TODO: Fill in acknowledgements in cover.tex

TODO: Move related work to end, so it flows better as explaining various other ways of parsing? Or introduce CFGs earlier.

TODO: Fill in related work with detailed explanations of various ways of writing parsers

TODO: Splitter now returns numbers, not strings

TODO: Explain how soundness can be done without parser extensionality, at the cost of algorithmic complexity elsewhere.

TODO: Find a citation for Fiat, test with [?]

TODO: (Optional) Section on showing that parser has "reasonable" performance on grammars with non-brute-force splitter (by using arrays and native strings)

TODO: (Really Optional) Section on building parse trees, not just recognizers)

QUESTION FOR ADAM: Should I include an appendix of all of the code of Fiat and parsers that is used, rendered verbatim?

A Formally Verified Parser for a JavaScript Subset in a New 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
July 9, 2015

Certified by .....
Adam Chlipala
Associate Professor
Thesis Supervisor

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

A Formally Verified Parser for a JavaScript Subset in a New Extensible Framework for Synthesizing Efficient Verified Parsers

by
Jason S. Gross

Submitted to the Department of Electrical Engineering and Computer Science
on July 9, 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 framework for synthesizing efficient, verified parsers by refinement, and a demonstration of this framework by synthesizing a JavaScript parser competitive with existing open-source parsers. 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

Acknowledgments

This is the acknowledgements section. You should replace this with your own acknowledgements.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

TODO: Very basic Background on parsers, grammars

1.1 Related Work

The field of parsing is one of the most venerable in computer-science. Still with us are a variety of parsing approaches born in times of much more severe constraints on memory and processor speed, including various flavors of LR parsers, which apply only to strict subsets of the context-free grammars, to guarantee ability to predict which production applies based on finite look-ahead into a string. However, despite rumors to the contrary, the field of parsing is far from dead. In the twentieth century, the functional-programming world experimented with a variety of approaches to *parser combinators* [5], where parsers are higher-order functions built from a small set of typed combinators. In the twenty-first century alone, a number of new parsing approaches have been proposed or popularized, including parsing expression grammars (PEGs) [4], derivative-based parsing [8], and GLL parsers [12].

However, our approach is essentially the same, algorithmically, as the one that Ridge demonstrated with a verified parser-combinator system [11], taking naive recursive-descent parsing and adding a layer to prune duplicative calls to the parser. His proof was carried out in HOL4, necessarily without using dependent types. Our new work may be interesting for the aesthetic appeal of our unusual application of dependent types to get the parser to generate some of its own soundness proof. Ridge's parser also has worst-case $O(n^5)$ running time in the input-string length. In the context of our verified implementation, we plan to explore a variety of optimizations based on clever, grammar-specific choices of string-splitter functions, which should have a substantial impact on the run-time cost of parsing some relevant grammars, and

which we conjecture will not require any changes to the development presented in this paper.

A few other past projects have verified parsers with proof assistants, applying to derivative-based parsing [1] and SLR [2] and LR(1) [6] parsers. Several projects have used proof assistants to apply verified parsers within larger programming-language tools. RockSalt [9] 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 [10] and CakeML [7] language implementations include verified parsers, handling Lisp and ML languages, respectively.

Our final parser derivation relies on a relational parametricity property for polymorphic functions in Coq's type theory Gallina. With Coq as it is today, we need to prove this property manually for each eligible function, even though we can prove metatheoretically that it holds for them all. Bernardy and Guilhem [3] have shown how to extend type theories with support for materializing "free theorem" parametricity facts internally, and we might be able to simplify our implementation using such a feature. **TODO:** Flesh this out

1.2 What's New

TODO: Fill out what's new

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

Start with naive recursive-descent parsing. We ensure termination via memoization, a la [11]. We parameterize the parser on a "splitting oracle", which describes how to recurse. <section citation> As far as we can tell, the idea of factoring the algorithmic complexity like this is new. <section citation>

We use fiat to incrementally build efficient parsers by refinement. <section citation>

The idea of reusing the parsing algorithm to prove its own completeness, by parsing parse trees rather than strings, is not found in the literature, to the authors' knowledge. <section citation>

Chapter 2

Parsing Context Free Grammars

We begin with an overview of the general setting, and a description of our approach to parsing.

2.1 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.

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

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 paper, 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.

Note that, for soundness and completeness, there is a trivial splitter: it returns a list of all numbers between 0 and the length of the string.

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 sections **TODO:** secref, we focus only on correctness, we set up the framework we use to achieve efficiency in section **TODO:** secref, and we demonstrate the use of the framework in sections **TODO:** secref.

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 being a sufficiently simple language to make proofs relatively straightforward. For example, to achieve linear parse time on the `(ab)`* grammar, we could have a splitter that, when trying to parse `'`c_1`'` $\cdot$ `'`c_2`'` $\cdot$ `s` as `ab(ab)`*, splits the string into (`'`c_1`'`, `'`c_2`'`, `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 algorithm, the logic program, is not guaranteed to terminate.

2.1.1 Infinite regress

Since we have programmed our parser in Coq, our program must be terminating by construction. However, naive recursive-descent parsers do not always terminate!

To see how such parsers can diverge, consider the following example. When defining the grammar `(ab)`*, perhaps we give the following production rules:

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

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

Now, let us try to parse the string `"ab"` as `(ab)*`:

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

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 `split : String → [String × 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)
```

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

2.1.2 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. Note that

this strategy only works for grammars with finite sets of nonterminals, in line with most formalizations of context-free grammars. 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'd 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 4-2 in Subsection 4.3.2.

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 this paper to describing an elegant approach to proving completeness. Ridge [11] 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.

2.2 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.

2.2.1 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.

Example: `(ab)`*

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:

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

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

2.2.2 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 `ParseTreeOf`s and `ParseItemsTreeOf`s 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.

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

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

For brevity, we will sometimes use the notation $\overdashed{\texttt{s} \in \texttt{X}}$ 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

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

Chapter 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 → String → 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:

```
has_parse_sound : (nt : Nonterminal) → (s : String)
                  → has_parse nt s = true
                  → ParseTreeOf nt s
has_parse_complete : (nt : Nonterminal) → (s : String)
                     → ParseTreeOf nt s
                     → has_parse nt s = true
```

For any parser

```
parse : Nonterminal → String → 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 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 4

Completeness, Soundness, and Parsing Parse Trees

4.1 Proving Completeness: Conceptual Approach

Recall from Subsection 2.1.2 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 mini-

mal parse trees, which will provide motivation for the type signatures of our parser's arguments.

4.2 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 2.2.2, 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. Letting $\texttt{unseen}' \coloneqq \texttt{unseen} - \{\texttt{nt}\}$, we have the following, where we interpret the $<$ relation on strings in terms of lengths.

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

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

$$\begin{aligned}
&\texttt{min_parse}_{[]} : \texttt{MinItemsTreeOf}\ \texttt{s}_0\ \texttt{unseen}\ []\ \texttt{""} \\
&\texttt{min_parse}_{::} : \texttt{s}_1\texttt{s}_2 \leq \texttt{s}_0 \\
&\qquad\qquad \to \texttt{MinParseTreeOf}\ \texttt{s}_0\ \texttt{unseen}\ \texttt{it}\ \texttt{s}_1 \\
&\qquad\qquad \to \texttt{MinItemsTreeOf}\ \texttt{s}_0\ \texttt{unseen}\ \texttt{its}\ \texttt{s}_2 \\
&\qquad\qquad \to \texttt{MinItemsTreeOf}\ \texttt{s}_0\ \texttt{unseen}\ (\texttt{it} :: \texttt{its})\ \texttt{s}_1\texttt{s}_2
\end{aligned}$$

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

Once again, for brevity, we will sometimes use the notation $\overset{\cdots\cdots}{\texttt{s} \in \texttt{X}}{}^{<(\texttt{s}_0,\texttt{v})}$ to denote both `MinParseTreeOf` $\texttt{s}_0$ `v X s` and `MinItemsTreeOf` $\texttt{s}_0$ `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.

4.3 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 4-1. 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 paper. The interface we present here fixes `String`, for conciseness.

Since we want to be able to specialize our parser to return either `Bool` or `option ParseTree`,

We use `ParseQuery` to denote the type of all propositions like "`"a" ∈ '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 s_0 and a list of nonterminals `unseen`. We assume we are given $\texttt{unseen}_0 : \texttt{[Nonterminal]}$.

$$\texttt{T}_{\texttt{success}},\ \texttt{T}_{\texttt{failure}} : \texttt{String} \to \texttt{[Nonterminal]} \to \texttt{ParseQuery} \to \textbf{Type}$$
$$\texttt{split} : \texttt{String} \to \texttt{[Nonterminal]} \to \texttt{ParseQuery} \to [\mathbb{N}]$$

$$\texttt{terminal\_success} : (\texttt{ch} : \texttt{Char}) \to \texttt{T}_{\texttt{success}}\ \ s_0\ \ \texttt{unseen}\ \ (\overbrace{\texttt{"ch"} \in \texttt{'ch'}})$$
$$\texttt{terminal\_failure} : (\texttt{ch} : \texttt{Char}) \to (\texttt{s} : \texttt{String}) \to \texttt{s} \neq \texttt{"ch"} \to \texttt{T}_{\texttt{failure}}\ \ s_0\ \ \texttt{unseen}\ \ (\overbrace{\texttt{s} \in \texttt{'ch'}})$$
$$\texttt{nil\_success} : \texttt{T}_{\texttt{success}}\ \ s_0\ \ \texttt{unseen}\ \ (\overbrace{\texttt{""} \in \epsilon})$$
$$\texttt{nil\_failure} : (\texttt{s} : \texttt{String}) \to \texttt{s} \neq \texttt{""} \to \texttt{T}_{\texttt{failure}}\ \ s_0\ \ \texttt{unseen}\ \ (\overbrace{\texttt{s} \in \epsilon})$$

$$\begin{aligned}\texttt{cons\_success} : {} & (\texttt{it} : \texttt{Item}) \to (\texttt{its} : \texttt{[Item]}) \to (s_1 : \texttt{String}) \to (s_2 : \texttt{String}) \\ & \to s_1 s_2 \leq s_0 \\ & \to \texttt{T}_{\texttt{success}}\ \ s_0\ \ \texttt{unseen}\ \ (\overbrace{s_1 \in \texttt{it}}) \\ & \to \texttt{T}_{\texttt{success}}\ \ s_0\ \ \texttt{unseen}\ \ (\overbrace{s_2 \in \texttt{its}}) \\ & \to \texttt{T}_{\texttt{success}}\ \ s_0\ \ \texttt{unseen}\ \ (\overbrace{s_1 s_2 \in \texttt{it} :: \texttt{its}})\end{aligned}$$
$$\begin{aligned}\texttt{cons\_failure} : {} & (\texttt{it} : \texttt{Item}) \to (\texttt{its} : \texttt{[Item]}) \to (\texttt{s} : \texttt{String}) \\ & \to \texttt{s} \leq s_0 \\ & \to \big(\forall\ (s_1, s_2) \in \texttt{split}\ \ s_0\ \ \texttt{unseen}\ \ (\overbrace{\texttt{s} \in \texttt{it} :: \texttt{its}}), \\ & \qquad \texttt{T}_{\texttt{failure}}\ \ s_0\ \ \texttt{unseen}\ \ (\overbrace{s_1 \in \texttt{it}}) + \texttt{T}_{\texttt{failure}}\ \ s_0\ \ \texttt{unseen}\ \ (\overbrace{s_2 \in \texttt{its}})\big) \\ & \to \texttt{T}_{\texttt{failure}}\ \ s_0\ \ \texttt{unseen}\ \ (\overbrace{\texttt{s} \in \texttt{it} :: \texttt{its}})\end{aligned}$$

$$\begin{aligned}\texttt{production\_success}_{<} : {} & (\texttt{its} : \texttt{[Item]}) \to (\texttt{nt} : \texttt{Nonterminal}) \to (\texttt{s} : \texttt{String}) \\ & \to \texttt{s} < s_0 \\ & \to (\texttt{p} : \text{a production mapping } \texttt{nt} \text{ to } \texttt{its}) \\ & \to \texttt{T}_{\texttt{success}}\ \ \texttt{s}\ \ \texttt{unseen}_0\ \ (\overbrace{\texttt{s} \in \texttt{its}}) \\ & \to \texttt{T}_{\texttt{success}}\ \ s_0\ \ \texttt{unseen}\ \ (\overbrace{\texttt{s} \in \texttt{nt}})\end{aligned}$$
$$\begin{aligned}\texttt{production\_success}_{=} : {} & (\texttt{its} : \texttt{[Item]}) \to (\texttt{nt} : \texttt{Nonterminal}) \to (\texttt{s} : \texttt{String}) \\ & \to \texttt{nt} \in \texttt{unseen} \\ & \to (\texttt{p} : \text{a production mapping } \texttt{nt} \text{ to } \texttt{its}) \\ & \to \texttt{T}_{\texttt{success}}\ \ s_0\ \ (\texttt{unseen} - \{\texttt{nt}\})\ \ (\overbrace{\texttt{s} \in \texttt{its}}) \\ & \to \texttt{T}_{\texttt{success}}\ \ s_0\ \ \texttt{unseen}\ \ (\overbrace{\texttt{s} \in \texttt{nt}})\end{aligned}$$
$$\begin{aligned}\texttt{production\_failure}_{<} : {} & (\texttt{nt} : \texttt{Nonterminal}) \to (\texttt{s} : \texttt{String}) \\ & \to \texttt{s} < s_0 \\ & \to \big(\forall\ (\texttt{its} : \texttt{[Item]})\ (\texttt{p} : \text{a production mapping } \texttt{nt} \text{ to } \texttt{its}), \texttt{T}_{\texttt{failure}}\ \ \texttt{s}\ \ \texttt{unseen} \\ & \to \texttt{T}_{\texttt{failure}}\ \ s_0\ \ \texttt{unseen}\ \ (\overbrace{\texttt{s} \in \texttt{nt}})\end{aligned}$$
$$\begin{aligned}\texttt{production\_failure}_{=} : {} & (\texttt{nt} : \texttt{Nonterminal}) \to (\texttt{s} : \texttt{String}) \\ & \to \texttt{s} = s_0 \\ & \to \big(\forall\ (\texttt{its} : \texttt{[Item]})\ (\texttt{p} : \text{a production mapping } \texttt{nt} \text{ to } \texttt{its}), \texttt{T}_{\texttt{failure}}\ \ s_0\ \ (\texttt{unse}\end{aligned}$$

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

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

To instantiate the parser as a Boolean recognizer, we instantiate everything trivially; we use the fact that $\top + \top \cong \texttt{Bool}$. 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 $\top$:

$$\begin{aligned}
&\texttt{T}_{\texttt{success}}\ _\ _\ _ := \top \\
&\texttt{T}_{\texttt{failure}}\ _\ _\ _ := \top \\
&\\
&\texttt{terminal_success}\ _\ _\ _ := () \\
&\texttt{terminal_failure}\ _\ _\ _\ _\ _ := () \\
&\texttt{nil_success}\ _\ _ := () \\
&\texttt{nil_failure}\ _\ _\ _\ _ := () \\
&\texttt{cons_success}\ _\ _\ _\ _\ _\ _\ _\ _\ _ := () \\
&\texttt{cons_failure}\ _\ _\ _\ _\ _\ _\ _ := () \\
&\texttt{production_success}_{<}\ _\ _\ _\ _\ _\ _\ _\ _\ _ := () \\
&\texttt{production_success}_{=}\ _\ _\ _\ _\ _\ _\ _\ _\ _ := () \\
&\texttt{production_failure}_{<}\ _\ _\ _\ _\ _\ _ := () \\
&\texttt{production_failure}_{=}\ _\ _\ _\ _\ _\ _ := () \\
&\texttt{production_failure}_{\notin}\ _\ _\ _\ _\ _\ _ := ()
\end{aligned}$$

To instantiate our parser so that it returns `option ParseTree` (rather, the dependently typed flavor, `ParseTreeOf`), we take advantage of the isomorphism $T + \top \cong \texttt{option}\ T$. 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.

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

$$\mathtt{terminal\_success}\ \_\ \_\ \mathtt{ch} \coloneqq (\mathtt{'ch'})$$
$$\mathtt{nil\_success}\ \_\ \_ \coloneqq \overline{\mathtt{""} \in \epsilon}$$
$$\mathtt{cons\_success}\ \_\ \_\ \mathtt{it}\ \mathtt{its}\ \mathtt{s_1}\ \mathtt{s_2}\ \_\ \mathtt{d_1}\ \mathtt{d_2} \coloneqq \frac{\frac{\mathtt{d_1}}{\mathtt{s_1} \in \mathtt{it}} \qquad \frac{\mathtt{d_2}}{\mathtt{s_2} \in \mathtt{its}}}{\mathtt{s_1 s_2} \in \mathtt{it} :: \mathtt{its}}$$
$$\mathtt{production\_success_<}\ \_\ \_\ \mathtt{it}\ \mathtt{nt}\ \mathtt{s}\ \_\ \mathtt{p}\ \mathtt{d} \coloneqq \frac{\frac{\mathtt{d}}{\mathtt{s} \in \mathtt{its}}}{\mathtt{s} \in \mathtt{nt}}\ (\mathtt{p})$$
$$\mathtt{production\_success_=}\ \_\ \_\ \mathtt{it}\ \mathtt{nt}\ \mathtt{s}\ \_\ \mathtt{p}\ \mathtt{d} \coloneqq \frac{\frac{\mathtt{d}}{\mathtt{s} \in \mathtt{its}}}{\mathtt{s} \in \mathtt{nt}}\ (\mathtt{p})$$

What remains is to instantiate 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 $\bot$ is the empty type (equivalently, the false proposition).

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

Using $\lightning$ to denote deriving a contradiction, we can unenlighteningly instantiate the arguments as

$$\mathtt{terminal\_success}\ \_\ \_\ \_ \coloneqq ()$$
$$\mathtt{terminal\_failure}\ \_\ \_\ \_\ \_\ \_ \coloneqq \lightning$$
$$\mathtt{nil\_success}\ \_\ \_ \coloneqq ()$$
$$\mathtt{nil\_failure}\ \_\ \_\ \_\ \_ \coloneqq \lightning$$
$$\mathtt{cons\_success}\ \_\ \_\ \_\ \_\ \_\ \_\ \_\ \_\ \_ \coloneqq ()$$
$$\mathtt{cons\_failure}\ \_\ \_\ \_\ \_\ \_\ \_\ \_ \coloneqq \lightning$$
$$\mathtt{production\_success_<}\ \_\ \_\ \_\ \_\ \_\ \_\ \_\ \_ \coloneqq ()$$
$$\mathtt{production\_success_=}\ \_\ \_\ \_\ \_\ \_\ \_\ \_\ \_\ \_ \coloneqq ()$$
$$\mathtt{production\_failure_<}\ \_\ \_\ \_\ \_\ \_\ \_ \coloneqq \lightning$$
$$\mathtt{production\_failure_=}\ \_\ \_\ \_\ \_\ \_\ \_ \coloneqq \lightning$$
$$\mathtt{production\_failure_{\notin}}\ \_\ \_\ \_\ \_\ \_\ \_ \coloneqq \lightning$$

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 `s` $\neq$ `""` hypothesis of `nil_failure` contradicts the equalities implied by the type signature of $\texttt{min\_parse}_{[]}$, and the use of `[]` contradicts the equality implied by the use of `it::its` in the type signature of $\texttt{min\_parse}_{[]}$. Similarly, the `s` $\neq$ `"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 $\texttt{min\_parse}_{\texttt{'ch'}}$.

4.3.1 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 4.1 and defined formally in Section 4.2. 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:

```
split _ _ (s ∈ it :: its) :=
  case parse_tree_data s of
  | p1: s1∈it   p2: s2∈its  ⊢ s1s2 ∈ it :: its => [(deloop p1, deloop p2)]
  | _ => ↯
split _ _ _ := []
```

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 match 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 $\mathtt{T}_{\mathtt{success}} + \mathtt{T}_{\mathtt{failure}}$, for some $\mathtt{T}_{\mathtt{success}}$ and $\mathtt{T}_{\mathtt{failure}}$. 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 $\mathtt{T}_{\mathtt{failure}}$ with $\bot$, the uninhabited type; and then apply a simple fix-up wrapper on top. Thus, we instantiate the general parser like so:

```
T_success s0 unseen (d : s ∈ X) := s ∈ X <(s0,unseen)
T_failure _ _ _ := ⊥
```

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 (`s` $\neq$ `"ch"` and `s` $\neq$ `""`, 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 $\bot + \bot \to \bot$, which is trivial. In the $\mathtt{production\_failure}_{<}$ and $\mathtt{production\_failure}_{=}$ cases, it is sufficient to note that every nonterminal is mapped by some production to some sequence of items. Finally, to instantiate the $\mathtt{production\_failure}_{\notin}$ case, we need to appeal to the `Unloopy`-ness of the tree to deduce that `nt` $\in$ `unseen`. Then we can derive $\bot$ from the hypothesis that `nt` $\notin$ `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.

4.3.2 Example

In Subsection 2.1.1, 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 4-2 contains the fully general implementation of our parser, modulo type signatures.

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

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

Then we consider minimizing the parse tree:

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

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

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

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"` ∈ `(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

```
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

$$\forall\, x' \in (x :: xs),\ \mathtt{T_{failure}}\ \_\ \_\ (\mathtt{take}_{x'}\ s \in it) + \mathtt{T_{failure}}\ \_\ \_\ (\mathtt{drop}_{x'}\ s \in its)$$

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 4-2: Pseudo-Implementation of our parser. We take the convention that dependent indices to functions (e.g., unseen) are implicit.

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 4.4.2. The only such production in this case is the one that lines up with the production used in the parse tree, labeled (ab)*(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 ("ab" ∈ (ab)*), the label of the root node, we promote the deepest one. Letting T' denote the tree

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

   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 "ab" ∈ (ab)* that was returned by deloop.

4. We invoke parse on each of the items in the sequence of items associated to (ab)* via the rule ((ab)*(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 s < s_0. This check succeeds, for "ab" is shorter than "abab". We thus reset unseen and s_0; calls made by this stack frame will pass $\texttt{unseen}_0 \equiv$ [(ab)*] for unseen, and s ≡ "ab" for 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

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

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

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

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

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

4.3.3 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 *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`.

4.3.4 Putting it all together

Now we have parsers returning the following types:

$$\begin{aligned}\texttt{has\_parse} &: \texttt{Nonterminal} \to \texttt{String} \to \texttt{Bool}\\ \texttt{parse} &: (\texttt{nt} : \texttt{Nonterminal}) \to (\texttt{s} : \texttt{String})\\ &\quad \to \texttt{option}\ (\texttt{ParseTreeOf}\ \texttt{nt}\ \texttt{s})\\ \texttt{has\_no\_parse} &: (\texttt{nt} : \texttt{Nonterminal}) \to (\texttt{s} : \texttt{String})\\ &\quad \to \top + (\texttt{MinParseTreeOf}\ \texttt{nt}\ \texttt{s} \to \bot)\\ \texttt{min\_parse} &: (\texttt{nt} : \texttt{Nonterminal}) \to (\texttt{s} : \texttt{String})\\ &\quad \to \texttt{ParseTreeOf}\ \texttt{nt}\ \texttt{s}\\ &\quad \to \texttt{MinParseTreeOf}\ \texttt{nt}\ \texttt{s}\end{aligned}$$

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:

```
parse_full : (nt : Nonterminal) → (s : String)
           → ParseTreeOf nt s + (ParseTreeOf nt s → ⊥)
parse_full nt s :=
  case parse nt s, has_no_parse nt s of
  | Some d, _      → inl d
  | _     , inr nd → inr (nd ∘ min_parse)
  | _     , _      → ↯
```

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`.

4.4 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.

4.4.1 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 highropes 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 highropes 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.

4.4.2 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 paper, 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 4.3.2, 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.

4.4.3 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 4.3.2 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 $(\mathtt{s} \in \mathtt{nt})$ to appear one extra time along a path if, the first time it appears, its parent node's label, $(\mathtt{s}' \in \mathtt{nt}')$, satisfies $\mathtt{s} < \mathtt{s}'$. That is, whenever the string being parsed shrinks, the first nonterminal the shrunken string is parsed as may be duplicated once before shrinking the string again.

Chapter 5

Refining Splitters by Fiat

5.1 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 parametrized 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 paper, 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 1.2, 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 [**?**], 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.

5.2 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!). **TODO:** CITATION NEEDED 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)).

Here is an example of hitting the worst-case scenario. **TODO:** Is this actually possible?

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.

5.3 Introducing Fiat

5.3.1 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 [**?**] 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.

5.3.2 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 non-deterministic choice operator; we define the following combinators:

$\mathtt{x} \leftarrow \mathtt{c};\ \ \mathtt{c}'$	Run $\mathtt{c}$ and store the result in $\mathtt{x}$; continue with $\mathtt{c}'$, which may mention $\mathtt{x}$
$\mathtt{c};;\ \ \mathtt{c}'$	Run $\mathtt{c}$. If it terminates, throw away the result, and run $\mathtt{c}'$
$\mathtt{ret}\ \ \mathtt{x}$	A computation that immediately returns the value $\mathtt{x}$
$\{\mathtt{x} \mid \mathtt{P}(\mathtt{x})\}$	Nondeterministically choose a value of $\mathtt{x}$ satisfying $\mathtt{P}$. 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 $\mathtt{c}'$ *refines* a computation $\mathtt{c}$, written $\mathtt{c}' \subseteq \mathtt{c}$, if every value that $\mathtt{c}'$ can compute to, $\mathtt{c}$ can also compute to. We freely generate the relation "the computation $\mathtt{c}$ can compute to the value $\mathtt{v}$", written $\mathtt{c} \rightsquigarrow \mathtt{v}$, by the rules:

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

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. 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, if we are looking to split the string at the location of the first `'+'` character, we might first pick the list of "valid locations to split at", and then refine that into a computation that picks the location of the first `'+'`, and returns the singleton list containing that, and then replace the remaining bit of nondeterminism with a computation of the location of the first `'+'`.

The key 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.

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

5.4 Optimizations

5.4.1 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.

5.4.2 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 paper. 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.

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 `(ab)*` grammar looks like this: **TODO:** <INSERT GOAL>. To get to this goal, we write this code: **TODO:** <COQ CODE HERE, with comments describing what each line does> We thus have to describe how to split a string for the rules **TODO:** <RULES HERE>, and provide proofs that these splitting strategies are complete. We begin the next section with this splitting strategy.

Chapter 6

fixed length nonterminals, parsing (ab)*; parsing #s; parsing #, ()

- Goals
  - Explore the framework
  - Demonstrate that we can implement the "obvious" rules to handle a large swath of CFG rules
- At a Glance
  - There are some strategies that are obvious enough that it's easy for the computer to decide whether or not it's good to apply them. For example, if a rule starts with a terminal, then we should split off one character, because terminals are always single characters. These rules are enough to parse some simple grammars, such as the regular expression grammar (ab)*; the grammar accepting numbers; and the grammar accepting parenthesized numbers. [GIVE GRAMMARS HERE] In this section, we explain how we parse these grammars.
- The splitting strategy: if all strings parsed by a given item are the same length, then we can always split the string when faced with that nonterminal.
  - Walk through an example of parsing "abab" as "(ab)*", describing the splitting at each point.
  - Another example: "((123))"
  - For (ab)*, the item "a", and the item "b" only parse strings of length 1. Thus when asked for the split for "a", then "b(ab)*", we can split after one character, and similarly after "b".
  - For numbers with parentheses (of which numbers are a subgrammar), digits always have length 1, so we can split after the first character.

- Implementation as a refinement rule:
  - Describe the obligation Fiat presents us with for the splitter for (#) [CODE HERE]
  - Describe how each rule is handled
  - For the relevant rules, we can compute the length at compile time. Here is the algorithm. <Coq code here>
  - To actually make use of this, we must satisfy the correctness criterion. This is what refinement means. We relate the length to the parse trees by a few correctness criteria.
    - Note that we need to use only well-founded recursion.
  - We provide a decision procedure for the validity of this rule.
- In practice, we don't actually need to rewrite it; because it's never suboptimal to apply this rule (returning a single split location is just about the best we can do (TODO: handle invalid parses and backtracking better)), so we do it automatically, baking it into the initial goal, along with the indexed representation change (explain)

Chapter 7

disjoint items, parsing #, +

- Goals
  - More exploration
  - Demonstrate a slightly less obvious strategy that handles even more rules
- [Intro]
- The splitting strategy: if the set of all characters in one item are disjoint from the set of possible first characters of the next item, then we can split at either the first character not in the first set, or at the first character that is in the second set.
  - For example, if the nonterminal "number" accepts 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, and the nonterminal "binop" accepts +, -, then we can either look for the first non-digit and split there, or we can look for the first + or -, and split there.
- Write down the grammars we want to handle.
- We compute the sets reflectively. Here is the algorithm. Again, note the well-founded recursion. <Coq code here>
- We relate the computation to the parse trees by a few correctness criteria, similar to above.
  - Describe the structures and lemmas that go into it.
- This rule is not applied automatically, because we have this choice about what sets to look for. In the future, we might pick whichever set is smaller, and do that (but perhaps we think one is more likely than the other?) Instead we use `setoid_rewrite`, with [reflexivity] to solve the side-conditions.
- Example: Parse "1+2+3"

Chapter 8

Parsing well-parenthesized expressions

8.1 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.

8.2 Grammars we can parse

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

Parenthesized addition: **QUESTION FOR ADAM:** Should I write these as inference rules, or as standard CFG presentations?

$$\frac{s \in \epsilon}{s \in \texttt{number?}}\ (\texttt{number?-}\epsilon) \qquad \frac{s \in \texttt{number}}{s \in \texttt{number?}}\ (\texttt{number?})$$

$$\frac{s_0 \in \texttt{digit} \qquad s_1 \in \texttt{number?}}{s_0 s_1 \in \texttt{number}}\ (\texttt{number})$$

$$\frac{s_0 \in \texttt{'('} \qquad s_1 \in \texttt{expr} \qquad s_2 \in \texttt{')'}}{s \in \texttt{pexpr}}\ (\texttt{pexpr})$$

$$\frac{s \in \texttt{number}}{s \in \texttt{expr}}\ (\texttt{expr-number}) \qquad \frac{s_0 \in \texttt{pexpr} \qquad s_1 \in \texttt{+expr}}{s \in \texttt{expr}}\ (\texttt{expr-pxpr})$$

$$\frac{s \in \epsilon}{s \in \texttt{+expr}}\ (\epsilon\texttt{-+expr}) \qquad \frac{s_0 \in \texttt{'+'} \qquad s_1 \in \texttt{expr}}{s_0 s_1 \in \texttt{+expr}}\ (\texttt{+expr})$$

TODO: Pick one of these

Or, in the standard presentation:

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

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:

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

8.3 The Splitting Strategy

8.3.1 The Main Idea

The only rule not already handled 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 parenthetization is. If we can compute an appropriate lookup table in time linear in the length of the string, then our splitter overall with be linear.

8.3.2 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 parenthetization. To compute this location we keep a list of the location of next `'+'` at every level of parenthetization. For example, for the string "((1+2)+3)+4", we have the following lists at each point in the string: **TODO:** Insert computation here

- The generated table is
  "5, 1, 0, -, -, 0 . , 4, 3, 2, 1 "

8.3.3 The Code

TODO: Insert Haskell-like code here

Optimization: Compute based on original string, lookup based on indices, don't need to compute substrings.

8.3.4 The Correctness Proof

To use this as a refinement rule, we need to prove it complete. To do this, we prove **TODO:** explain correctness criterion and insert relation of paren-balanced-hiding on a string to properties of grammar.

Chapter 9

Future work

- Grammars and efficiency: 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. <lookup list of JS implementations> We plan to profile our parser against these on <lookup test suite>
  - Description of anticipated challenges, based on the JavaScript grammar <lookup JS grammar>
- 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.
- Validating extraction
  - By adapting <Clement's work>, our parsers will be able to be compiled to verified bedrock/assembly, within Coq

9.1 Future work with dependent types

Recall from Chapter 4 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 paper, we need to prove the parser extensionality axiom from Subsection 4.3.3. 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, as mentioned above, we could hope to use an extension of Coq with internalized parametricity [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:

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

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 is, conceptually, a composition of this inhabitation algorithm with recursion and inversion principles for the type family we are inhabiting (`ParseTreeOf` in this

paper). Our techniques for refining this algorithm so that it could prove itself sound and complete should therefore generalize to this viewpoint.

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, pages 59–68, Berlin, Heidelberg, 2011. Springer-Verlag.

[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, pages 160–174, Berlin, Heidelberg, 2009. Springer-Verlag.

[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, pages 61–72, New York, NY, USA, 2013. ACM.

[4] 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, pages 111–122, New York, NY, USA, 2004. ACM.

[5] Graham Hutton. Higher-order functions for parsing. *Journal of Functional Programming*, 2(3):323–343, July 1992.

[6] 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, pages 397–416, Berlin, Heidelberg, 2012. Springer-Verlag.

[7] 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, pages 179–191, New York, NY, USA, 2014. ACM.

[8] 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, pages 189–195, New York, NY, USA, 2011. ACM.

[9] 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, pages 395–404, New York, NY, USA, 2012. ACM.

[10] 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, pages 265–280, Berlin, Heidelberg, 2011. Springer-Verlag.

[11] 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, pages 103–118, Berlin, Heidelberg, 2011. Springer-Verlag.

[12] Elizabeth Scott and Adrian Johnstone. GLL parsing. In *Proceedings of the Ninth Workshop on Language Descriptions Tools and Applications*, LDTA '09, pages 177–189, 2009.