttt{String]}$ which is our splitting oracle, we may write out a potentially divergent parser specialized to this grammar.

```
any_parses : [String × String] → Bool
any_parses [] := false
any_parses (("a", "b") :: _) := true
any_parses ((s₁, s₂) :: rest_splits)
    := (parses s₁ && parses s₂) || any_parses rest_splits

parses : String → Bool
parses "" := true
parses str := any_parses (split str)
```

Here and throughout this thesis, we take the Haskell convention of using [T] to denote a list whose elements are of type T.

If split returns ("", "ab") as the first item in its list when given "ab", then parses will diverge in the way demonstrated above with the infinite derivation tree.

1.1.2 Aborting Early

To work around this wrinkle, we keep track of what nonterminals we have not yet tried to parse the current string as, and we abort early if we see a repeat. For our example grammar, since there is only one nonterminal, we only need to keep track of the current string. We refactor the above code to introduce a new parameter prev_s, recording the previous string we were parsing. We use s < prev_s to denote the test that s is strictly shorter than prev_s.

```
any_parses : String → [String × String] → Bool
any_parses _ [] := false
any_parses _ (("a", "b") :: _) := true
any_parses prev_s ((s₁, s₂) :: rest_splits)
   := (s₁ < prev_s && s₂ < prev_s
        && parses s₁ && parses s₂)
       || any_parses prev_s rest_splits

parses : String → Bool
parses "" := true
parses str := any_parses str (split str)
```

We can convince Coq that this definition is total via well-founded recursion on the length of the string passed to parses. For a more complicated grammar, we would need to use a well-founded relation that also included the number of nonterminals not yet tried for this string; we do this in Figure 3-3 in Subsection 3.3.2.

With this refactoring, however, completeness is no longer straightforward. We must show that aborting early does not eliminate good parse trees.

We devote the rest of Chapters 1 and 3 to describing an elegant approach to proving completeness. Ridge [22] carried out a proof about essentially the same algorithm in HOL4, a proof assistant that does not support dependent types. We instead refine our parser to have a more general polymorphic type signature that takes advantage of dependent types, supporting a proof strategy with a different kind of aesthetic appeal. Relational parametricity frees us from worrying about different control flows with different instantiations of the arguments: when care is taken to ensure that the execution of the algorithm does not depend on the values of the arguments, we are guaranteed that all instantiations succeed or fail together. Freed from this worry, we convince our parser to prove its own soundness and completeness by instantiating its arguments correctly.

1.1.3 Aside: Removing Left Recursion

To wrap up the description of our parsing algorithm, we call attention to a venerable technique for eliminating nontermination: preprocessing the grammar to remove left recursion. Intuitively, *left recursion* occurs whenever it is possible to encounter the same inference rule multiple times without removing any characters from the beginning of the string [18] The standard technique for removing left recursion involves ordering the inference rules; the idea is that, before the first terminal that shows up in any rule, only nonterminals earlier in the ordering should appear.

We choose a slightly different approach to eliminating nontermination. Since we will want to prove correctness properties of our parser, we have to verify the correctness of each step of our algorithm. Verifying the correctness of such a left-recursion-eliminating step is non-trivial, and, furthermore, preprocessing the grammar in such a fashion does not significantly simplify the evidence Coq requires to ensure termination. The approach we take is a kind of lazy variant of this; rather than preemptively eliminating the possibility of infinite chains of identical inference rules, we forbid such parses on-the-fly.

1.2 Standard Formal Definitions

Before proceeding, we pause to standardize on terminology and notation for context-free grammars and parsers. In service of clarity for some of our later explanations, we formalize grammars via natural-deduction inference rules, a slightly nonstandard choice.

1.2.1 Context-Free Grammar

A *context-free grammar* consists of *items*, which may be either *terminals* (characters) or *nonterminals*; plus a set of *productions*, each mapping a nonterminal to a sequence of items.

As in standard presentations, we restrict our attention to grammars where the set of nonterminals is finite. In our formalization, since a nonterminal is named by a string, we require that each grammar provide a list of “valid” nonterminals, each of which must only reference other valid nonterminals.

Example: `(ab)`*

The regular-expression grammar `(ab)`* has a single nonterminal `(ab)`*, which parses empty strings, as well as parsing strings which are an `'a'`, followed by a `'b'`, followed by a string which parses as the nonterminal `(ab)`*. In the standard, compact, notation for specifying context free grammars, we can write this as:

$$\texttt{(ab)}^* \ ::= \ \epsilon \mid \texttt{'a'}\ \texttt{'b'}\ \texttt{(ab)}^*$$

We can also present this grammar as a collection of inference rules, one for each production, and one for each terminal, in the grammar. This presentation is most useful for describing parse trees, so we will use it primarily in Section 1.3; we'll use the more compact representation for the larger grammars described in later chapters.

The inference rules of the regular-expression grammar `(ab)`* are:

Terminals:

$$\frac{}{\texttt{"a"} \in \texttt{'a'}} \qquad \frac{}{\texttt{"b"} \in \texttt{'b'}}$$

Productions and nonterminals:

$$\frac{s \in \epsilon}{s \in \texttt{(ab)}^*} \qquad \frac{}{\texttt{""} \in \epsilon}$$

$$\frac{s_0 \in \texttt{'a'} \qquad s_1 \in \texttt{'b'} \qquad s_2 \in \texttt{(ab)}^*}{s_0 s_1 s_2 \in \texttt{(ab)}^*}$$

1.2.2 Parse Trees

A string `s` *parses* as:

- a given terminal `ch` iff `s` = `'ch'`.
- a given sequence of items $\texttt{x}_i$ iff `s` splits into a sequence of strings $\texttt{s}_i$, each of which parses as the corresponding item $\texttt{x}_i$.
- a given nonterminal `nt` iff `s` parses as one of the item sequences that `nt` maps to under the set of productions.

We may define mutually inductive dependent type families of `ParseTreeOfs` and `ParseItemsTreeOfs` for a given grammar:

$$\texttt{ParseTreeOf} : \texttt{Item} \to \texttt{String} \to \textbf{Type}$$
$$\texttt{ParseItemsTreeOf} : \texttt{[Item]} \to \texttt{String} \to \textbf{Type}$$

For any terminal character ch, we have the constructor

$$(\texttt{'ch'}) : \texttt{ParseTreeOf 'ch' "ch"}$$

For any production rule mapping a nonterminal nt to a sequence of items its, and any string s, we have this constructor:

$$(\texttt{rule}) : \texttt{ParseItemsTreeOf its s} \to \texttt{ParseTreeOf nt s}$$

We have the following two constructors of ParseItemsTree. In writing the type of the latter constructor, we adopt a common space-saving convention where we assume that all free variables are quantified implicitly with dependent function (Π) types. We also write constructors in the form of schematic natural-deduction rules, since that notation will be convenient to use later on.

$$\frac{}{\texttt{""} \in \epsilon} : \texttt{ParseItemsTreeOf [] ""}$$

$$\begin{aligned}\frac{\texttt{s}_1 \in \texttt{it} \qquad \texttt{s}_2 \in \texttt{its}}{\texttt{s}_1\texttt{s}_2 \in \texttt{it} :: \texttt{its}} : &\ \texttt{ParseTreeOf it s}_1 \\ &\to \texttt{ParseItemsTreeOf its s}_2 \\ &\to \texttt{ParseItemsTreeOf (it :: its) s}_1\texttt{s}_2\end{aligned}$$

For brevity, we will sometimes use the notation $\overline{\texttt{s} \in \texttt{X}}$ to denote both ParseTreeOf X s and ParseItemsTreeOf X s, relying on context to disambiguate based on the type of X. Additionally, we will sometimes fold the constructors of ParseItemsTreeOf into the (rule) constructors of ParseTreeOf, to mimic the natural-deduction trees.

We also define a type of all parse trees, independent of the string and item, as this dependent-pair (Σ) type, using set-builder notation; we use ParseTree to denote the type

$$\{(\texttt{nt}, \texttt{s}) : \texttt{Nonterminal} \times \texttt{String} \mid \texttt{ParseTreeOf nt s}\}$$

1.3 Completeness and Soundness

Parsers come in a number of flavors. The simplest flavor is the *recognizer*, which simply says whether or not there exists a parse tree of a given string for a given nonterminal; it returns Booleans. There is also a richer flavor of parser that returns inhabitants of option ParseTree.

For any recognizer has_parse : Nonterminal $\to$ String $\to$ Bool, we may ask whether it is *sound*, meaning that when it returns true, there is always a parse tree; and *complete*, meaning that when there is a parse tree, it always returns true. We may express these properties as theorems (alternatively, dependently typed functions) with the following type signatures:

$$
\begin{aligned}
\texttt{has\_parse\_sound} : {} & (\texttt{nt} : \texttt{Nonterminal}) \to (\texttt{s} : \texttt{String}) \\
& \to \texttt{has\_parse nt s} = \texttt{true} \\
& \to \texttt{ParseTreeOf nt s} \\
\texttt{has\_parse\_complete} : {} & (\texttt{nt} : \texttt{Nonterminal}) \to (\texttt{s} : \texttt{String}) \\
& \to \texttt{ParseTreeOf nt s} \\
& \to \texttt{has\_parse nt s} = \texttt{true}
\end{aligned}
$$

For any parser

$$
\texttt{parse} : \texttt{Nonterminal} \;\to\; \texttt{String} \;\to\; \texttt{option ParseTree},
$$

we may also ask whether it is sound and complete, leading to theorems with the following type signatures, using $\texttt{p}_1$ to denote the first projection of $\texttt{p}$:

$$
\begin{aligned}
\texttt{parse\_sound} : {} & (\texttt{nt} : \texttt{Nonterminal}) \\
& \to (\texttt{s} : \texttt{String}) \\
& \to (\texttt{p} : \texttt{ParseTree}) \\
& \to \texttt{parse nt s} = \texttt{Some p} \\
& \to \texttt{p}_1 = (\texttt{nt}, \texttt{s}) \\
\texttt{parse\_complete} : {} & (\texttt{nt} : \texttt{Nonterminal}) \\
& \to (\texttt{s} : \texttt{String}) \\
& \to \texttt{ParseTreeOf nt s} \\
& \to \texttt{parse nt s} \neq \texttt{None}
\end{aligned}
$$

Since we are programming in Coq, this separation into code and proof actually makes for more awkward type assignments. We also have the option of folding the soundness and completeness conditions into the types of the code. For instance, the following type captures the idea of a sound and complete parser returning parse trees, using the type constructor + for disjoint union (i.e., sum or variant type):

$$
\begin{aligned}
\texttt{parse} : {} & (\texttt{nt} : \texttt{Nonterminal}) \\
& \to (\texttt{s} : \texttt{String}) \\
& \to \texttt{ParseTreeOf nt s} + (\texttt{ParseTreeOf nt s} \to \bot)
\end{aligned}
$$

That is, given a nonterminal and a string, `parse` either returns a valid parse tree, or returns a *proof* that the existence of any parse tree is *contradictory* (i.e., implies $\bot$, the empty type). Our implementation follows this dependently typed style. Our main initial goal in the project was to arrive at a `parse` function of just this type, generic in an arbitrary choice of context-free grammar, implemented and proven correct in an elegant way.

Chapter 2

Related Work and Other Approaches to Parsing

Stepping back a bit, we describe how our approach to parsing relates to existing work.

2.1 Coq

As stated in Subsection 1.1.1, we define our parser and prove its correctness in the proof assistant Coq [8]. Like other proof assistants utilizing dependent type theory, Coq takes advantage of the Curry-Howard isomorphism [9, 13] to allow proofs to be written as functional programs; dependent types allow universal and existential quantification. Coq natively permits only structural recursion, where recursive function calls may be invoked only on direct structural subterms of a given, specified argument. The standard library defines combinators for turning well-founded recursion into structural recursion, which can be used to define essentially all recursive functions which provably halt in all cases (which is a class containing, as it turns out, essentially all algorithms of interest). Coq's mathematical language, Gallina, implements Martin-Löf's dependent lambda calculus. Coq has a separate tactic language, called $\mathcal{L}_{tac}$ [10], which allows imperative construction of proof objects, and functions, by forwards and backwards reasoning.

2.2 Recursive-Descent Parsing

The most conceptually straightforward approaches to parsing fall into the class called recursive-descent parsing, where, to parse a string s as a given production p, you attempt to parse various parts of s as each of the items in the list p. The control flow of the code mirrors the structure of the grammar, as well as the structure of the eventual parse tree, descending down the branches of the parse tree, recursively calling itself at each step. The algorithm we have described in Chapter 1 seems to fall out almost trivially from the inductive description of parse trees; we come back to this in Section 8.1 when we briefly sketch how it should be possible to generalize this algorithm to other inductive type families.

2.2.1 Parser Combinators

A popular approach to implementing recursive-descent parsing, called *combinator parsing* [14], involves writing a small set of typed combinators, or higher-order functions, which are then applied to each other in various combinations to write a parser that mimics closely the structure of the grammar.

Essentially, parsers defined via parser combinators answer the question "what prefixes of a given string can be parsed as a given item?" Each function returns a list of postfixes of the string it is passed, indicating all of the strings that might remain for the other items in a given rule.

Basic Combinators

We now define the basic combinators. In the simplest form, each combinator takes in a string, and returns a list of strings (the postfixes); we can define the type

$$\texttt{parser} \coloneqq \texttt{String} \to \texttt{[String]}.$$

We can define the empty-string parser, as well as the parser for a nonterminal with no production rules, which always fails:

$$\begin{aligned}&\epsilon\ :\ \texttt{parser}\\&\epsilon\,\texttt{str} \coloneqq \texttt{[str]}\\[1em]&\texttt{fail}\ :\ \texttt{parser}\\&\texttt{fail \_} \coloneqq \texttt{[]}\end{aligned}$$

Failure is indicated by returning the empty list; success at parsing the entire string is indicated by returning a list containing the empty string.

The parser for a given terminal fails if the string does not start with that character, and returns all but the first character if it does:

$$\begin{aligned}&\texttt{terminal}\ :\ \texttt{Char} \to \texttt{parser}\\&\texttt{terminal ch}\ (\texttt{ch} :: \texttt{str}) \coloneqq \texttt{[str]}\\&\texttt{terminal \_}\ \ \texttt{\_} \coloneqq \texttt{[]}\end{aligned}$$

We now define combinators for sequencing and alternatives:

$$(\texttt{>>>})\ :\ \texttt{parser} \to \texttt{parser} \to \texttt{parser}$$

$$(\texttt{p}_0 \;\texttt{>>>}\; \texttt{p}_1)\;\texttt{str} \coloneqq \texttt{flat\_map}\;\texttt{p}_1\;(\texttt{p}_0\;\texttt{str})$$

$$\begin{aligned}(\texttt{|||}) &: \texttt{parser} \to \texttt{parser} \to \texttt{parser}\\ (\texttt{p}_0 \;\texttt{|||}\; \texttt{p}_1)\;\texttt{str} &\coloneqq \texttt{p}_0\;\texttt{str}\;\texttt{++}\;\texttt{p}_1\;\texttt{str}\end{aligned}$$

where `++` is list concatenation, and `flat_map`, which concatenates the lists returned by mapping its first argument over each of the elements in its second argument, has type `(A → [B]) → [A] → [B]`.

An Example

We can now easily define a parser for the grammar $\texttt{(ab)}^*$:

$$\begin{aligned}\texttt{parse\_(ab)}^* &: \texttt{parser}\\ \texttt{parse\_(ab)}^* &\coloneqq (\texttt{terminal 'a'} \;\texttt{>>>}\; \texttt{terminal 'b'} \;\texttt{>>>}\; \texttt{parse\_(ab)}^*) \;\;\texttt{|||}\;\; \epsilon\end{aligned}$$

Note that, by putting ϵ last, we ensure that this parser returns the list in order of longest parse (shortest postfix) to shortest parse (longest postfix).

Semantic Actions

Frequently, programmers want parsers to not just say whether or not a given string, or prefix of a string, can be parsed, but to also build a parse tree, or perform some other computation or construction on the structure of the string. A common way to accomplish this is with *semantic actions*: Associate to each production a function which, when given values associated to each of the nonterminals in its sequence, computes a value to associate to the given nonterminal. By calling these functions at each node of the parse tree, passing the function at each node the values returned by its descendants, we can compute a value associated to a string as we parse it. For example, we might annotate a simple expression grammar, to compute the numerical value associated with a string expression, like this:

$$\begin{aligned} e ::= {}& n && \{\texttt{int\_of\_string}(n)\}\\ \mid {}& e_1 \;\texttt{"+"}\; e_2 && \{e_1 + e_2\}\\ \mid {}& \texttt{"("}\; e \;\texttt{")"} && \{e\}\end{aligned}$$

Parser combinators can be easily extended to return a list not just of postfixes, but of pairs of a value and a postfix. The `parser` type can be parameterized over the type of the value returned. The alternative combinator would return a disjoint union, and the sequencing combinator would return a pair of the two values returned by its inputs. The terminal parser would return the single character it parsed, and the empty string parser would return an element of the singleton type. Each rule for a nonterminal could then be wrapped with a combinator which applies the semantic action to the relevant values. A more detailed explanation can be found in [14]. We describe in Section 3.4 how our parser can easily accommodate semantic actions.

Proving Correctness and Dealing with Nontermination

Although parser combinators are straightforward, it is easy to make them loop forever. It is well-known that parsers defined naively using parser combinators don't handle grammars with *left recursion*, where the first item in a given production rule is the nonterminal currently being defined. For example, if we have the nonterminal `expr` ::= `number` | `expr '+' expr`, then the parser for `expr '+' expr` will call the parser for `expr`, which will call the parse for `expr '+' expr`, which will quickly loop forever.

The algorithm we presented in Subsection 1.1.2 is essentially the same as the algorithm Ridge presents in [22] to deal with this problem. By wrapping the calls to the parsers, in each combinator, with a function that prunes duplicative calls, Ridge provides a way to ensure that parsers terminate. Also included in [22] are proofs in HOL4 that such wrapped parsers are both sound (and therefore terminating) and complete. Furthermore, Ridge's parser has worst-case $O(n^5)$ running time in the input-string length.

2.2.2 Parsing with Derivatives

Might, Darais, and Spiewak describe an elegant method for recursive-descent parsing in [17], based on Brzozowski's derivatives [5], which might be considered a conceptual dual to standard combinator parsing. Rather than returning a list of possible string remnants, constructed by recursing down the structure of the grammar, we can iterate down the characters of a string, computing an updated language, or grammar, at each point.

The *language* defined by a grammar is the set of strings accepted by that grammar. Here we describe the mathematical ideas behind parsing with derivatives. Might et al. take a slightly different approach to ensure termination; where we will describe the mathematical operations on languages, they define these operations on a structural representation of the language, akin to an inductive definition of the grammar.

Much as we defined parser combinators for the elementary operations of a grammar (ϵ, terminals, sequencing, and alternatives), we can define similar combinators for defining a (lazy, or coinductive) language for a grammar. Defining the type `language` to be a set (or just a coinductive list) of strings, we have:

$$
\begin{aligned}
&\epsilon\ :\ \texttt{language}\\
&\epsilon\ \coloneqq\ \{\texttt{""}\}\\[1em]
&\texttt{terminal}\ :\ \texttt{Char}\ \rightarrow\ \texttt{language}\\
&\texttt{terminal ch}\ \coloneqq\ \{\texttt{ch}\}\\[1em]
&(\texttt{>>>})\ :\ \texttt{language}\ \rightarrow\ \texttt{language}\ \rightarrow\ \texttt{language}
\end{aligned}
$$

$$\mathcal{L}_0 \texttt{ >>> } \mathcal{L}_1 \coloneqq \{\ \mathtt{s}_0\mathtt{s}_1 \mid \mathtt{s}_0 \in \mathcal{L}_0 \text{ and } \mathtt{s}_1 \in \mathcal{L}_1\}$$

$$\begin{aligned}&(\texttt{|||}) : \texttt{language} \to \texttt{language} \to \texttt{language}\\ &\mathcal{L}_0 \texttt{ ||| } \mathcal{L}_1 \coloneqq \mathcal{L}_0 \cup \mathcal{L}_1\end{aligned}$$

The essential operations for computing derivatives are *filtering* and *chopping*. To *filter* a language $\mathcal{L}$ by a character c is to take the subset of strings in $\mathcal{L}$ which start with c. To chop a language $\mathcal{L}$ is to remove the first character from every string. The derivative $D_c(\mathcal{L})$ with respect to c of a language $\mathcal{L}$ is then the language $\mathcal{L}$, filtered by c and chopped:

$$\begin{aligned}&\mathtt{D_c} : \texttt{language} \to \texttt{language}\\ &\mathtt{D_c}\ \mathcal{L} \coloneqq \bigcup_{(\texttt{c::str})\in\mathcal{L}} \{\texttt{str}\}\end{aligned}$$

We can then define a `has_parse` proposition by taking successive derivatives:

$$\begin{aligned}&\texttt{has\_parse} : \texttt{language} \to \texttt{String} \to \textbf{\texttt{Prop}}\\ &\texttt{has\_parse}\ \mathcal{L}\ \texttt{""} \coloneqq \texttt{""} \in \mathcal{L}\\ &\texttt{has\_parse}\ \mathcal{L}\ (\texttt{ch} :: \texttt{str}) \coloneqq \texttt{has\_parse}\ (\mathtt{D_{ch}}\ \mathcal{L})\ \texttt{str}\end{aligned}$$

To ensure termination and good performance, Might et al. define the derivative operation on the structure of the grammar, rather than defining combinators that turn a grammar into a language, and furthermore take advantage of laziness and memoization. After adding code to prune the resulting language of useless content, they argue that the cost of parsing with derivatives is reasonable.

Formal Verification

Almeida et al. formally verify, in Coq, finite automata for parsing the fragment of derivative-based parsing which applies to regular expressions [1]. This fragment dates back to Brzozowski's original presentation of derivatives [5].

2.3 Other Approaches to Parsing

Recursive-descent parsing is not the only strategy for parsing.

Top-Down Parsers: LL(k) Recursive-descent parsing is a flavor of so-called "top-down" parsing; at each point in the algorithm, we know which nonterminal we are parsing the string as. We thus build the parse tree from the top down, filling in more portions of the parse tree by picking which rule of a fixed nonterminal we should use.

Some context-free grammars have linear-time recursive-descent parsers that only require k tokens after the current one being considered to decide which rule to apply; these grammars are called LL(k) grammars. More recently, arbitrary context-free grammars can be handled with Generalized LL parsers [23], or with ALL(*) parsers [21], which are based on arbitrary look-ahead using regular expressions.

Bottom-Up Parsers: LR Bottom-up parsers, of which LR parsers [6] are one of the most well-known flavors, instead associate the parts of the string which have already been parsed to complete parse trees. For example, consider the grammar with two nonterminals, `ab` ::= `'a' 'b'`, and `(ab)`* ::= ϵ `| ab (ab)`*. When parsing `"abab"` as `(ab)`*, an LR parser would parse `'a'` as the terminal `'a'`, parse `'b'` as the terminal `'b'`, and then reduce those two parse trees into a single parse tree for `ab`. It would then parse `'a'` as `'a'`, parse `'b'` as `'b'`, and then reduce those into the parse tree for `ab`; we now have two parse trees for `ab`. Noticing that there are no characters remaining, the parser would reduce the latter `ab` into a parse tree for `(ab)`* and then combine that with the earlier `ab` parse tree to get a parse of the entire string as `(ab)`*.

LR parsers originated in the days when computers has much more stringent constraints on memory and processing power, and they apply only to strict subsets of context-free grammars; correctness and complexity guarantees rely on being able to uniquely determine what rule to apply based on a fixed lookahead.

More recently, Generalize LR (GLR) parsers have been devised which can handle all context-free grammars [24].

Parsing expression grammars (PEGs) Ford proposes an alternative to context-free grammars, called parsing expression grammars [12], which can always be deterministically parsed in linear time. The basic idea is to incorporate some of the features of regular expressions directly into the grammar specification, and to drop the ability to have ambiguous alternatives; PEGs instead have prioritized alternatives.

2.4 Related Work on Verifying Parsers

In addition to the work on verifying derivative-based parsing of regular expressions [1], a few other past projects have verified parsers with proof assistants, applying to SLR [2] and LR(1) [15] parsers. Several projects have used proof assistants to apply verified parsers within larger programming-language tools. RockSalt [19] does run-time memory-safety enforcement for x86 binaries, relying on a verified machine-code parser that applies derivative-based parsing for regular expressions. The verified Jitawa [20] and CakeML [16] language implementations include verified parsers, handling Lisp and ML languages, respectively.

2.5 What's New and What's Old

The goal of this project is to demonstrate a new approach to generating parsers: incrementally building efficient parsers by refinement.

We begin with naive recursive-descent parsing. We ensure termination via memoization, a la [22]. We parameterize the parser on a "splitting oracle", which describes how to recurse (Section 1.1). As far as we can tell, the idea of factoring the algorithmic complexity like this is new.

We use the Coq library Fiat [11] to incrementally build efficient parsers by refinement; we describe Fiat starting in Chapter 4.

Additionally, we take a digression in Chapter 3 to describe how our parser can be used to prove its own completeness; the idea of reusing the parsing algorithm to generate proofs, parsing parse trees rather than strings, is not found in the literature, to the author's knowledge.

Chapter 3

Completeness, Soundness, and Parsing Parse Trees

3.1 Proving Completeness: Conceptual Approach

Recall from Subsection 1.1.2 that the essential difficulty with proving completeness is dealing with the cases where our parser aborts early; we must show that doing so does not eliminate good parse trees.

The key is to define an intermediate type, that of "minimal parse trees." A "minimal" parse tree is simply a parse tree in which the same (string, nonterminal) pair does not appear more than once in any path of the tree. Defining this type allows us to split the completeness problem in two; we can show separately that every parse tree gives rise to a minimal parse tree, and that having a minimal parse tree in hand implies that our parser succeeds (returns `true` or `Some _`).

Our dependently typed parsing algorithm subsumes the soundness theorem, the minimization of parse trees, and the proof that having a minimal parse tree implies that our parser succeeds. We write one parametrically polymorphic parsing function that supports all three modes, plus the several different sorts of parsers (recognizers, generating parse trees, running semantic actions). That level of genericity requires us to be flexible in which type represents "strings," or inputs to parsers. We introduce a parameter that is often just the normal `String` type, but which needs to be instantiated as the type of *parse trees themselves* to get a proof of parse tree minimizability. That is, we "parse" parse trees to minimize them, reusing the same logic that works for the normal parsing problem.

Before presenting our algorithm's interface, we will formally define and explain minimal parse trees, which will provide motivation for the type signatures of our parser's arguments.

3.2 Minimal Parse Trees: Formal Definition

In order to make tractable the second half of the completeness theorem, that having a minimal parse tree implies that parsing succeeds, it is essential to make the inductive structure of minimal parse trees mimic precisely the structure of the parsing algorithm. A minimal parse tree thus might better be thought of as a parallel trace of parser execution.

As in Subsection 1.2.2, we define mutually inductive type families of `MinParseTreeOf`s and `MinItemsTreeOf`s for a given grammar. Because our parser proceeds by well-founded recursion on the length of the string and the list of nonterminals not yet attempted for that string, we must include both of these in the types. Let us call the initial list of all nonterminals $\texttt{unseen}_0$.

$$\begin{aligned}\texttt{MinParseTreeOf} &: \texttt{String} \to \texttt{[Nonterminal]} \\ &\to \texttt{Item} \to \texttt{String} \to \textbf{Type} \\ \texttt{MinItemsTreeOf} &: \texttt{String} \to \texttt{[Nonterminal]} \\ &\to \texttt{[Item]} \to \texttt{String} \to \textbf{Type}\end{aligned}$$

Much as in the case of parse trees, for any terminal character `ch`, any string $\texttt{s}_0$, and any list of nonterminals `unseen`, we have the constructor

$$\texttt{min\_parse}_{\texttt{'ch'}} : \texttt{MinParseTreeOf}\ \ \texttt{s}_0\ \ \texttt{unseen}\ \ \texttt{'ch'}\ \ \texttt{"ch"}$$

For any production `rule` mapping a nonterminal `nt` to a sequence of items `its`, any string $\texttt{s}_0$, any list of nonterminals `unseen`, and any string `s`, we have two constructors, corresponding to the two ways of progressing with respect to the well-founded relation. We have the following, where we interpret the $<$ relation on strings in terms of lengths.

$$\begin{aligned}(\texttt{rule})_< &: \texttt{s} < \texttt{s}_0 \\ &\to \texttt{MinItemsTreeOf}\ \ \texttt{s}\ \ \texttt{unseen}_0\ \ \texttt{its}\ \ \texttt{s} \\ &\to \texttt{MinParseTreeOf}\ \ \texttt{s}_0\ \ \texttt{unseen}\ \ \texttt{nt}\ \ \texttt{s} \\ (\texttt{rule})_= &: \texttt{s} = \texttt{s}_0 \\ &\to \texttt{nt} \in \texttt{unseen} \\ &\to \texttt{MinItemsTreeOf}\ \ \texttt{s}_0\ \ (\texttt{unseen} - \{\texttt{nt}\})\ \ \texttt{its}\ \ \texttt{s} \\ &\to \texttt{MinParseTreeOf}\ \ \texttt{s}_0\ \ \texttt{unseen}\ \ \texttt{nt}\ \ \texttt{s}\end{aligned}$$

In the first case, the length of the string has decreased, so we may reset the list of not-yet-seen nonterminals, as long as we reset the base of well-founded recursion $\texttt{s}_0$ at the same time. In the second case, the length of the string has not decreased, so we require that we have not yet seen this nonterminal, and we then remove it from the list of not-yet-seen nonterminals.

Finally, for any string $\mathtt{s}_0$ and any list of nonterminals `unseen`, we have the following two constructors of `MinItemsTreeOf`.

$$
\begin{aligned}
\mathtt{min\_parse}_{[]} &: \mathtt{MinItemsTreeOf}\ \ \mathtt{s}_0\ \ \mathtt{unseen}\ \ []\ \ \texttt{""} \\
\mathtt{min\_parse}_{::} &: \mathtt{s}_1\mathtt{s}_2 \le \mathtt{s}_0 \\
&\to \mathtt{MinParseTreeOf}\ \ \mathtt{s}_0\ \ \mathtt{unseen}\ \ \mathtt{it}\ \ \mathtt{s}_1 \\
&\to \mathtt{MinItemsTreeOf}\ \ \mathtt{s}_0\ \ \mathtt{unseen}\ \ \mathtt{its}\ \ \mathtt{s}_2 \\
&\to \mathtt{MinItemsTreeOf}\ \ \mathtt{s}_0\ \ \mathtt{unseen}\ \ (\mathtt{it} :: \mathtt{its})\ \ \mathtt{s}_1\mathtt{s}_2
\end{aligned}
$$

The requirement that $\mathtt{s}_1\mathtt{s}_2 \le \mathtt{s}_0$ in the second case ensures that we are only making well-founded recursive calls.

Once again, for brevity, we will sometimes use the notation $\overline{\mathtt{s} \in \mathtt{X}}^{<(\mathtt{s}_0,\mathtt{v})}$ to denote both `MinParseTreeOf` $\mathtt{s}_0$ `v X s` and `MinItemsTreeOf` $\mathtt{s}_0$ `v X s`, relying on context to disambiguate based on the type of `X`. Additionally, we will sometimes fold the constructors of `MinItemsTreeOf` into the two (`rule`) constructors of `MinParseTreeOf`, to mimic the natural-deduction trees.

3.3 Parser Interface

Roughly speaking, we read the interface of our general parser off from the types of the constructors for minimal parse trees. Every constructor leads to one parameter passed to the parser, much as one derives the types of general "fold" functions for arbitrary inductive datatypes. For instance, lists have constructors `nil` and `cons`, so a fold function for lists has arguments corresponding to `nil` (initial accumulator) and `cons` (step function). The situation for the type of our parser is similar, though we need parallel success (managed to parse the string) and failure (could prove that no parse is possible) parameters for each constructor of minimal parse trees.

The type signatures in the interface are presented in Figure 3-1. We explain each type one by one, presenting various instantiations as examples. Note that the interface we actually implemented is also parameterized over a type of `String`s, which we will instantiate with parse trees later in this chapter. The interface we present here fixes `String`, for conciseness.

Since we want to be able to specialize our parser to return either `option ParseTree` or `Bool`, we want to be able to reuse our soundness and completeness proofs for both. Our strategy for generalization is to parameterize on dependent type families for "success" and "failure", so we can use relational parametricity to ensure that all instantiations of the parser succeed or fail together. The parser has the rough type signature

$$\mathtt{parse} : \mathtt{Nonterminal}\ \to\ \mathtt{String}\ \to\ \mathtt{T}_{\mathtt{success}} + \mathtt{T}_{\mathtt{failure}}.$$

We use `ParseQuery` to denote the type of all propositions like “$\texttt{"a"} \in \texttt{'a'}$”; a query consists of a string and a grammar rule the string might be parsed into. We use the same notation for `ParseQuery` and `ParseTree` inhabitants. All `*_success` and `*_failure` type signatures are implicitly parameterized over a string $\texttt{s}_0$ and a list of nonterminals `unseen`. We assume we are given $\texttt{unseen}_0 : \texttt{[Nonterminal]}$.

$$
\begin{aligned}
\texttt{T}_{\texttt{success}},\ \texttt{T}_{\texttt{failure}} &: \texttt{String} \to \texttt{[Nonterminal]} \to \texttt{ParseQuery} \to \textbf{Type} \\
\texttt{split} &: \texttt{String} \to \texttt{[Nonterminal]} \to \texttt{ParseQuery} \to [\mathbb{N}] \\
\\
\texttt{terminal\_success} &: (\texttt{ch} : \texttt{Char}) \to \texttt{T}_{\texttt{success}}\ \ \texttt{s}_0\ \ \texttt{unseen}\ \ (\overline{\texttt{"ch"} \in \texttt{'ch'}}) \\
\texttt{terminal\_failure} &: \forall\ \texttt{ch}\ \texttt{s},\ \texttt{s} \neq \texttt{"ch"} \to \texttt{T}_{\texttt{failure}}\ \ \texttt{s}_0\ \ \texttt{unseen}\ \ (\overline{\texttt{s} \in \texttt{'ch'}}) \\
\texttt{nil\_success} &: \texttt{T}_{\texttt{success}}\ \ \texttt{s}_0\ \ \texttt{unseen}\ \ (\overline{\texttt{""} \in \epsilon}) \\
\texttt{nil\_failure} &: (\texttt{s} : \texttt{String}) \to \texttt{s} \neq \texttt{""} \to \texttt{T}_{\texttt{failure}}\ \ \texttt{s}_0\ \ \texttt{unseen}\ \ (\overline{\texttt{s} \in \epsilon}) \\
\\
\texttt{cons\_success} &: (\texttt{it} : \texttt{Item}) \to (\texttt{its} : \texttt{[Item]}) \to (\texttt{s}_1\ \texttt{s}_2 : \texttt{String}) \\
&\to \texttt{s}_1\texttt{s}_2 \leq \texttt{s}_0 \\
&\to \texttt{T}_{\texttt{success}}\ \ \texttt{s}_0\ \ \texttt{unseen}\ \ (\overline{\texttt{s}_1 \in \texttt{it}}) \\
&\to \texttt{T}_{\texttt{success}}\ \ \texttt{s}_0\ \ \texttt{unseen}\ \ (\overline{\texttt{s}_2 \in \texttt{its}}) \\
&\to \texttt{T}_{\texttt{success}}\ \ \texttt{s}_0\ \ \texttt{unseen}\ \ (\overline{\texttt{s}_1\texttt{s}_2 \in \texttt{it} :: \texttt{its}}) \\
\texttt{cons\_failure} &: (\texttt{it} : \texttt{Item}) \to (\texttt{its} : \texttt{[Item]}) \to (\texttt{s} : \texttt{String}) \\
&\to \texttt{s} \leq \texttt{s}_0 \\
&\to \Big(\forall\ (\texttt{s}_1, \texttt{s}_2) \in \texttt{split}\ \ \texttt{s}_0\ \ \texttt{unseen}\ \ (\overline{\texttt{s} \in \texttt{it} :: \texttt{its}}), \\
&\qquad \texttt{T}_{\texttt{failure}}\ \ \texttt{s}_0\ \ \texttt{unseen}\ \ (\overline{\texttt{s}_1 \in \texttt{it}}) \\
&\qquad + \texttt{T}_{\texttt{failure}}\ \ \texttt{s}_0\ \ \texttt{unseen}\ \ (\overline{\texttt{s}_2 \in \texttt{its}})\Big) \\
&\to \texttt{T}_{\texttt{failure}}\ \ \texttt{s}_0\ \ \texttt{unseen}\ \ (\overline{\texttt{s} \in \texttt{it} :: \texttt{its}}) \\
\\
\texttt{production\_success}_{<} &: (\texttt{its} : \texttt{[Item]}) \to (\texttt{nt} : \texttt{Nonterminal}) \to (\texttt{s} : \texttt{String}) \\
&\to \texttt{s} < \texttt{s}_0 \\
&\to (\texttt{p} : \text{a production mapping } \texttt{nt} \text{ to } \texttt{its}) \\
&\to \texttt{T}_{\texttt{success}}\ \ \texttt{s}\ \ \texttt{unseen}_0\ \ (\overline{\texttt{s} \in \texttt{its}}) \\
&\to \texttt{T}_{\texttt{success}}\ \ \texttt{s}_0\ \ \texttt{unseen}\ \ (\overline{\texttt{s} \in \texttt{nt}}) \\
\texttt{production\_success}_{=} &: (\texttt{its} : \texttt{[Item]}) \to (\texttt{nt} : \texttt{Nonterminal}) \to (\texttt{s} : \texttt{String}) \\
&\to \texttt{nt} \in \texttt{unseen} \\
&\to (\texttt{p} : \text{a production mapping } \texttt{nt} \text{ to } \texttt{its}) \\
&\to \texttt{T}_{\texttt{success}}\ \ \texttt{s}_0\ \ (\texttt{unseen} - \{\texttt{nt}\})\ \ (\overline{\texttt{s} \in \texttt{its}}) \\
&\to \texttt{T}_{\texttt{success}}\ \ \texttt{s}_0\ \ \texttt{unseen}\ \ (\overline{\texttt{s} \in \texttt{nt}})
\end{aligned}
$$

Figure 3-1: The dependently typed interface of our parser, Part 1 of 2

$$
\begin{aligned}
\texttt{production\_failure}_{<} : {} & (\texttt{nt} : \texttt{Nonterminal}) \to (\texttt{s} : \texttt{String}) \\
& \to \texttt{s} < \texttt{s}_0 \\
& \to \Big(\forall\ (\texttt{its} : \texttt{[Item]})\ (\texttt{p} : \text{a production mapping } \texttt{nt} \text{ to } \texttt{its}), \\
& \qquad \texttt{T}_{\texttt{failure}}\ \ \texttt{s}\ \ \texttt{unseen}_0\ \ (\overline{\texttt{s} \in \texttt{its}})\Big) \\
& \to \texttt{T}_{\texttt{failure}}\ \ \texttt{s}_0\ \ \texttt{unseen}\ \ (\overline{\texttt{s} \in \texttt{nt}}) \\
\texttt{production\_failure}_{=} : {} & (\texttt{nt} : \texttt{Nonterminal}) \to (\texttt{s} : \texttt{String}) \\
& \to \texttt{s} = \texttt{s}_0 \\
& \to \Big(\forall\ (\texttt{its} : \texttt{[Item]})\ (\texttt{p} : \text{a production mapping } \texttt{nt} \text{ to } \texttt{its}), \\
& \qquad \texttt{T}_{\texttt{failure}}\ \ \texttt{s}_0\ \ (\texttt{unseen} - \{\texttt{nt}\})\ \ (\overline{\texttt{s} \in \texttt{its}})\Big) \\
& \to \texttt{T}_{\texttt{failure}}\ \ \texttt{s}_0\ \ \texttt{unseen}\ \ (\overline{\texttt{s} \in \texttt{nt}}) \\
\texttt{production\_failure}_{\notin} : {} & (\texttt{nt} : \texttt{Nonterminal}) \to (\texttt{s} : \texttt{String}) \\
& \to \texttt{s} = \texttt{s}_0 \\
& \to \texttt{nt} \notin \texttt{unseen} \\
& \to \texttt{T}_{\texttt{failure}}\ \ \texttt{s}_0\ \ \texttt{unseen}\ \ (\overline{\texttt{s} \in \texttt{nt}})
\end{aligned}
$$

Figure 3-2: The dependently typed interface of our parser, Part 2 of 2

To instantiate the parser as a Boolean recognizer, we instantiate everything trivially; we use the fact that $\top + \top \cong \texttt{Bool}$, where $\top$ is the singleton type inhabited by `()`. Just to show how trivial everything is, here is a precise instantiation of the parser, still parameterized over the initial list of nonterminals and the splitter, where $\top$ is the one constructor of the one-element type $\top$:

$$
\begin{aligned}
& \texttt{T}_{\texttt{success}}\ \_\ \_\ \_ \coloneqq \top \\
& \texttt{T}_{\texttt{failure}}\ \_\ \_\ \_ \coloneqq \top \\
\\
& \texttt{terminal\_success}\ \_\ \_\ \_ \coloneqq () \\
& \texttt{terminal\_failure}\ \_\ \_\ \_\ \_\ \_ \coloneqq () \\
& \texttt{nil\_success}\ \_\ \_ \coloneqq () \\
& \texttt{nil\_failure}\ \_\ \_\ \_\ \_ \coloneqq () \\
& \texttt{cons\_success}\ \_\ \_\ \_\ \_\ \_\ \_\ \_\ \_\ \_\ \_ \coloneqq () \\
& \texttt{cons\_failure}\ \_\ \_\ \_\ \_\ \_\ \_\ \_\ \_ \coloneqq () \\
& \texttt{production\_success}_{<}\ \_\ \_\ \_\ \_\ \_\ \_\ \_\ \_\ \_ \coloneqq () \\
& \texttt{production\_success}_{=}\ \_\ \_\ \_\ \_\ \_\ \_\ \_\ \_\ \_\ \_ \coloneqq () \\
& \texttt{production\_failure}_{<}\ \_\ \_\ \_\ \_\ \_\ \_\ \_ \coloneqq () \\
& \texttt{production\_failure}_{=}\ \_\ \_\ \_\ \_\ \_\ \_\ \_ \coloneqq () \\
& \texttt{production\_failure}_{\notin}\ \_\ \_\ \_\ \_\ \_\ \_\ \_ \coloneqq ()
\end{aligned}
$$

To instantiate our parser so that it returns `option ParseTree` (rather, the dependently typed flavor, `ParseTreeOf`), we take advantage of the isomorphism $T + \top \cong$ `option` T. We show only the `success` instantiations, as the `failure` ones are identical with the Boolean recognizer. For readability of the code, we write schematic natural-deduction proof trees inline.

$$\mathtt{T_{success}}\ \_\ \_\ \ (\overline{\mathtt{s} \in \mathtt{X}}) \coloneqq \overline{\mathtt{s} \in \mathtt{X}}$$

$$\mathtt{terminal\_success}\ \_\ \_\ \mathtt{ch} \coloneqq (\mathtt{'ch'})$$

$$\mathtt{nil\_success}\ \_\ \_ \coloneqq \overline{\mathtt{""} \in \epsilon}$$

$$\mathtt{cons\_success}\ \_\ \_\ \mathtt{it}\ \mathtt{its}\ \mathtt{s_1}\ \mathtt{s_2}\ \_\ \mathtt{d_1}\ \mathtt{d_2} \coloneqq \frac{\dfrac{\mathtt{d_1}}{\mathtt{s_1}\in\mathtt{it}} \qquad \dfrac{\mathtt{d_2}}{\mathtt{s_2}\in\mathtt{its}}}{\mathtt{s_1}\mathtt{s_2} \in \mathtt{it} :: \mathtt{its}}$$

$$\mathtt{production\_success}_{<}\ \_\ \_\ \mathtt{it}\ \mathtt{nt}\ \mathtt{s}\ \_\ \mathtt{p}\ \mathtt{d} \coloneqq \frac{\dfrac{\mathtt{d}}{\mathtt{s}\in\mathtt{its}}}{\mathtt{s} \in \mathtt{nt}}\ (\mathtt{p})$$

$$\mathtt{production\_success}_{=}\ \_\ \_\ \mathtt{it}\ \mathtt{nt}\ \mathtt{s}\ \_\ \mathtt{p}\ \mathtt{d} \coloneqq \frac{\dfrac{\mathtt{d}}{\mathtt{s}\in\mathtt{its}}}{\mathtt{s} \in \mathtt{nt}}\ (\mathtt{p})$$

What remains is instantiating the parser in such a way that proving completeness is trivial. The simpler of our two tasks is to show that when the parser fails, no minimal parse tree exists. Hence we instantiate the types as follows, where $\perp$ is the empty type (equivalently, the false proposition).

$$\mathtt{T_{success}}\ \_\ \_\ \_ \coloneqq \top$$

$$\mathtt{T_{failure}}\ \mathtt{s_0}\ \mathtt{unseen}\ \ (\overline{\mathtt{s} \in \mathtt{X}}) \coloneqq \left(\overline{\mathtt{s} \in \mathtt{X}}^{\,<(\mathtt{s_0},\mathtt{unseen})}\right) \to \perp$$

Using $\lightning$ to denote a (possibly automated) proof deriving a contradiction, we can unenlighteningly instantiate the arguments as

$$\begin{aligned}
&\mathtt{terminal_success}\ _\ _\ _ \coloneqq ()\\
&\mathtt{terminal_failure}\ _\ _\ _\ _\ _ \coloneqq \lightning\\
&\mathtt{nil_success}\ _\ _ \coloneqq ()\\
&\mathtt{nil_failure}\ _\ _\ _\ _ \coloneqq \lightning\\
&\mathtt{cons_success}\ _\ _\ _\ _\ _\ _\ _\ _\ _\ _ \coloneqq ()\\
&\mathtt{cons_failure}\ _\ _\ _\ _\ _\ _\ _ \coloneqq \lightning\\
&\mathtt{production_success}_{<}\ _\ _\ _\ _\ _\ _\ _\ _ \coloneqq ()\\
&\mathtt{production_success}_{=}\ _\ _\ _\ _\ _\ _\ _\ _\ _ \coloneqq ()\\
&\mathtt{production_failure}_{<}\ _\ _\ _\ _\ _\ _ \coloneqq \lightning\\
&\mathtt{production_failure}_{=}\ _\ _\ _\ _\ _\ _ \coloneqq \lightning\\
&\mathtt{production_failure}_{\notin}\ _\ _\ _\ _\ _\ _ \coloneqq \lightning
\end{aligned}$$

A careful inspection of the proofy arguments to each `failure` case will reveal that there is enough evidence to derive the appropriate contradiction. For example, the $\texttt{s} \neq \texttt{""}$ hypothesis of `nil_failure` contradicts the equalities implied by the type signature of $\texttt{min\_parse}_{\texttt{[]}}$, and the use of `[]` contradicts the equality implied by the use of `it::its` in the type signature of $\texttt{min\_parse}_{\texttt{[]}}$. Similarly, the $\texttt{s} \neq \texttt{"ch"}$ hypothesis of `terminal_failure` contradicts the equality implied by the usage of the single identifier `ch` in two different places in the type signature of $\texttt{min\_parse}_{\texttt{'ch'}}$.

3.3.1 Parsing Parses

We finally come to the most twisty part of the parser: parsing parse trees. Recall that our parser definition is polymorphic in a choice of `String` type. We proceed with the straw-man solution of literally passing in parse trees as strings to be parsed, such that parsing generates *minimal* parse trees, as introduced in Section 3.1 and defined formally in Section 3.2. Intuitively, we run a top-down traversal of the tree, pausing at each node before descending to its children. During that pause, we *eliminate one level of wastefulness*: if the parse tree is proving $\texttt{s} \in \texttt{X}$, we look for any subtrees also proving $\texttt{s} \in \texttt{X}$. If we find any, we replace the original tree with *the smallest duplicative subtree*. If we do not find any, we leave the tree unchanged. In either case, we then descend into "parsing" each subtree.

We define a function `deloop` to perform the one step of eliminating waste:

$$\texttt{deloop} : \texttt{ParseTreeOf nt s} \to \texttt{ParseTreeOf nt s}$$

This transformation is straightforward to define by structural recursion.

To implement all of the generic parameters of the parser, we must actually augment the result type of `deloop` with stronger types. Define the predicate $\texttt{Unloopy}(t)$ on parse trees t to mean that, where the root node of t proves $\texttt{s} \in \texttt{nt}$, for every subtree proving $\texttt{s} \in \texttt{nt}'$ (same string, possibly different nonterminal), (1) $\texttt{nt}'$ is in the set of allowed nonterminals, `unseen`, associated to the overall tree with dependent types, and (2) if this is not the root node, then $\texttt{nt}' \neq \texttt{nt}$.

We augment the return type of `deloop`, writing:

$$\{t : \texttt{ParseTreeOf nt s} \mid \texttt{Unloopy}(t)\}.$$

We instantiate the generic "string" type parameter of the general parser with this type family, so that, in implementing the different parameters to pass to the parser, we have the property available to us.

Another key ingredient is the "string" splitter, which naturally breaks a parse tree into its child trees. We define it like so:

$$\texttt{split \_ \_ } (\texttt{s} \in \texttt{it :: its}) :=$$

$$
\begin{array}{l}
\quad \textbf{case}\ \ \texttt{parse\_tree\_data s}\ \ \textbf{of} \\
\quad\ \ \left|\ \dfrac{\dfrac{\texttt{p}_1}{\texttt{s}_1 \in \texttt{it}} \qquad \dfrac{\texttt{p}_2}{\texttt{s}_2 \in \texttt{its}}}{\texttt{s}_1\texttt{s}_2 \in \texttt{it} :: \texttt{its}}\ \ \rightarrow\ \ \big[(\texttt{deloop}\ \texttt{p}_1, \texttt{deloop}\ \texttt{p}_2)\big]\right. \\
\quad\ \ \big|\ \_\ \ \rightarrow\ \ \lightning \\
\texttt{split}\ \_\ \_\ \_ \coloneqq \texttt{[]}
\end{array}
$$

Note that we use `it` and `its` nonlinearly; the pattern only binds if its `it` and `its` match those passed as arguments to `split`. We thus return a nonempty list only if the query is about a nonempty sequence of items. Because we use dependent types to enforce the requirement that the parse tree associated with a string matches the query we are considering, we can derive contradictions in the non-matching cases.

This splitter satisfies two important properties. First, it never returns the empty list on a parse tree whose list of productions is nonempty; call this property *nonempty preservation*. Second, it preserves `Unloopy`. We use both facts in the other parameters to the generic parser (and we leave their proofs as exercises for the reader—Coq solutions may be found in our source code).

Now recall that our general parser always returns a type of the form $\texttt{T}_{\texttt{success}} + \texttt{T}_{\texttt{failure}}$, for some $\texttt{T}_{\texttt{success}}$ and $\texttt{T}_{\texttt{failure}}$. We want our tree minimizer to return just the type of minimal trees. However, we can take advantage of the type isomorphism $T + \bot \cong T$ and instantiate $\texttt{T}_{\texttt{failure}}$ with $\bot$, the uninhabited type; and then apply a simple fix-up wrapper on top. Thus, we instantiate the general parser like so:

$$
\begin{array}{l}
\texttt{T}_{\texttt{success}}\ \ \texttt{s}_0\ \ \texttt{unseen}\ \ (\texttt{d} : \overline{\texttt{s} \in \texttt{X}}) \coloneqq \overline{\texttt{s} \in \texttt{X}}^{\ <(\texttt{s}_0, \texttt{unseen})} \\
\texttt{T}_{\texttt{failure}}\ \ \_\ \ \_\ \ \_ \coloneqq \bot
\end{array}
$$

The `success` cases are instantiated in an essentially identical way to the instantiation we used to get `option ParseTree`. The `terminal_failure` and `nil_failure` cases provide enough information ($\texttt{s} \neq \texttt{"ch"}$ and $\texttt{s} \neq \texttt{""}$, respectively) to derive $\bot$ from the existence of the appropriately typed parse tree. In the `cons_failure` case, we make use of the splitter's *nonempty preservation* behavior, after which all that remains is $\bot + \bot \rightarrow \bot$, which is trivial. In the $\texttt{production\_failure}_<$ and $\texttt{production\_failure}_=$ cases, it is sufficient to note that every nonterminal is mapped by some production to some sequence of items. Finally, to instantiate the $\texttt{production\_failure}_{\notin}$ case, we need to appeal to the `Unloopy`-ness of the tree to deduce that $\texttt{nt} \in \texttt{unseen}$. Then we can derive $\bot$ from the hypothesis that $\texttt{nt} \notin \texttt{unseen}$, and we are done.

We instantiate the general parser with an input type that requires `Unloopy`, so our final tree minimizer is really the composition of the instantiated parser with `deloop`, ensuring that invariant as we kick off the recursion.

3.3.2 Example

In Subsection 1.1.1, we defined an ambiguous grammar for `(ab)`* which led our naive parser to diverge. We will walk through the minimization of the following parse tree of `"abab"` into this grammar. For reference, Figure 3-3 contains the fully general implementation of our parser, modulo type signatures.

For reasons of space, define $\overline{T}$ to be the parse tree

$$\dfrac{\dfrac{\dfrac{}{\texttt{""} \in \epsilon}}{\texttt{""} \in \texttt{(ab)}^*} \qquad \dfrac{\dfrac{\dfrac{}{\texttt{"a"} \in \texttt{'a'}} \quad \dfrac{}{\texttt{"b"} \in \texttt{'b'}}}{\texttt{"a"} \cdot \texttt{"b"} \in \texttt{(ab)}^*}}{\texttt{"ab"} \in \texttt{(ab)}^*}}{\texttt{""} \cdot \texttt{"ab"} \in \texttt{(ab)}^*}\;\texttt{((ab)}^*\texttt{(ab)}^*\texttt{)}$$

Then we consider minimizing the parse tree:

$$\dfrac{\dfrac{\dfrac{\overline{T}}{\texttt{"ab"} \in \texttt{(ab)}^*} \qquad \dfrac{\overline{T}}{\texttt{"ab"} \in \texttt{(ab)}^*}}{\texttt{"ab"} \cdot \texttt{"ab"} \in \texttt{(ab)}^*}\;\texttt{((ab)}^*\texttt{(ab)}^*\texttt{)}}{\texttt{"abab"} \in \texttt{(ab)}^*}$$

Letting $\overline{T'_m}$ denote the same tree as $\overline{T'}$, the eventual minimized version of $\overline{T}$, but constructed as a `MinParseTree` rather than a `ParseTree`, the tree we will end up with is:

$$\dfrac{\dfrac{\dfrac{\overline{T'_m}}{\texttt{"ab"} \in \texttt{(ab)}^*} < (\texttt{"ab"}, [\texttt{(ab)}^*]) \qquad \dfrac{\overline{T'_m}}{\texttt{"ab"} \in \texttt{(ab)}^*} < (\texttt{"ab"}, [\texttt{(ab)}^*])}{\texttt{"ab"} \cdot \texttt{"ab"} \in \texttt{(ab)}^*} < (\texttt{"abab"}, [])}{\texttt{"abab"} \in \texttt{(ab)}^*} < (\texttt{"abab"}, [\texttt{(ab)}^*])$$

To begin, we call `parse`, passing in the entire tree as the string, and `(ab)`* as the nonterminal. To transform the tree into one that satisfies `Unloopy`, the first thing `parse` does is call `deloop` on our tree. In this case, `deloop` is a no-op; it promotes the deepest non-root nodes labeled with (`"abab"` $\in$ `(ab)`*), of which there are none.

We then take the following execution steps, starting with `unseen` $:=$ $\texttt{unseen}_0$ $:=$ `[(ab)*]`, the singleton list containing the only nonterminal, and $\texttt{s}_0 := $ `"abab"`.

1. We first ensure that we are not in an infinite loop. We check if $\texttt{s} < \texttt{s}_0$ (it is not, for they are both equal to `"abab"`), and then check if our current nonterminal, `(ab)`*, is in `unseen`. Since the second check succeeds, we remove `(ab)`* from `unseen`; calls made by this stack frame will pass `[]` for `unseen`.

2. We may consider only the productions for which the parse tree associated to the string is well-typed; we will describe the headaches this seemingly innocuous simplification caused us in Subsection 3.5.2. The only such production in this case is the one that lines up with the production used in the parse tree, labeled

```
parse nt s := parse' (s0 := s) (unseen := unseen0) (s ∈ nt)

parse' ("ch" ∈ 'ch') := inl terminal_success
parse' (_ ∈ 'ch') := inr (terminal_failure ↯)
parse' ("" ∈ ϵ) := inl nil_success
parse' (_ ∈ ϵ) := inr (nil_failure ↯)
parse' (s ∈ it :: its) :=
    case any_parse s it its (split (s ∈ it :: its)) of
     | inl ret → inl ret
     | inr ret → inr (cons_failure _ ret)
parse' (s ∈ nt) :=
    if s < s0
    then if (parse' (s0 := s) (unseen := unseen0) (s ∈ its)) succeeds
                    returning d for any production p mapping nt to its
         then inl (production_success< _ p d)
         else inr (production_failure< _ _)
    else if nt ∈ unseen
         then if (parse' (unseen := unseen − {nt}) (s ∈ its)) succeeds
                      returning d for any production p mapping nt to its
              then inl (production_success= _ p d)
              else inr (production_failure= _ _)
         else inr (production_failure∉ _ _)
any_parse s it its [] := inr (λ _ : (_ ∈ []). ↯)
any_parse s it its (x :: xs) :=
    case parse' (take_x s ∈ it), parse' (drop_x s ∈ its),
         any_parse s it its xs of
     | inl ret1, inl ret2, _        → inl (cons_success _ ret1 ret2)
     | _       , _       , inl ret' → inl ret'
     | ret1    , ret2    , inr ret' → inr _
```

where the hole on the last line constructs a proof of
$\forall\, x' \in (x :: xs),\ \ T_{failure}\ \_\ \_\ (take_{x'}\ s \in it) + T_{failure}\ \_\ \_\ (drop_{x'}\ s \in its)$
by using ret′ directly when $x' \in xs$, and using whichever one of ret_1 and ret_2 is on the right when $x' = x$. While straightforward, the use of sum types makes it painfully verbose without actually adding any insight; we prefer to elide the actual term.

Figure 3-3: Pseudo-Implementation of our parser. We adopt the convention that dependent indices to functions (e.g., unseen) are implicit.

$\texttt{(ab)}^*\texttt{(ab)}^*$.

3. We invoke `split` on our parse tree.

   (a) The `split` that we defined then invokes `deloop` on the two copies of the parse tree

$$\frac{\overline{T}}{\texttt{"ab"} \in \texttt{(ab)}^*}$$

   Since there are non-root nodes labeled with ($\texttt{"ab"} \in \texttt{(ab)}^*$), the label of the root node, we promote the deepest one. Letting T' denote the tree

$$\frac{\overline{\texttt{"a"} \in \texttt{'a'}} \qquad \overline{\texttt{"b"} \in \texttt{'b'}}}{\texttt{"a"} \cdot \texttt{"b"} \in \texttt{(ab)}^*} \; (\texttt{"ab"})$$

   the result of calling `deloop` is the tree

$$\frac{\overline{T'}}{\texttt{"ab"} \in \texttt{(ab)}^*}$$

   (b) The return of `split` is thus the singleton list containing a single pair of two parse trees; each element of the pair is the parse tree for $\texttt{"ab"} \in \texttt{(ab)}^*$ that was returned by `deloop`.

4. We invoke `parse` on each of the items in the sequence of items associated to $\texttt{(ab)}^*$ via the rule ($\texttt{(ab)}^*\texttt{(ab)}^*$). The two items are identical, and their associated elements of the pair returned by `split` are identical, so we only describe the execution once, on

$$\frac{\overline{T'}}{\texttt{"ab"} \in \texttt{(ab)}^*}$$

   (a) We first ensure that we are not in an infinite loop. We check if $\texttt{s} < \texttt{s}_0$. This check succeeds, for `"ab"` is shorter than `"abab"`. We thus reset `unseen` and $\texttt{s}_0$; calls made by this stack frame will pass $\texttt{unseen}_0 \equiv \texttt{[(ab)}^*\texttt{]}$ for `unseen`, and $\texttt{s} \equiv \texttt{"ab"}$ for $\texttt{s}_0$.

   (b) We may again consider only the productions for which the parse tree associated to the string is well-typed. The only such production in this case is the one that lines up with the production used in the parse tree T', labeled (`"ab"`).

   (c) We invoke `split` on our parse tree.

      i. The `split` that we defined then invokes `deloop` on the trees $\overline{\texttt{"a"} \in \texttt{'a'}}$ and $\overline{\texttt{"b"} \in \texttt{'b'}}$. Since these trees have no non-root nodes (let alone non-root nodes sharing a label with the root), `deloop` is a no-op.

      ii. The return of `split` is thus the singleton list containing a single pair of two parse trees; the first is the parse tree $\overline{\texttt{"a"} \in \texttt{'a'}}$, and the second is the parse tree $\overline{\texttt{"b"} \in \texttt{'b'}}$.

(d) We invoke `parse` on each of the items in the sequence of items associated to $\texttt{(ab)}^*$ via the rule ("ab"). Since both of these items are terminals, and the relevant equality check (that "a" is equal to "a", and similarly for "b") succeeds, `parse` returns `terminal_success`. We thus have the two `MinParseTrees`: $\overline{\texttt{"a"} \in \texttt{'a'}}$ and $\overline{\texttt{"b"} \in \texttt{'b'}}$.

(e) We combine these using `cons_success` (and `nil_success`, to tie up the base case of the list). We thus have the tree $\overline{T'_m}$.

(f) We apply $\texttt{production\_success}_<$ to this tree, and return the tree

$$\frac{\overline{T'_m}}{\texttt{"ab"} \in \texttt{(ab)}^*} < (\texttt{"ab"}, \texttt{[(ab)}^*\texttt{]})$$

5. We now combine the two identical trees returned by `parse` using `cons_success` (and `nil_success`, to tie up the base case of the list). We thus have the tree

$$\frac{\dfrac{\overline{T'_m}}{\texttt{"ab"} \in \texttt{(ab)}^*} < (\texttt{"ab"}, \texttt{[(ab)}^*\texttt{]}) \qquad \dfrac{\overline{T'_m}}{\texttt{"ab"} \in \texttt{(ab)}^*} < (\texttt{"ab"}, \texttt{[(ab)}^*\texttt{]})}{\texttt{"ab"} \cdot \texttt{"ab"} \in \texttt{(ab)}^*} < (\texttt{"abab"}, \texttt{[]})$$

6. We apply $\texttt{production\_success}_=$ to this tree, and return the tree we claimed we would end up with,

$$\frac{\dfrac{\dfrac{\overline{T'_m}}{\texttt{"ab"} \in \texttt{(ab)}^*} < (\texttt{"ab"}, \texttt{[(ab)}^*\texttt{]}) \qquad \dfrac{\overline{T'_m}}{\texttt{"ab"} \in \texttt{(ab)}^*} < (\texttt{"ab"}, \texttt{[(ab)}^*\texttt{]})}{\texttt{"ab"} \cdot \texttt{"ab"} \in \texttt{(ab)}^*} < (\texttt{"abab"}, \texttt{[]})}{\texttt{"abab"} \in \texttt{(ab)}^*} < (\texttt{"abab"}, \texttt{[(ab)}^*\texttt{]})$$

3.3.3 Parametricity

Before we can combine different instantiations of this interface, we need to know that they behave similarly. Inspection of the code, together with relational parametricity, validates assuming the following axiom, which should also be internally provable by straightforward induction (though we have not bothered to prove it). The alternative that we have actually taken to get our end-to-end proof to be axiom-free, in the code base we use to perform various optimizations described in Chapters 4, 5, 6 and 7, is to prove soundness more manually, for the instantiation of the parser that we use in those sections.

The *parser extensionality axiom* states that, for any fixed instantiation of `split`, and any arbitrary instantiations of the rest of the interface, giving rise to two different functions $\texttt{parse}_1$ and $\texttt{parse}_2$, we have

$$\begin{aligned} &\forall\ (\texttt{nt} : \texttt{Nonterminal})\ (\texttt{s} : \texttt{String}),\\ &\quad \texttt{bool\_of\_sum}\ (\texttt{parse}_1\ \texttt{nt}\ \texttt{s}) = \texttt{bool\_of\_sum}\ (\texttt{parse}_2\ \texttt{nt}\ \texttt{s}) \end{aligned}$$

where bool_of_sum is, for any types A and B, the function of type $A + B \to \texttt{Bool}$ obtained by sending everything in the left component to true and everything in the right component to false.

3.3.4 Putting It All Together

Now we have parsers returning the following types:

$$\begin{aligned}
\texttt{has_parse} &: \texttt{Nonterminal} \to \texttt{String} \to \texttt{Bool} \\
\texttt{parse} &: (\texttt{nt} : \texttt{Nonterminal}) \to (\texttt{s} : \texttt{String}) \\
&\to \texttt{option}\ (\texttt{ParseTreeOf}\ \texttt{nt}\ \texttt{s}) \\
\texttt{has_no_parse} &: (\texttt{nt} : \texttt{Nonterminal}) \to (\texttt{s} : \texttt{String}) \\
&\to \top + (\texttt{MinParseTreeOf}\ \texttt{nt}\ \texttt{s} \to \bot) \\
\texttt{min_parse} &: (\texttt{nt} : \texttt{Nonterminal}) \to (\texttt{s} : \texttt{String}) \\
&\to \texttt{ParseTreeOf}\ \texttt{nt}\ \texttt{s} \\
&\to \texttt{MinParseTreeOf}\ \texttt{nt}\ \texttt{s}
\end{aligned}$$

Note that we have taken advantage of the isomorphism $\top + \top \cong \texttt{Bool}$ for has_parse, the isomorphism $A + \top \cong \texttt{option}\ A$ for parse, and the isomorphism $A + \bot \cong A$ for min_parse.

We can compose these functions to obtain our desired correct-by-construction parser:

$$\begin{aligned}
&\texttt{parse_full} : (\texttt{nt} : \texttt{Nonterminal}) \to (\texttt{s} : \texttt{String}) \\
&\qquad\qquad \to \texttt{ParseTreeOf}\ \texttt{nt}\ \texttt{s} + (\texttt{ParseTreeOf}\ \texttt{nt}\ \texttt{s} \to \bot) \\
&\texttt{parse_full}\ \texttt{nt}\ \texttt{s} \coloneqq \\
&\quad \textbf{case}\ \ \texttt{parse}\ \texttt{nt}\ \texttt{s},\ \texttt{has_no_parse}\ \texttt{nt}\ \texttt{s}\ \ \textbf{of} \\
&\quad\ \mid \texttt{Some d},\ _ \qquad\quad \to\ \ \texttt{inl d} \\
&\quad\ \mid _ \qquad\ \ ,\ \texttt{inr nd}\ \ \to\ \ \texttt{inr}\ (\texttt{nd} \circ \texttt{min_parse}) \\
&\quad\ \mid _ \qquad\ \ ,\ _ \qquad\quad \to\ \ \lightning
\end{aligned}$$

In the final case, we derive a contradiction by applying the parser extensionality axiom, which says that parse and has_no_parse must agree on whether or not s parses as nt.

3.4 Semantic Actions

Our parsing algorithm can also be specialized to handle the common use case of semantic actions. Consider, for example, the following simultaneous specification of a grammar and some semantic actions:

$$e ::= n \qquad\qquad \{\texttt{int\_of\_string}(n)\}$$

$$\begin{array}{ll} \mid e_1 \ \texttt{"+"}\ e_2 & \{e_1 + e_2\} \\ \mid \texttt{"("}\ e\ \texttt{")"} & \{e\} \end{array}$$

Supposing we have defined this grammar separately for our parser, we can instantiate the interface as follows to implement these semantic actions:

$$\begin{array}{lll} \texttt{T}_{\texttt{success}}\ \_\ \_ & (\overline{\_ \in e}) & := \mathbb{Z} \\ \texttt{T}_{\texttt{success}}\ \_\ \_ & (\overline{\_ \in \texttt{'ch'}}) & := \top \\ \texttt{T}_{\texttt{success}}\ \_\ \_ & (\overline{\_ \in (\texttt{its} : [\texttt{Item}])}) & := \prod_{\texttt{it} \in \texttt{its}} \texttt{T}_{\texttt{success}}\ \_\ \_\ (\overline{\_ \in \texttt{it}}) \\ & & \\ \texttt{T}_{\texttt{failure}}\ \_\ \_\ \_ & & := \top \end{array}$$

As all `failure` cases are instantiated with `()`, we elide them.

The `terminal` case is trivial:

$$\texttt{terminal\_success}\ \_\ \_\ \_ := ()$$

The `nil` and `cons` cases are similarly straightforward; we have defined $\texttt{T}_{\texttt{success}}$ on item sequences to be the corresponding tuple type.

$$\begin{array}{l} \texttt{nil\_success}\ \_\ \_ := () \\ \texttt{cons\_success}\ \_\ \_\ \_\ \_\ \_\ \_\ \_\ \texttt{x}\ \ \texttt{xs} := (\texttt{x}, \texttt{xs}) \end{array}$$

We will use a single definition definition `production_success` to combine $\texttt{production\_success}_{<}$ and $\texttt{production\_success}_{=}$ here, as the definition does not depend on any of the arguments that vary between them. This is where the semantic actions take place. We deal first with the case of a number:

$$\texttt{production\_success}\ \_\ n\ \ e\ \ \texttt{s}\ \_\ \_\ \_ := \texttt{int\_of\_string}\ \ \texttt{s}$$

In the case of e_1 `"+"` e_2, we get a tuple of three values: the value corresponding to e_1, the value corresponding to `"+"` (which in this case must just be `()`), and the value corresponding to e_2:

$$\begin{array}{l} \texttt{production\_success}\ \_\ [e, \texttt{'+'}, e]\ \ e\ \ \_\ \_\ \_\ \ (\texttt{v}_1, \_, \texttt{v}_2) \\ \quad := \texttt{v}_1 + \texttt{v}_2 \end{array}$$

Finally, we deal with the case of `"("` e `")"`. We again get a tuple of three values: the value corresponding to `"("`, the value corresponding to e, and the value corresponding to `")"`. As above, the character literals are mapped to dummy $\top$ semantic values, so we ignore them.

$$\texttt{production\_success}\ \_\ [\texttt{'('}, e, \texttt{')'}]\ \ e\ \ \_\ \_\ \_\ \ (\_, \texttt{v}, \_)$$

```
:= v
```

3.5 Missteps, Insights, and Dependently Typed Lessons

We will now take a step back from the parser itself, and briefly talk about the process of coding it. We encountered a few pitfalls that we think highlight some key aspects of dependently typed programming, and our successes suggest benefits to be reaped from using dependent types.

3.5.1 The Trouble of Choosing the Right Types

Although we began by attempting to write the type signature of our parser, we found that trying to write down the correct interface, without any code to implement it, was essentially intractable. Giving your functions dependent types requires performing a nimble balancing act between being uselessly general on the one hand, and too overly specific on the other, all without falling from the high ropes of well-typedness onto the unforgiving floor of type errors.

We have found what we believe to be the worst sin the typechecker will let you get away with: having different levels of generality in different parts of your code base, which are supposed to interface with each other, without a thoroughly vetted abstraction barrier between them. Like setting your high ropes at different tensions, every trip across the interface will be costly, and if the abstraction levels get too far away, recovering your balance will require Herculean effort.

We eventually gave up on writing a dependently typed interface from the start, and decided instead to implement a simply typed Boolean recognizer, together with proofs of soundness and completeness. Once we had in hand these proofs, and the data types required to carry them out, we found that it was mostly straightforward to write down the interface and refine our parser to inhabit its newly generalized type.

3.5.2 Misordered Splitters

One of our goals in this presentation was to hide most of the abstraction-level mismatch that ended up in our actual implementation, often through clever use of notation overloading. One of the most significant mismatches we managed to overcome was the way to represent the set of productions. In this chapter, we left the type as an abstract mathematical set, allowing us to forgo concerns about ordering, quantification, and occasionally well-typedness.

In our Coq implementation, we fixed the type of productions to be a list very early on, and paid the price when we implemented our parse-tree parser. As mentioned in the execution of the example in Subsection 3.3.2, we wanted to restrict our attention to certain productions and rule out the other ones using dependent types. This should be possible if we parameterize over not just a splitter, but a production-selector, and only require that our string type be well-typed for productions given by the production-selector. However, the implementation that we currently have requires a well-typed string type for all productions; furthermore, it does not allow the order in which productions are considered to depend on the augmented string data. We paid for this with the extra 300 lines of code we had to write to interleave two different splitters, so that we could handle the cases that we dismissed above as being ill-typed and therefore not necessary to consider. That is, because our types were not formulated in a way that actually made these cases ill-typed, we had to deal with them, much to our displeasure.

3.5.3 Minimal Parse Trees vs. Parallel Traces

Taking another step back, our biggest misstep actually came before we finished the completeness proof for our simply typed Boolean recognizer.

When first constructing the type `MinParseTree`, we thought of them genuinely as minimal parse trees (ones without a duplicate label in any single path). After much head-banging, of knowledge that a theorem was obviously true, against proof goals that were obviously impossible, we discovered the single biggest insight—albeit a technical one—of the project. The type of "minimal parse trees" we had originally formulated did not match the parse trees produced by our algorithm. A careful examination of the algorithm execution in Subsection 3.3.2 should reveal the difference.[1] Our insight, thus, was to conceptualize the data type as the type of traces of parallel executions of our particular parser, rather than as truly minimal parse trees.

This may be an instance of a more general phenomenon present when programming with dependent types: subtle friction between what you think you are doing and what you are actually doing often manifests as impossible proof goals.

[1] For readers wanting to skip that examination: the algorithm we described allows a label $(\texttt{s} \in \texttt{nt})$ to appear one extra time along a path if, the first time it appears, its parent node's label, $(\texttt{s}' \in \texttt{nt}')$, satisfies $\texttt{s} < \texttt{s}'$. That is, between shrinking the string being parsed and shrinking it again, the first nonterminal that appears may be duplicated once.

Chapter 4

Refining Splitters by Fiat

4.1 Splitters at a Glance

We have now finished describing the general parsing algorithm, as well as its correctness proofs; we have an algorithm that decides whether or not a given structure can be imposed on any block of unstructured text. The algorithm is parameterized on an "oracle" that describes how to split the string for each rule; essentially all of the algorithmically interesting content is in the splitters. For the remainder of this thesis, we will focus on how to implement the splitting oracle. Correctness is not enough, in general; algorithms also need to be fast to use. We thus focus primarily on efficiency when designing splitting algorithms, and work in a framework that guarantees correctness.

The goals of this work, as mentioned in Section 2.5, are to present a framework for constructing proven-correct parsers incrementally and argue for its eventual feasibility. To this end, we build on the previous work of Fiat [11], to allow us to build programs incrementally while maintaining correctness guarantees. This section will describe Fiat and how it is used in this project. The following sections will focus more on the details of the splitting algorithms and less on Fiat itself.

4.2 What Counts as Efficient?

To guide our implementations, we characterize efficient splitters informally, as follows. Although our eventual concrete efficiency target is to be competitive with extant open-source JavaScript parsers, when designing algorithms, we aim at the asymptotic efficiency target of linearity in the length of the string. In practice, the dominating concern is that doubling the length of the string should only double the duration of the parse, and not quadruple it (or more!). To be efficient, it suffices to have the splitter return at most one index. In this case, the parsing time is $\mathcal{O}($length of string $\times$ (product over all nonterminals of the number of possible rules for that nonterminal)).

Consider, for example, the following very silly grammar for parsing either the empty string or the character `'a'`: let $\texttt{E}_0 ::= \epsilon$ and $\texttt{F}_0 ::= \epsilon$ denote nonterminals with a single production rule which allows them to parse the empty string. Let $\texttt{E}_{i+1} ::= \texttt{E}_i \mid \texttt{F}_i$ and $\texttt{F}_{i+1} ::= \texttt{E}_i \mid \texttt{F}_i$ be nonterminals, for each i, which have two rules which both eventually allow them to parse the empty string. If we let $\texttt{G} ::= \texttt{E}_i \mid \texttt{'a'}$ for some i, then, to successfully parse the string `"a"`, we end up making approximately 2^{i+1} calls to the splitter.

To avoid hitting this worst-case scenario, we can use a nonterminal-picker, which returns the list of possible production rules for a given string and nonterminal. As long as it returns at most one possible rule in most cases, in constant time, the parsing time will be $\mathcal{O}$(length of string); backtracking will never happen. This is future work.

4.3 Introducing Fiat

4.3.1 Incremental Construction by Refinement

Efficiency targets in hand, we move on to incremental construction. The key idea is that parsing rules tend to fall into clumps that are similar between grammars. For example, many grammars use delimiters (such as whitespace, commas, or binary operation symbols) as splitting points, but only between well-balanced brackets (such as double quotes, parentheses, or comment markers). We can take advantage of these similarities by baking the relevant algorithms into basic building blocks, which can then be reused across different grammars. To allow this reuse, we construct the splitters incrementally, allowing us to deal with different rules in different ways.

The Fiat framework [11] is the scaffolding of our splitter implementations. As a framework, the goal of Fiat is to enable library writers to construct algorithmic building blocks packaged with correctness guarantees, in such a way that users can easily and mostly automatically make use of these building blocks when they apply.

4.3.2 The Fiat Mindset

The correctness guarantees of Fiat are based on specifications in the form of propositions in Gallina, the mathematical language used by Coq. For example, the specification of a valid `has_parse` method is that `has_parse nt str` $=$ `true` $\longleftrightarrow$ `inhabited (ParseTreeOf nt s)`. Fiat allows incremental construction of algorithms by providing a language for seamlessly mixing specifications and code. The language is a light-weight monadic syntax with one extra operator: a nondeterministic choice operator; we define the following combinators:

$\texttt{x} \leftarrow \texttt{c};\ \ \texttt{c}'$	Run `c` and store the result in `x`; continue with $\texttt{c}'$, which may mention `x`.
$\texttt{c};;\ \ \texttt{c}'$	Run `c`. If it terminates, throw away the result, and run $\texttt{c}'$.

$\texttt{ret x}$	Immediately return the value $\texttt{x}$.
$\{\texttt{x} \mid \texttt{P(x)}\}$	Nondeterministically choose a value of $\texttt{x}$ satisfying $\texttt{P}$. If none exists, the program is considered to not terminate.

An algorithm starts out as a nondeterministic choice of a value satisfying the specification. Coding then proceeds by refinement. Formally, we say that a computation $\texttt{c}'$ *refines* a computation $\texttt{c}$, written $\texttt{c}' \subseteq \texttt{c}$, if every value that $\texttt{c}'$ can compute to, $\texttt{c}$ can also compute to. We freely generate the relation "the computation $\texttt{c}$ can compute to the value $\texttt{v}$", written $\texttt{c} \rightsquigarrow \texttt{v}$, by the rules:

$$\begin{array}{rcll}
\texttt{ret v} & \rightsquigarrow & \texttt{v} & \\
\{\texttt{x} \mid \texttt{P}(\texttt{x})\} & \rightsquigarrow & \texttt{v} & \text{iff } \texttt{v} \text{ satisfies } \texttt{P} \\
(\texttt{c};;\ \ \texttt{c}') & \rightsquigarrow & \texttt{v} & \text{iff there is a } \texttt{v}' \text{ such that } \texttt{c} \rightsquigarrow \texttt{v}' \text{ and } \texttt{c}' \rightsquigarrow \texttt{v} \\
(\texttt{x} \leftarrow \texttt{c};\ \ \texttt{c}'(\texttt{x})) & \rightsquigarrow & \texttt{v} & \text{iff there is a } \texttt{v}' \text{ such that } \texttt{c} \rightsquigarrow \texttt{v}' \text{ and } \texttt{c}'(\texttt{v}') \rightsquigarrow \texttt{v}
\end{array}$$

In our use case, we express the specification of the splitter as a nondeterministic choice of a list of split locations, such that any splitting location that results in a valid parse tree is contained in the list. More formally, we can define the proposition

$$\begin{array}{l}
\texttt{split_list_is_complete} : \texttt{Grammar} \to \texttt{String} \to \texttt{[Item]} \to [\mathbb{N}] \to \textbf{Prop} \\
\texttt{split_list_is_complete G str [] splits} = \bot \\
\texttt{split_list_is_complete G str (it :: its) splits} \\
\quad = \forall\, \texttt{n},\ \texttt{n} < \texttt{length str} \\
\qquad\qquad \to (\texttt{has_parse it (take n str)} \wedge \texttt{has_parse its (drop n str)}) \\
\qquad\qquad \to \texttt{n} \in \texttt{splits}
\end{array}$$

where we overload `has_parse` to apply to items and productions alike, and where we use $[\mathbb{N}]$ to denote the type of lists of natural numbers. In practice, we pass the first item and the rest of the items as separate arguments, so that we don't have to deal with the empty list case.

Let `production_is_reachable G p` be the proposition that `p` could show up during parsing, i.e., that `p` is a tail of a rule associated to some valid nonterminal in the grammar; we define this by folding over the list of valid nonterminals. The specification of the splitter, as a nondeterministic computation, for a given grammar `G`, a given string `str`, and a given rule `it::its`, is then:

$$\begin{array}{l}
\{\ \texttt{splits} : [\mathbb{N}] \\
\mid \texttt{production_is_reachable G (it :: its)} \\
\quad \to \texttt{split_list_is_complete G str it its splits}\ \}
\end{array}$$

We then refine this into a choice of a splitting location for each rule actually in the grammar (checking for equality with the given rule), and then can refine (implement) the splitter for each rule separately. For example, for the grammar `(ab)`*, defined to have a single nonterminal `(ab)`* which can either be empty, or be mapped to `'a' 'b' (ab)`*, we would refine this to the computation:

```
If [(ab)*]  =p  it :: its Then
    { splits : [ℕ]
    | split_list_is_complete G str (ab)* [] splits }
Else If ['b', (ab)*]  =p  it :: its Then
    { splits : [ℕ]
    | split_list_is_complete G str 'b' [(ab)*] splits }
Else If ['a', 'b', (ab)*]  =p  it :: its Then
    { splits : [ℕ]
    | split_list_is_complete G str 'a' ['b', (ab)*] splits }
Else
    { dummy_splits : list ℕ | ⊤ }
```

where $=_p$ refers to a Boolean equality test for productions. Note that in the final case, we permit any list to be picked, because whenever the production we are handling is reachable, the value returned by that case will never be used.

We can now refine each of these cases separately, using `setoid_rewrite`; this tactic replaces one subterm of the goal with an "equivalent" subterm, where the "equivalence" can be any transitive relation which is respected by the functions applied to the subterm. Using `setoid_rewrite` allows us to hide the glue required to state our lemmas about computations as wholes, while using them to replace *subterms* of other computations. The key to refining each part separately, to making Fiat work, is that the refinement rules package their correctness properties, so users don't have to worry about correctness when programming by refinement. We use Coq's setoid rewriting machinery to automatically glue together the various correctness proofs when refining only a part of a program.

For example, we might have a lemma `singleton` which says that returning the length of the string is a valid refinement for any rule that has only one nonterminal; its type, for a particular grammar `G`, a particular string `str`, and a particular nonterminal `nt`, would be

```
singleton G str nt
    : (ret [length str])
    ⊆
```

```
{ splits : [ℕ]
| split_list_is_complete G str nt [] splits }
```

Then `setoid_rewrite (singleton _ _ (ab)*)` would refine

```
If [(ab)*] =p it :: its Then
    { splits : [ℕ]
    | split_list_is_complete G str (ab)* [] splits }
Else If ['b', (ab)*] =p it :: its Then
    { splits : [ℕ]
    | split_list_is_complete G str 'b' [(ab)*] splits }
Else If ['a', 'b', (ab)*] =p it :: its Then
    { splits : [ℕ]
    | split_list_is_complete G str 'a' ['b', (ab)*] splits }
Else
    { dummy_splits : [ℕ] | ⊤ }
```

into

```
If [(ab)*] =p it :: its Then
    ret [length str]
Else If ['b', (ab)*] =p it :: its Then
    { splits : [ℕ]
    | split_list_is_complete G str 'b' [(ab)*] splits }
Else If ['a', 'b', (ab)*] =p it :: its Then
    { splits : [ℕ]
    | split_list_is_complete G str 'a' ['b', (ab)*] splits }
Else
    { dummy_splits : [ℕ] | ⊤ }
```

Note that the only change is in the computation returned in the first branch of the conditional.

We now describe the refinements that we do within this framework, to implement efficient splitters.

4.4 Optimizations

4.4.1 An Easy First Optimization: Indexed Representation of Strings

One optimization that is always possible is to represent the current string being parsed in this recursive call as a pair of indices into the original string. This allows us to optimize the code doing string manipulation, as it will no longer need to copy strings around, only do index arithmetic.

This optimization, as well as the trivial optimizations described in Chapter 5, are implemented automatically by the initial lines of any parser refinement process in Fiat.

4.4.2 Putting It All Together

Now that we have the concepts and ideas behind refining parsers, or, more precisely, splitting oracles for parsers, in the Fiat framework, what does the code actually look like? Every refinement process, which defines a representation for strings, along with a proven-correct method of splitting them, begins with the same code:

```
Lemma ComputationalSplitter' : FullySharpened (string_spec G).
Proof.
  start honing parser using indexed representation.

  hone method "splits".
  {
    simplify parser splitter.
```

We begin the proof with a call to the tactic `start honing parser using indexed representation`; Coq's mechanism for custom tactic notations makes it easy to define such space-separated identifiers. This tactic takes care of the switch to using indices into the original string, of replacing the single nondeterministic choice of a complete list of splits with a sequence of `If` statements returning separate computations for each rule, and of replacing nondeterministic choices with direct return values when such values can be determined by trivial rules (which will be described in Chapter 5). The tactic `hone method "splits"` says that we want to refine the splitter, rather than, say, the representation of strings that we are using. The tactic `simplify parser splitter` performs a number of straightforward and simple optimizations, such as combining nested `If` statements which return the same value.

4.4.3 Upcoming Optimizations

In the next few sections, we build up various strategies for splitters. Although our eventual target is JavaScript, we cover only a more modest target of very simple arithmetical expressions in this thesis. We begin by tying up the `(ab)`* grammar, and then moving on to parse numbers, parenthesized numbers, expressions with only numbers and `'+'`, and then expressions with numbers, `'+'`, and parentheses.

Chapter 5

Fixed-Length Items, Parsing (ab)*; Parsing #s; Parsing #, ()

In this chapter, we explore the Fiat framework with a few example grammars, which we describe how to parse. Because these rules are so straightforward, they can be handled automatically, in the very first step of the derivation; we will explain how this works, too.

Recall the grammar for the regular expression $\texttt{(ab)}^*$:

$$\texttt{(ab)}^* ::= \epsilon \mid \texttt{'a'}\ \texttt{'b'}\ \texttt{(ab)}^*$$

In addition to parsing this grammar, we will also be able to parse the grammar for non-negative parenthesized integers:

$$\begin{aligned}
\texttt{pexpr} &::= \texttt{'('}\ \texttt{pexpr}\ \texttt{')'} \mid \texttt{number} \\
\texttt{number} &::= \texttt{digit}\ \texttt{number?} \\
\texttt{number?} &::= \epsilon \mid \texttt{number} \\
\texttt{digit} &::= \texttt{'0'} \mid \texttt{'1'} \mid \texttt{'2'} \mid \texttt{'3'} \mid \texttt{'4'} \mid \texttt{'5'} \mid \texttt{'6'} \mid \texttt{'7'} \mid \texttt{'8'} \mid \texttt{'9'}
\end{aligned}$$

5.1 Parsing (ab)*: At Most One Nonterminal

The simpler of these grammars is the one for $\texttt{(ab)}^*$. The idea is that if any rule has at most one nonterminal, then there is only one possible split: we assign one character to each terminal and the remaining characters to the single nonterminal.

For any given rule, we can compute straightforwardly whether or not this is the case;

Haskell-like pseudocode for doing so is:

```
has_at_most_one_nt : [Item] → Bool
has_at_most_one_nt [] ≔ true
has_at_most_one_nt ('ch'::xs) ≔ has_at_most_one_nt xs
has_at_most_one_nt (nt::xs) ≔ has_only_terminals xs

has_only_terminals : [Item] → Bool
has_only_terminals [] ≔ true
has_only_terminals ('ch'::xs) ≔ has_only_terminals xs
has_only_terminals (nt::xs) ≔ false
```

The code for determining the split location is even easier: if the first item of the rule is a terminal, then split at character 1; if the first item of the rule is a nonterminal, and there are n remaining items in the rule, then split n characters before the end of the string.

5.2 Parsing Parenthesized Numbers: Fixed Lengths

The grammar for parenthesized numbers has only one rule with multiple nonterminals: the rule for `number` ::= `digit number?`. The strategy here is also simple: because `digit` only accepts strings of length exactly 1, we always want to split after the first character.

The following pseudocode determines whether or not all strings parsed by a given item are a fixed length, and, if so, what that length is:

```
fixed-length-of : [Nonterminal] → [Item] → option ℤ
fixed-length-of valid_nonterminals [] := Some 0
fixed-length-of valid_nonterminals (Terminal _::xs)
  := case fixed-length-of valid_nonterminals xs of
       | Some k → Some (1 + k)
       | None   → None
fixed-length-of valid_nonterminals (Nonterminal nt::xs)
  if nt ∈ valid_nonterminals
  then let lengths :=
             map (fixed-length-of (remove nt valid_nonterminals))
                 (Lookup nt)
       in if all of the elements of lengths are equal to L for some L
          then case L, fixed-length-of valid_nonterminals xs of
                 | Some L', Some k → Some (L' + k)
                 | _      , _      → None
```

```
        else None
  else None
```

We have proven that for any nonterminal for which this method returns `just` k, the only valid split of any string for this rule is at location k. This is the correctness obligation that Fiat demands of us to be able to use this rule.

5.3 Putting It Together

Both of these refinement strategies are simple and complete for the rules they handle; if a rule has at most one nonterminal, or if the first element of a rule has a fixed length, then we can't do any better than these rules. Therefore, we incorporate them into the initial invocation of `start honing parser using indexed representation`. As described in Chapter 4, to do this, we express the splitter by folding `If` statements over all of the rules of the grammar that are reachable from valid nonterminals. The `If` statements check equality of the rule against the one we were given, and, if they match, look to see if either of these strategies applies. If either does, than we return the appropriate singleton value. If neither applies, then we default to the nondeterministic pick of a list containing all possible valid splits. The results of applying this procedure without treating any rules specially was shown in Subsection 4.3.2. The readers interested in precise details can find verbatim code for the results of applying this procedure, including the rules of this chapter, in Appendix A.1. For the similarly interested readers, the Coq code that computes the goal that the user is presented with, after `start honing parser using indexed representation`, can be found in Appendix A.2.

Chapter 6

Disjoint Items, Parsing #, +

Consider now the following grammar for arithmetic expressions involving '+' and numbers:

$$
\begin{aligned}
\texttt{expr} &::= \texttt{number +expr?} \\
\texttt{+expr?} &::= \epsilon \mid \texttt{'+' expr} \\
\texttt{number} &::= \texttt{digit number?} \\
\texttt{number?} &::= \epsilon \mid \texttt{number} \\
\texttt{digit} &::= \texttt{'0'} \mid \texttt{'1'} \mid \texttt{'2'} \mid \texttt{'3'} \mid \texttt{'4'} \mid \texttt{'5'} \mid \texttt{'6'} \mid \texttt{'7'} \mid \texttt{'8'} \mid \texttt{'9'}
\end{aligned}
$$

The only rule not handled by the strategies of the previous chapter is the rule `expr` ::= `number +expr?`. We can handle this rule by noticing that the set of characters in the strings accepted by `number` is disjoint from the set of possible first characters of the strings accepted by `+expr?`. Namely, all characters in strings accepted by `number` are digits, while the first character of a string accepted by `+expr?` can only be '+'.

The following code computes the set of possible characters of a rule:

```
possible-terminals' : [Nonterminal] → [Item] → [Char]
possible-terminals' _ [] := []
possible-terminals' valid_nonterminals ('ch' :: xs)
  := [ch] ∪ possible-terminals' valid_nonterminals xs
possible-terminals' valid_nonterminals (nt :: xs)
  := if nt ∈ valid_nonterminals
     then fold
            (∪)
            (possible-terminals' valid_nonterminals xs)
            (map (possible-terminals' (remove nt valid_nonterminals))
```

```
              (Lookup nt))
     else possible-terminals' valid_nonterminals xs

possible-terminals : Grammar → [Item] → [Char]
possible-terminals G its
  := possible-terminals' (valid_nonterminals_of G) its
```

In the case where the nonterminal is not in the list of valid nonterminals, we assume that we have already seen that nonterminal higher up the chain of recursion (which we will have, if it is valid according to the initial list), and thus don't have to recompute its possible terminals.

The following code computes the set of possible first characters of a rule:

```
possible-first-terminals' : [Nonterminal] → [Item] → [Char]
possible-first-terminals' _ [] := []
possible-first-terminals' valid_nonterminals ('ch' :: xs)
  := [ch]
possible-first-terminals' valid_nonterminals (nt :: xs)
  := (if nt ∈ valid_nonterminals
      then
        fold
          (∪)
          []
          (map (possible-first-terminals' (remove nt valid_nonterminals))
               (Lookup nt))
      else [])
     ∪
     (if has_parse nt ""
      then possible-first-terminals' valid_nonterminals xs
      else [])


possible-first-terminals : Grammar → [Item] → [Char]
possible-first-terminals G its
  := possible-first-terminals' (valid_nonterminals_of G) its
```

We can decide `has_parse` at refinement time with the brute-force parser, which tries every split; when the string we're parsing is empty, $\mathcal{O}$(length!) is not that long. The idea is that we recursively look at the first element of each production; if it is a terminal, then that is the only possible first terminal of that production. If it's a nonterminal, then we have to fold the recursive call over the alternatives. Additionally, if the nonterminal might end up parsing the empty string, then we have to also move on to the next item in the production and see what its first characters might be.

By computing whether or not these two lists are disjoint, we can decide whether or not this rule applies. When it applies, we can either look for the first character not in the first list (in this example, the list of digits), or we can look for the first character which is in the second list (in this case, the `'+'`). Since there are two alternatives, we leave it up to the user to decide which one to use.

For this grammar, we choose the shorter list. We define a function:

$$\texttt{index\_of\_first\_character\_in} : \texttt{String} \to \texttt{[Char]} \to \mathbb{N}$$

by folding over the string. We can then phrase the refinement rule as having type

$$\begin{aligned}
&\texttt{is_disjoint}\ (\texttt{possible-terminals G [it]})\ (\texttt{possible-first-terminals G its}) \\
&\quad = \texttt{true} \\
\to\ &\texttt{ret [index_of_first_character_in str}\ (\texttt{possible-first-terminals G its})\texttt{]} \\
&\subseteq \\
&\{\ \texttt{splits} : [\mathbb{N}] \\
&\mid \texttt{split_list_is_complete G str it its splits}\ \}
\end{aligned}$$

Applying this rule involves normalizing the calls to `is_disjoint`, `possible-terminals`, and `possible-first-terminals`. This normalization shows up as a side condition, given to us by `setoid_rewrite`, which can be solved by the tactic `reflexivity`; the `reflexivity` tactic proves the equality of two things which are syntactically equal modulo computation.

Chapter 7

Parsing Well-Parenthesized Expressions

7.1 At a Glance

We finally get to a grammar that requires a non-trivial splitting strategy. In this section, we describe how to parse strings for a grammar that accepts arithmetical expressions involving numbers, pluses, and well-balanced parentheses. More generally, this strategy handles any binary operation with guarded brackets.

7.2 Grammars We Can Parse

Consider the following two grammars, with `digit` denoting the nonterminal that accepts any single decimal digit.

Parenthesized addition:

$$
\begin{aligned}
\texttt{expr} &::= \texttt{pexpr}\ \texttt{+expr}\\
\texttt{+expr} &::= \epsilon \mid \texttt{'+'}\ \texttt{expr}\\
\texttt{pexpr} &::= \texttt{number} \mid \texttt{'('}\ \texttt{expr}\ \texttt{')'}\\
\texttt{number} &::= \texttt{digit}\ \texttt{number?}\\
\texttt{number?} &::= \epsilon \mid \texttt{number}\\
\texttt{digit} &::= \texttt{'0'} \mid \texttt{'1'} \mid \texttt{'2'} \mid \texttt{'3'} \mid \texttt{'4'} \mid \texttt{'5'} \mid \texttt{'6'} \mid \texttt{'7'} \mid \texttt{'8'} \mid \texttt{'9'}
\end{aligned}
$$

We have carefully constructed this grammar so that the first character of the string suffices to uniquely determine which rule of any given nonterminal to apply.

S-expressions are a notation for nested space-separated lists. By replacing `digit` with a nonterminal that accepts any symbol in a given set, which must not contain either of the brackets, nor whitespace, and replacing `'+'` with a space character `' '`, we get a grammar for S-expressions:

$$
\begin{aligned}
\texttt{expr} &::= \texttt{pexpr}\ \texttt{sexpr} \\
\texttt{sexpr} &::= \epsilon \mid \texttt{whitespace}\ \texttt{expr} \\
\texttt{pexpr} &::= \texttt{atom} \mid \texttt{'('}\ \texttt{expr}\ \texttt{')'} \\
\texttt{atom} &::= \texttt{symbol}\ \texttt{atom?} \\
\texttt{atom?} &::= \epsilon \mid \texttt{atom} \\
\texttt{whitespace} &::= \texttt{whitespace-char}\ \texttt{whitespace?} \\
\texttt{whitespace?} &::= \epsilon \mid \texttt{whitespace} \\
\texttt{whitespace-char} &::= \texttt{'\ '} \mid \texttt{'\textbackslash n'} \mid \texttt{'\textbackslash t'} \mid \texttt{'\textbackslash r'}
\end{aligned}
$$

7.3 The Splitting Strategy

7.3.1 The Main Idea

The only rule not already handled by the baseline automation of `start honing parser using indexed representation` is the rule that says that a `pexpr +expr` is an `expr`. The key insight here is that, to know where to split, we need to know where the next `'+'` at the current level of parenthesization is. If we can compute an appropriate lookup table in time linear in the length of the string, then our splitter overall will be linear.

7.3.2 Building the Lookup Table

We build the table by reading the string from right to left, storing for each character the location of the next `'+'` at the current level of parenthesization. To compute this location we keep a list of the location of the next `'+'` at every level of parenthesization.

Let's start with a very simple example, before moving to a more interesting one. To parse `"4+5"`, we are primarily interested in the case where we are parsing something that is a number, or parenthesized on the left, followed by a `'+'`, followed by any expression. For this item, we want to split the string right before the `'+'`, and say that the `"4"` can be parsed as a number (or parenthesized expression), and that the `"+5"` can be parsed as a `'+'` followed by an expression.

To do this, we build a table that keeps track of the location of the next `'+'`, starting from the right of the string. We will end up with the table:

Table Entries:		1	0	∅	
String:	"	4	+	5	"

At the `'5'`, there is no next `'+'`, and we are not parenthesized at all, so we record this as ∅. At the `'+'`, we record that there is a `'+'` at the current level of parenthesization, with `0`. Then, since the `'4'` is not a `'+'`, we increment the previous number, and store `1`. This is the correct location to split the string: we parse one character as a `number` and the rest as `+expr`.

Now let's try a more complicated example: `"(1+2)+3"`. We want to split this string into `"(1+2)"` and `"+3"`. The above strategy is insufficient to do this; we need to keep track of the next `'+'` at all levels of parenthesization at each point. We will end up with the following table, where the bottom-most element is the current level, and the ones above that are higher levels. We use lines to indicate chains of numbers at the same level of parenthesization.

```
                              4 – 3 – 2 – 1
Table Entries:               /             \
                          5   1 – 0 – ∅ – ∅   0 – ∅
String:                "  (   1   +   2   )   +   3  "
```

We again start from the right. Since there are no `'+'`s that we have seen, we store the singleton list `[∅]`, indicating that we know about only the current level of parenthesization, and that there is no `'+'` to the right. At the `'+'` before the `"3"`, we store the singleton list `[0]`, indicating that the current character is a `'+'`, and we only know about one level of parenthesization. At the `')'`, we increment the counter for `'+'`, but we also now know about a new level of parenthesization. So we store the two element list `[∅, 1]`. At the `'3'`, we increment all numbers, storing `[∅, 2]`. At the `'+'` before the `"2"`, we store `0` at the current level, and increment higher levels, storing `[0, 3]`. At the `'1'`, we simply increment all numbers, storing `[1, 4]`. Finally, at the `'('`, we pop a level of parenthesization and increment the remaining numbers, storing `[5]`. This is correct; we have 5 characters in the first string, and when we go to split `"1+2"` into `"1"` and `"+2"`, we have the list `[1, 4]`, and the first string does indeed contain 1 character.

As an optimization, we can drop all but the first element of each list once we're done computing, and, in fact, can do this as we construct the table. However, for correctness, it is easier to reason about a list located at each location.

7.3.3 Table Correctness

What is the correctness condition on this table? The correctness condition Fiat gives us is that the splits we compute must be the only ones that give rise to valid parses. This is hard to reason about directly, so we use an intermediate correctness condition. Intuitively, this condition corresponds to saying that the table should have a cell pointing to a given location if and only if that location is the first `'+'` at the relevant level of parenthesization; importantly, we talk about *every* level of parenthesization at every point in the string.

More formally, each cell in the table corresponds to some level of parenthesization; if a list ℓ is associated to a given position in the string, then the n^{th} element of that list talks about the `'+'` at parenthesization level n. The correctness condition is then: for any cell of the table, talking about parenthesization level n, if the cell is empty (is $\emptyset$, or does not exist at all), then, for any substring s of the string starting at the location corresponding to that cell, and ending with a `'+'`, the result of prepending n open parentheses to s must *not* be well-parenthesized. Additionally, if the cell points to a given location, then that location must contain a `'+'`, and the fragment of the string starting at the current location and going up to but not including the `'+'`, must not contain any `'+'`s which are "exposed"; all intervening `'+'`s must be hidden by a pair of parentheses enclosing a well-parenthesized substring of this fragment.

Even more formally, we can define a notation of paren-balanced and paren-balanced-hiding-`'+'`. Say that a string is paren-balanced at level n if it closes exactly n more parentheses than it opens, and there is no point at which it has closed more than n more parentheses than it has opened. So the string `"1+2)"` is paren-balanced at level 1 (because it closes 1 more parenthesis than it opens), and the string `"1+2)+(3"` is not paren balanced at any level (though the string `"1+2)+(3)"` is paren-balanced at level 1). A string is paren-balanced-hiding-`'+'` at level n if it is paren-balanced at level n, and, at any point at which there is a `'+'`, at most $n-1$ more parentheses have been closed than opened. So `"(1+2)"` is paren-balanced-hiding-`'+'` at level 0, and `"(1+2))"` is paren-balanced-hiding-`'+'` at level 1, and `"(1+2)+3"` is not paren-balanced-hiding-`'+'` at any level, though it is paren-balanced at level 0.

Then, the formal correctness condition is that if a cell at parenthesis level n points to a location ℓ, then the string from the cell up to but not including ℓ must be paren-balanced-hiding-`'+'` at level n, and the character at location ℓ must be a `'+'`. If the cell is empty, then the string up to but not including any subsequent `'+'` must not be paren-balanced at level n.

The table computed by the algorithm given above satisfies this correctness condition, and this correctness condition implies that the splitting locations given by the table are the only ones that produce valid parse trees; there is a unique table satisfying this correctness condition (because it picks out the *first* `'+'` at the relevant level), and any split location which is not appropriately paren-balanced/paren-balanced-hiding results in no parse tree.

7.3.4 Diving into Refinement Code

Although the rule itself is non-trivial, our goal is to make using this rule as trivial as possible; we now explain how this refinement rule requires only one line of code to use (modulo long tactic names):

```
setoid_rewrite refine_binop_table;
[ presimpl_after_refine_binop_table | reflexivity.. ].
```

The tactic `presimpl_after_refine_binop_table` is nothing more than a neatly packaged collection of calls to the `unfold` tactic, which performs δ-reduction (definition unfolding); this unfolding allows the subsequent call to `reflexivity` to instantiate some existential variables (placeholders which tell Coq "fill me in later"), without needing higher-order unification. As mentioned at the end of Chapter 6, `reflexivity` takes care of discharging the side conditions which can be decided by computation; this is commonly called "reflective automation," for its widespread use in proving a lemma by appealing to computation run on a "reflection," or syntactic representation, of that lemma [4].

There are three components to making a rule that can be used with a one-liner: not requiring a change of representation; reflective computation of the side conditions, allowing them all to be discharged with `reflexivity`; and automatic discovery of any arguments to the rule. We cover each of these separately.

Doing Without a New Representation of Strings

Recall from Section 5.3 that the first step in any splitter refinement, implemented as part of the `start honing parser using indexed representation` tactic, is to use an indexed representation of strings, where splitting a string only involves updating the indices into the original string. Naively, to implement the refinement strategy of this chapter, we'd either need to store a fresh table, somehow derived from the previous one, every time we split the string, or recompute the entire table from scratch on every invocation of the splitting method.

How do we get around computing a new table on splits? We pull the same trick here that we pulled on strings; we refer only to the table that is built from the initial string, and describe the correctness condition on the table in terms of arbitrary substrings of that string.

Fiat presents us with a goal involving a statement of the form "nondeterministically pick a list of splitting locations that is complete for the substring of `str` starting at `n` and ending at `m`, for the rule `pexpr+expr`." In code form, this is:

```
{ splits : [ℕ]
| split_list_is_complete
    G
    (substring n m str)
    nt
    ('ch'::its)
    splits }
```

This code, which matches our current implementation, doesn't quite allow us to handle what we claim to handle in this section; we should be able to handle any rule that starts with a nonterminal nt, such that the set of possible characters in any string which can be parsed as nt is disjoint from the set of possible first characters of any string parsed by the rest of the rule. This code only handles the case where the rest of the rule begins with a terminal.

The final refinement rule, which we use with setoid_rewrite, says that this is refined by a lookup into a suitably defined table:

```
(ret [case  List.nth n table None  of
       | Some idx  →  idx
       | None      →  dummy_value
      ])
 ⊆
{ splits : [ℕ]
| split_list_is_complete
    G
    (substring n m str)
    nt
    ('ch'::its)
    splits }
```

By phrasing the rule in terms of substring n m str, rather than in terms of an arbitrary string, the computation of the table is the same in every call to the splitter. All that remains is to lift the computation of the table to a location outside the recursive call to the parsing function; we plan to implement code to do this during the extraction phase soon.

Before moving on to the other components of making usage of this rule require only one line of code, we note that we make use of the essential property that removing characters from the end of the string doesn't add new locations where splitting could yield a valid parse; if a given location is the first '+' at the current level of parenthesization, this does not change when we remove characters from the end of the string.

Discharging the Side Conditions Trivially

To prove the correctness condition on the table, we need to know some things about the grammar that we are handling. In particular, we need to know that if we are trying to parse a string as a rule analogous to pexpr, then there can be no exposed '+' characters, and, furthermore, that every such parseable string has well-balanced parentheses. To allow discharging the side conditions trivially, we define a function that computes whether or not this is the case for a given nonterminal in a given grammar. We then prove that, whenever this function returns `true`, valid tables yield complete lists of splitting locations.

To make things simple, we approximate which grammars are valid; we require that every open parenthesis be closed in the same rule (rather than deeply nested in further nonterminals). In Haskell-like pseudocode, the function we use to check validity of a grammar can be written as:

```
grammar-and-nonterminal-is-valid : Grammar → Nonterminal → Bool
grammar-and-nonterminal-is-valid G nt
  := fold (&&) true (map (paren-balanced-hiding G) (G.(Lookup) nt))

pb' : Grammar → Nat → [Item] → Bool
pb' G n []          := (n == 0)
pb' G n (Nonterminal nt :: s)
  := fold (&&) (pb' G 0) (G.(Lookup) nt) && pb' G n s
pb' G n ('(' :: s) := pb' G (n + 1) s
pb' G n (')' :: s) := n > 0 && pb' G (n - 1) s
pb' G n (_ :: s)   := pb' G n s

paren-balanced G := pb' G 0

pbh' : Grammar → Nat → [Item] → Bool
pbh' G n []          := (n == 0)
pbh' G n (Nonterminal nt :: s)
  := fold (pbh' G 0) (G.(Lookup) nt) && pb' G n s
pbh' G n ('+' :: s) := n > 0 && pbh' G n s
pbh' G n ('(' :: s) := pbh' G (n + 1) s
pbh' G n (')' :: s) := n > 0 && pbh' G (n - 1) s
pbh' G n (_ :: s)   := pbh' G n s

paren-balanced-hiding G := pbh' G 0
```

There is one subtlety here, that was swept under the rug in the above code: this computation might not terminate! We could deal with this by memoizing this computation in much the same way that we memoized the parser to deal with potentially infinite parse trees. Rather than dealing with the subtleties of figuring out what to do when we hit repeated nonterminals, we perform the computation in two steps. First, we trace the algorithm above, building up a list of which nonterminals need to be paren-balanced, and which ones need to be paren-balanced-hiding. Second, we fold the computation over these lists, replacing the recursive calls for nonterminals with list membership tests.

Automatic Discovery of Arguments

Throughout this chapter, we've been focusing on the arithmetic-expression example. However, the exact same rule can handle S-expressions, with just a bit of generalization. There are two things to be computed: the binary operation and the parenthesis characters.[1]

We require that the first character of any string parsed by the nonterminal analogous to `+expr` be uniquely determined; that character will be the binary operator; we can reuse the code from Chapter 6 to compute this character and ensure that it is unique.

To find the parenthesis characters, we first compute a list of candidate character pairs: for each rule associated to the nonterminal analogous to `pexpr`, we consider the pair of the first character and the last character (assuming both are terminals) to be a candidate pair.[2] We then filter the list for characters which yield the conclusion that this rule is applicable to the grammar, according to the algorithm of the previous subsubsection. We then require, as a side condition, that the length of this list be positive.

---

[1] Currently, our code requires the binary operation to be exposed as a terminal in the rule we are handling. We plan on generalizing this to the grammars described in this chapter shortly.

[2] Again, generalizing this to characters hidden by nested nonterminals should be straightforward.

Chapter 8

Future Work

Parsing JavaScript The eventual target for this demonstration of the framework is the JavaScript grammar, and we aim to be competitive, performance-wise, with popular open-source JavaScript implementations. We plan to profile our parser against these on various test suites and examples of JavaScript code.

Generating Parse Trees We plan to eventually generate parse trees, and error messages, rather than just Booleans, in the complete pipeline. We have already demonstrated that this requires only small adjustments to the algorithm in the section on the dependently typed parser; what remains is integrating it with the Fiat code for refining splitters.

Validating Extraction By adapting ongoing work by Pit–Claudel et al., our parsers will be able to be compiled to verified Bedrock [7], and thus to assembly, within Coq. Currently, we use Coq's unverified extraction mechanism to turn our parsers into OCaml.

Picking Productions As mentioned in Section 4.2, our parsers perform poorly on large grammars with many rules. We plan to improve performance by parameterizing over an oracle to pick which rules to look at for a given nonterminal; much like the oracle for splitting, it should also be possible to handle a wide swath of cases automatically, and handle the remaining ones by refinement.

Common Subexpression Elimination: Lifting Computation out of Recursive Calls As mentioned briefly in Section 7.3.4, we plan to implement common subexpression elimination during the extraction phase. This will effectively memoize the computation of the table for splitting locations described in Chapter 7.

8.1 Future Work with Dependent Types

Recall from Chapter 3 that dependent types have allowed us to refine our parsing algorithm to prove its own soundness and completeness.

However, we still have some work left to do to clean up the implementation of the dependently typed version of the parser.

Formal extensionality/parametricity proof To completely finish the formal proof of completeness, as described in this thesis, we need to prove the parser extensionality axiom from Subsection 3.3.3. We need to prove that the parser does not make any decisions based on any arguments to its interface other than `split`, internalizing the obvious parametricity proof. Alternatively, we could hope to use an extension of Coq which materializes internally the metathoretic "free theorem" parametricity facts [3].

Even more self-reference We might also consider reusing the same generic parser to generate the extensionality proofs, by instantiating the type families for success and failure with families of propositions saying that all instantiations of the parser, when called with the same parsing problem, always return values that are equivalent when converted to Booleans. A more specialized approach could show just that `has_parse` agrees with `parse` on successes and with `has_no_parse` on failures:

$$\begin{aligned}
&\mathtt{T}_{\mathtt{success}}\ _\ _\ (\overline{\mathtt{s} \in \mathtt{nt}}) \\
&\quad := \mathtt{has_parse}\ \mathtt{nt}\ \mathtt{s} = \mathtt{true} \wedge \mathtt{parse}\ \mathtt{nt}\ \mathtt{s} \neq \mathtt{None} \\
&\mathtt{T}_{\mathtt{failure}}\ _\ _\ (\overline{\mathtt{s} \in \mathtt{nt}}) \\
&\quad := \mathtt{has_parse}\ \mathtt{nt}\ \mathtt{s} = \mathtt{false} \wedge \mathtt{has_no_parse} \neq \mathtt{inl}\ ()
\end{aligned}$$

Synthesizing dependent types automatically? Although finding sufficiently general (dependent) type signatures was a Herculean task before we finished the completeness proof and discovered the idea of using parallel parse traces, it was mostly straightforward once we had proofs of soundness and completeness of the simply typed parser in hand; most of the issues we faced involving having to figure out how to thread additional hypotheses, which showed up primarily at the very end of the proof, through the entire parsing process. Subsequently instantiating the types was also mostly straightforward, with most of our time and effort being spent writing transformations between nearly identical types that had slightly different hypotheses, e.g., converting a `Foo` involving strings shorter than s_1 into another analogous `Foo`, but allowing strings shorter than s_2, where s_1 is not longer than s_2. Our experience raises the question of whether it might be possible to automatically infer dependently typed generalizations of an algorithm, which subsume already-completed proofs about it, and perhaps allow additional proofs to be written more easily.

Further generalization Finally, we believe our parser could be generalized even further; the algorithm we have implemented is essentially an algorithm for inhabiting arbitrary inductive type families, subject to some well-foundedness, enumerability, and finiteness restrictions on the arguments to the type family. The interface we described in Chapter 3 is, conceptually, a composition of this inhabitation algorithm with recursion and inversion principles for the type family we are inhabiting (`ParseTreeOf` in this thesis). Our techniques for refining this algorithm so that it could prove itself sound and complete should therefore generalize to this viewpoint.

Appendix A

Selected Coq Code

A.1 A Fiat Goal After Trivial Rules Are Refined

For each grammar, the Fiat framework presents us with goals describing the unimplemented portion of the splitter for this particular grammar. For example, the goal for the grammar that parses arithmetic expressions involving plusses and parentheses, after taking care of the trivial obligations that we describe in Chapter 5, looks like this:

```
1 focused subgoals (unfocused: 3)
, subgoal 1 (ID 3491)

r_n : string * (nat * nat)
n : item ascii * production ascii
H := ?88 : hiddenT
============================
 refine
   (ls <- If ([Nonterminal "pexpr"] =p fst n :: snd n)
            || (([Nonterminal "expr"] =p fst n :: snd n)
            || ([Nonterminal "number"] =p fst n :: snd n)
            || (([Terminal ")"] =p fst n :: snd n)
            || ([Terminal "0"] =p fst n :: snd n)
            || ([Terminal "1"] =p fst n :: snd n)
            || ([Terminal "2"] =p fst n :: snd n)
            || ([Terminal "3"] =p fst n :: snd n)
            || ([Terminal "4"] =p fst n :: snd n)
            || ([Terminal "5"] =p fst n :: snd n)
            || ([Terminal "6"] =p fst n :: snd n)
            || ([Terminal "7"] =p fst n :: snd n)
            || ([Terminal "8"] =p fst n :: snd n)
```

```
            || ([Terminal "9"] =p fst n :: snd n)
            || ([Nonterminal "digit"] =p fst n :: snd n)))
         Then ret [ilength r_n]
         Else (If [Nonterminal "pexpr"; Terminal "+";
                  Nonterminal "expr"] =p fst n :: snd n
         Then {splits : list nat
              | ParserInterface.split_list_is_complete
                plus_expr_grammar
                (string_of_indexed r_n)
                (Nonterminal "pexpr")
                [Terminal "+"; Nonterminal "expr"]
                splits}
         Else
           ret [If ([Terminal "+"; Nonterminal "expr"]
                       =p fst n :: snd n)
                   || ([Terminal "("; Nonterminal "expr"; Terminal ")"]
                       =p fst n :: snd n)
                   || ([Nonterminal "digit"; Nonterminal "number"]
                       =p fst n :: snd n)
                Then 1
                Else (If [Nonterminal "expr"; Terminal ")"]
                            =p fst n :: snd n
                Then  pred (ilength r_n)
                Else (If [Nonterminal "number"]
                            =p fst n :: snd n
                Then ilength r_n
                Else 0))]);
  ret (r_n, ls)) (H r_n n)
```

A.2 Coq Code for the First Refinement Step

The general code for computing the goal the user is presented with, after `start honing parser using indexed representation`, is:

```
Definition expanded_fallback_list'
           (P : String -> item Ascii.ascii -> production Ascii.ascii
                -> item Ascii.ascii -> production Ascii.ascii
                -> list nat -> Prop)
           (s : T)
           (it : item Ascii.ascii) (its : production Ascii.ascii)
           (dummy : list nat)
: Comp (T * list nat)
  := (ls <- (forall_reachable_productions
               G
```

```
          (fun p else_case
           => If production_beq ascii_beq p (it::its) Then
                (match p return Comp (list nat) with
                   | nil => ret dummy
                   | _::nil => ret [ilength s]
                   | (Terminal _):: _ :: _ => ret [1]
                   | (Nonterminal nt):: p'
           => If has_only_terminals p' Then
                ret [(ilength s - Datatypes.length p')%natr]
              Else
                (option_rect
                  (fun _ => Comp (list nat))
                  (fun (n : nat) => ret [n])
                  { splits : list nat
                  | P
                      (string_of_indexed s)
                      (Nonterminal nt)
                      p'
                      it
                      its
                      splits }%comp
                  (length_of_any G nt))
                 end)
              Else else_case)
          (ret dummy));
ret (s, ls))%comp.
```

Bibliography

[1] José Bacelar Almeida, Nelma Moreira, David Pereira, and Simão Melo de Sousa. "Partial Derivative Automata Formalized in Coq". In: *Proceedings of the 15th International Conference on Implementation and Application of Automata*. CIAA'10. Winnipeg, MB, Canada: Springer-Verlag, 2011, pp. 59–68. ISBN: 978-3-642-18097-2. URL: `http://dl.acm.org/citation.cfm?id=1964285.1964292`.

[2] Aditi Barthwal and Michael Norrish. "Verified, Executable Parsing". In: *Proceedings of the 18th European Symposium on Programming Languages and Systems: Held As Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009*. ESOP '09. York, UK: Springer-Verlag, 2009, pp. 160–174. ISBN: 978-3-642-00589-3. DOI: `10.1007/978-3-642-00590-9_12`. URL: `http://dx.doi.org/10.1007/978-3-642-00590-9_12`.

[3] Jean-Philippe Bernardy and Moulin Guilhem. "Type-theory in Color". In: *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming*. ICFP '13. Boston, Massachusetts, USA: ACM, 2013, pp. 61–72. ISBN: 978-1-4503-2326-0. DOI: `10.1145/2500365.2500577`. URL: `http://doi.acm.org/10.1145/2500365.2500577`.

[4] Samuel Boutin. "Using reflection to build efficient and certified decision procedures". In: *Proc. TACS*. 1997.

[5] Janusz A. Brzozowski. "Derivatives of Regular Expressions". In: *Journal of the ACM (JACM)* 11.4 (Oct. 1964), pp. 481–494. ISSN: 0004-5411. DOI: `10.1145/321239.321249`. URL: `http://doi.acm.org/10.1145/321239.321249`.

[6] Nigel P Chapman. *LR Parsing: Theory and Practice*. CUP Archive, 1987.

[7] Adam Chlipala. "The Bedrock Structured Programming System: Combining Generative Metaprogramming and Hoare Logic in an Extensible Program Verifier". In: *Proc. ICFP*. ACM, 2013, pp. 391–402.

[8] The Coq Development Team. *The Coq Reference Manual, version 8.4*. Available electronically at `http://coq.inria.fr/doc`. 2012.

[9] H. B. Curry. "Functionality in Combinatory Logic". In: *Proceedings of the National Academy of Sciences of the United States of America* 20(11) (1934), pp. 584–590.

[10] David Delahaye. "A tactic language for the system Coq". In: *Logic for Programming and Automated Reasoning*. Springer. 2000, pp. 85–95.

[11] Benjamin Delaware, Clément Pit-Claudel, Jason Gross, and Adam Chlipala. "Fiat: Deductive synthesis of abstract data types in a proof assistant". In: *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM. 2015, pp. 689–700.

[12] Bryan Ford. "Parsing Expression Grammars: A Recognition-based Syntactic Foundation". In: *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '04. Venice, Italy: ACM, 2004, pp. 111–122. ISBN: 1-58113-729-X. DOI: `10.1145/964001.964011`. URL: `http://doi.acm.org/10.1145/964001.964011`.

[13] William A. Howard. "The formulae-as-types notion of construction". In: *To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*. Ed. by Jonathan P. Seldin and J. Roger Hindley. Original paper manuscript from 1969. Academic Press, 1980, pp. 479–490.

[14] Graham Hutton. "Higher-order Functions for Parsing". In: *Journal of Functional Programming* 2.3 (July 1992), pp. 323–343.

[15] Jacques-Henri Jourdan, François Pottier, and Xavier Leroy. "Validating LR(1) Parsers". In: *Proceedings of the 21st European Conference on Programming Languages and Systems*. ESOP'12. Tallinn, Estonia: Springer-Verlag, 2012, pp. 397–416. ISBN: 978-3-642-28868-5. DOI: `10.1007/978-3-642-28869-2_20`. URL: `http://dx.doi.org/10.1007/978-3-642-28869-2_20`.

[16] Ramana Kumar, Magnus O. Myreen, Michael Norrish, and Scott Owens. "CakeML: A Verified Implementation of ML". In: *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '14. San Diego, California, USA: ACM, 2014, pp. 179–191. ISBN: 978-1-4503-2544-8. DOI: `10.1145/2535838.2535841`. URL: `http://doi.acm.org/10.1145/2535838.2535841`.

[17] Matthew Might, David Darais, and Daniel Spiewak. "Parsing with Derivatives: A Functional Pearl". In: *Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming*. ICFP '11. Tokyo, Japan: ACM, 2011, pp. 189–195. ISBN: 978-1-4503-0865-6. DOI: `10.1145/2034773.2034801`. URL: `http://doi.acm.org/10.1145/2034773.2034801`.

[18] Robert C Moore. "Removing left recursion from context-free grammars". In: *Proceedings of the 1st North American chapter of the Association for Computational Linguistics conference*. Association for Computational Linguistics. 2000, pp. 249–255.

[19] Greg Morrisett, Gang Tan, Joseph Tassarotti, Jean-Baptiste Tristan, and Edward Gan. "RockSalt: better, faster, stronger SFI for the x86". In: *Proceedings of the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation*. PLDI '12. Beijing, China: ACM, 2012, pp. 395–404. ISBN: 978-

1-4503-1205-9. DOI: 10.1145/2254064.2254111. URL: http://doi.acm.org/10.1145/2254064.2254111.

[20] Magnus O. Myreen and Jared Davis. "A Verified Runtime for a Verified Theorem Prover". In: *Proceedings of the Second International Conference on Interactive Theorem Proving*. ITP'11. Berg en Dal, The Netherlands: Springer-Verlag, 2011, pp. 265–280. ISBN: 978-3-642-22862-9. URL: http://dl.acm.org/citation.cfm?id=2033939.2033961.

[21] Terence Parr, Sam Harwell, and Kathleen Fisher. "Adaptive LL(*) Parsing: The Power of Dynamic Analysis". In: *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*. ACM. 2014, pp. 579–598.

[22] Tom Ridge. "Simple, Functional, Sound and Complete Parsing for All Context-free Grammars". In: *Proceedings of the First International Conference on Certified Programs and Proofs*. CPP'11. Kenting, Taiwan: Springer-Verlag, 2011, pp. 103–118. ISBN: 978-3-642-25378-2. DOI: 10.1007/978-3-642-25379-9_10. URL: http://dx.doi.org/10.1007/978-3-642-25379-9_10.

[23] Elizabeth Scott and Adrian Johnstone. "GLL Parsing". In: *Proceedings of the Ninth Workshop on Language Descriptions Tools and Applications*. LDTA '09. 2009, pp. 177–189.

[24] Masaru Tomita. *Efficient Parsing for Natural Language: A fast algorithm for practical systems*. Vol. 8. Springer Science & Business Media, 2013.