Principled Procedural Parsing

Nicolas Laurent

August 2019

Thesis submitted in partial fulfillment of the requirements for the degree of Doctor of Applied Science in Engineering

Institute of Information and Communication Technologies,
Electronics and Applied Mathematics (ICTEAM)

Louvain School of Engineering (EPL)

Université catholique de Louvain (UCLouvain)

Louvain-la-Neuve

Belgium

Thesis Committee

Prof. Kim **Mens**, *Advisor*	UCLouvain/ICTEAM, Belgium
Prof. Charles **Pecheur**, *Chair*	UCLouvain/ICTEAM, Belgium
Prof. Peter **Van Roy**	UCLouvain/ICTEAM Belgium
Prof. Anya Helene **Bagge**	UIB/II, Norway
Prof. Tijs **van der Storm**	CWI/SWAT & UG, The Netherlands

Contents

Contents **3**

1 Introduction **7**
1.1 Parsing . . . 7
1.2 Inadequacy: Flexibility Versus Simplicity . . . 8
1.3 The Best of Both Worlds . . . 10
1.4 The Approach: Principled Procedural Parsing . . . 13
1.5 Autumn: Architecture of a Solution . . . 15
1.6 Overview & Contributions . . . 17

2 Background **23**
2.1 Context Free Grammars (CFGs) . . . 23
2.1.1 The CFG Formalism . . . 23
2.1.2 CFG Parsing Algorithms . . . 25
2.1.3 Top-Down Parsers . . . 26
2.1.4 Bottom-Up Parsers . . . 30
2.1.5 Chart Parsers . . . 33
2.2 Top-Down Recursive-Descent Ad-Hoc Parsers . . . 35
2.3 Parser Combinators . . . 36
2.4 Parsing Expression Grammars (PEGs) . . . 39
2.4.1 Expressions, Ordered Choice and Lookahead . . . 39
2.4.2 PEGs and Recursive-Descent Parsers . . . 42
2.4.3 The Single Parse Rule, Greed and (Lack of) Ambiguity . . . 43
2.4.4 The PEG Algorithm . . . 44
2.4.5 Packrat Parsing . . . 45
2.5 Expression Parsing . . . 47
2.6 Error Reporting . . . 50
2.6.1 Overview . . . 51
2.6.2 The Furthest Error Heuristic . . . 52
2.6.3 Associating Errors With Custom Messages . . . 53

2.6.4 Error Recovery 54
2.7 Further Related Work 59

3 Autumn's Basics **61**
3.1 Introductory Example 62
3.1.1 `DSL`, `rule`, parsers and combinators 66
3.1.2 Whitespace Handling & String Literals 67
3.1.3 `lazy` and `sep` 67
3.1.4 Launching the Parse 68
3.2 `Parser` and `Parse` 68
3.3 Building An Abstract Syntax Tree (AST) 72
3.3.1 AST-Building Combinators 74
3.3.2 Value Stack as Context 76
3.4 Beyond Basics 76
3.5 Java 8 Grammar 77

4 Infix Expression Parsing **79**
4.1 Outline 80
4.2 Grammatical Encodings of Expression Syntax 82
4.3 The Semantics of Left-Recursion 84
4.4 The Semantics of Left-Associativity 86
4.4.1 Naive Formulation 87
4.4.2 Ambiguous Recursion 87
4.4.3 Restating the Problem 88
4.4.4 A Pragmatic Way Out 89
4.4.5 Related Work 90
4.5 Transparent Left-Recursion Handling in PEG 91
4.5.1 The Left-Recursive Algorithm 91
4.5.2 Transparent Left-Recursion in Autumn 94
4.5.3 Performance Woes 97
4.6 Automatic Left-Recursion Discovery 98
4.7 Expression Clusters 102
4.8 Left-Associativity via Left-Folding 106
4.8.1 Syntax Trees and Left Folds 106
4.8.2 Left-Folding by Hand 109
4.8.3 Left-Folding in Autumn 112
4.9 Defining Expression Families 113
4.10 Discussion 118
4.11 Related Work 120

5 Principled Stateful Parsing **121**
5.1 Context-Sensitive Parsing 122
5.1.1 Recall and Context-Sensitive Features 122

- 5.1.2 Context-Sensitive Grammars . . . 123
- 5.1.3 Context-Sensitivity & Parser Combinators . . . 124
- 5.2 State of The Art . . . 125
  - 5.2.1 Backtracking Semantic Actions . . . 125
  - 5.2.2 Data-Dependent Grammars . . . 127
  - 5.2.3 Monadic Parsers . . . 128
  - 5.2.4 Attribute Grammars . . . 129
  - 5.2.5 Unprincipled Stateful Parsing . . . 129
  - 5.2.6 Rats! . . . 130
  - 5.2.7 Marpa and Ruby Slippers . . . 131
- 5.3 Context Transparency . . . 131
- 5.4 Intuition . . . 133
- 5.5 Formalization . . . 135
  - 5.5.1 Parse State . . . 136
  - 5.5.2 Parsers . . . 137
  - 5.5.3 Primitive Operations . . . 138
  - 5.5.4 Parser Invocation Semantics . . . 140
- 5.6 Implementation . . . 141
  - 5.6.1 From Theory to Practice . . . 141
  - 5.6.2 Primitive Operations and the Log . . . 145
  - 5.6.3 Operationalization and Usability Concerns . . . 148
  - 5.6.4 Examples . . . 152
  - 5.6.5 Alternatives . . . 157
- 5.7 Conclusion . . . 159

6 Engineering Aspects **161**

- 6.1 Performance Considerations . . . 163
  - 6.1.1 The Conspicuous Absence of Exponentiality . . . 164
  - 6.1.2 Inefficient Idioms in Simple PEG Parsing . . . 164
  - 6.1.3 Packrat Parsing . . . 165
  - 6.1.4 Memoization in Autumn . . . 168
  - 6.1.5 Megamorphic Call Sites . . . 169
- 6.2 Performance Comparison . . . 172
  - 6.2.1 Run Times and Hot Spots . . . 173
  - 6.2.2 Virtual Machine Effects . . . 177
  - 6.2.3 Memory Footprint . . . 178
  - 6.2.4 Discussion . . . 178
- 6.3 Lexical Analysis . . . 179
  - 6.3.1 Motivation . . . 180
  - 6.3.2 Autumn's Lexical Analysis Emulation . . . 181
- 6.4 Error Reporting & Recovery . . . 184
  - 6.4.1 Performance & Error Reporting . . . 185
  - 6.4.2 Autumn's Error-Reporting Capabilities . . . 185

6.4.3 Custom Error States & Introspection 187
6.4.4 Longest-Match Analysis 187
6.4.5 Error-Recoverable Parsers 189
6.4.6 Lexical-Level Error Reporting 190
6.5 Grammar Traversal & Parser Visitors 192
6.5.1 Grammar Traversal 192
6.5.2 The Expression Problem 193
6.5.3 A Clean But Verbose Partial Solution 195
6.5.4 A User-Friendly Partial Solution 198
6.5.5 Well-Formedness Checking Using Built-In Visitors 201
6.5.6 Grammar Transformation Using `CopyVisitor` . . . 202
6.5.7 Abstract Parsers 203
6.5.8 Potential Further Applications 204
6.6 Debugging & Tracing 206
6.6.1 Parser Invocation Stack Traces 206
6.6.2 Custom Parsers for State Inspection 208
6.6.3 Testing Support 208
6.6.4 Performance Tracing 210
6.7 Grammar Composition 212
6.8 Conclusion 214

7 Conclusions and Future Directions 217

Bibliography 223

Appendices 235

A Autumn Java 8 Grammar 237

Chapter 1

Introduction

This thesis is concerned with the limitations of currently available parsing systems, and how to overcome them. In this chapter, we introduce parsing, some of the challenges and trade-offs we perceive in the field, and our approaches to tackle these challenges.

1.1 Parsing

Parsing is the process of analysing an input string in order to extract a structured representation of its content (a *syntax tree*) with respect to a specific language. In the thesis, we focus on parsing formal languages, such as programming or markup languages — as opposed to natural spoken languages. Unlike natural languages, formal languages are never ambiguous: there is only a single correct interpretation of the input.

Parsing is a pervasive activity: every time a source file must be turned into executable code, a parser is required. Similarly, parsers are used to convert input files into relevant data structures. It is fair to say that most programs include a parser — sometimes many.

As such, making parsers easier to write, use, and modify is a broadly beneficial endeavour. In particular, parsers should be written using a simple yet expressive notation. It should be easy to modify existing parsers, as the language definition may evolve. Finally, the parsers should be able to support diverse practical non-linguistic requirements, such as the generation of intelligible error messages when the input does not conform to expectations, permissive parsing in the presence of such errors,

and the ability to derive various artifacts from the parser specification.

Broadly, the practice of parsing can be divided between *ad-hoc parsing*, where a programmer writes code to implement a parser for a single language; and the use of *grammarware* [49] — parsing tools that derive from or use a grammar. Grammarware also includes other grammar-based tools, such as syntax highlighters and pretty-printers.

Parsing is one of the oldest disciplines in computer science, yet even today, practical challenges and inadequacies remain — whether one uses ad-hoc parsing or grammarware.

1.2 Inadequacy: Flexibility Versus Simplicity

Ad-hoc parsers have many advantages: they can be fast and can be customized to do exactly what the programmer requires. On the other hand, they are verbose and do not constitute a readable language description. Nor is it easy to verify that the parser conforms to an existing description of the language. Additionally, there is very little possibility of reusing the parsing code to perform other tasks that depend on the structure of the language, such as pretty-printing; or to parse other languages.[1] Composing two ad-hoc parsers might also prove difficult.

Parsing tools, on the other hand, use grammars which help alleviate many of the issues with ad-hoc parsers. However, they suffer from other issues. These tools are usually based on Context Free Grammars (CFGs) [14] or more recently Parsing Expression Grammars (PEGs) [27]. The syntax of many existing programming and input languages cannot be expressed in these formalisms because they exhibit context-sensitive features. (Such as the need to recall some earlier part of the input in order to make a parsing decision further down the line. Examples will be given in the next section.) Moreover, parsing tools are fairly rigid and typically cannot be easily customized.[2] Desirable customizations include custom error-reporting mechanisms and full control over the generated Abstract Syntax Trees (ASTs). We note that ad-hoc parsing also suffers from all

[1] This is by definition. As soon as you can reuse parts of your parsing code to fit different use-cases or languages, you are de facto dealing with a parsing framework, however incomplete.

[2] This is broad claim but — in our opinion — not a misrepresentation. We will go into considerably more details about this when discussing the related work to our various contributions.

these issues, in the sense that while it is possible to solve them, it is relatively arduous to do so.

This suggests a fundamental trade-off in parsing, between the simplicity and declarativeness of grammarware on the one hand, and the flexibility of ad-hoc parsing on the other.

The main objective of this thesis is to find the sweet spot to occupy on this trade-off, and then to develop solutions to fill that space.

In particular, we would like to reconcile programmers with parsing tools, which they often eschew in favor of ad-hoc parsing for a wide range of programming languages and input formats. The most frequently invoked reasons are lack of flexibility (often the inability to handle context-sensitivity), expressivity (often borne out of ignorance about more advanced parsing tools), poor error reporting, and performance.[3]

Three notes on terminology.

In what follows, and in the rest of the thesis, we will often use the term *"grammar"* to refer to the specification of a parser. A *"grammar"* can mean multiple things, but in the context of parsing it usually refers to the formal specification of a language. Every parser specifies a language: the set of input sentences accepted by the parser. As we shall see, our parsers can include non-declarative elements — arbitrary code. This does not make a parser any less formal: it still constitutes a description with precise semantics.

We will also use the term *"parser"* in two different but related ways. First, as we already have, to refer to the code that accepts or rejects an input sentence as part of a language and produces a syntax tree in the former case. Second, as a sub-unit of this global parser, which can be composed to create higher-level parsers. This is a staple of the *parser combinator* approach, on which we build. Under this paradigm, both

[3]It is pretty hard to find a reference to support these claims. No one actually polled programmers about whether they preferred parser generators or ad-hoc parsers. There is however some evidence in the fact that, among the top 20 open source programming languages in the TIOBE popularity index, only Ruby makes use of a parsing tool. Hobbyist languages do not fare much better.

meanings of *"parser"* are equivalent: the global parser being simply the parser at the top of the composition hierarchy.

A parser must generate some kind of tree to capture the structure of the input. We distinguish between *parse trees* or *concrete syntax trees* — which scrupulously follows the structure of the grammar rules (or parsers) used during the match, and *abstract syntax trees* (ASTs) in which tree nodes need not correspond to any specific grammar rule or parser at all. We will use *"syntax tree"* to refer to ASTs or to both kind of syntax trees in general. We sometimes use *"parse tree"*, even when no such tree is actually generated, to denote the way an input is matched by a grammar. Finally, we shall also use the abbreviation *"AST"* without expanding it each time.

1.3 The Best of Both Worlds

We want a flexible yet simple parsing system. A system that, as much as feasible, exhibits the desirable properties of both ad-hoc parsing and grammarware, while avoiding their pitfalls.

At this point, it is necessary to enunciate what we think these desirable properties are, and to justify them briefly. We do not make a list of pitfalls, as these can largely be seen as dual to the desirable properties: the unability to achieve a desirable property is a pitfall.

The flexibility of parsing systems should be manifested via the following capabilities:

- The ability to extend the parsing system with new *combinators*, *i.e.*, new ways of combining existing parsers. For instance, we could add a combinator that matches the same thing as its longest-matching sub-parser. Or a combinator that matches a list of items specified by a sub-parser, and whose separator is specified by another sub-parser.

- The ability to create new primitive parsers that consume part of the input stream without calling out to other parsers. With this, we can match objects in an input stream (*e.g.*, tokens) using custom logic, and not limit ourselves to a single property (*e.g.*, the token type: identifier, integer literal, ...), as is usually the case. It also enables reusing parsers defined outside the framework as building

blocks.

- Full control over the generation of the syntax trees. If you do not directly get the trees you want, you have to transform those you get, which is wasteful and cumbersome.

- Allowance for a wide variety of context-sensitive features in the language's syntax and generated syntax trees. Many mainstream programming and markup languages possess context-sensitive features that cannot be handled by traditional parsing tools. For instance, C has namespace-dependent ASTs (some statements have different meanings depending on whether an identifier designates a type or a variable); Python and Haskell have significant whitespace; XML must match the identifier in paired opening and closing tags (*e.g.*, `<foo>` and `</foo>`).

- The ability to customize the error-reporting strategy and the reported error messages.

- The ability to compose independently-developed grammars — at the very least to embed one language within another. It is common to embed a domain specific language — such as that of regular expressions, or SQL — into a general programming language.

- The reification (availability for programmatic inspection) of the grammar: It should be possible to traverse a description of the grammar — usually made out of rule definition and sometimes other combinators. This enables other language-based tools — such as syntax highlighters and pretty-printers — to be automatically generated from the grammar.

- Sufficient performance for parsers to be practically usable. While ad-hoc parsers are more amenable to high performance, parsing tools are often fast enough in practice. It is still necessary to pay attention to this aspect while devising new approaches, especially regarding what could be fundamental limitations on the achievable performance. We should also embrace the possibility of parser customization being used to tackle performance bottlenecks.

Conversely, the simplicity of the system should be manifested as follows:

- The grammar (the parser's specification) should be readable, in

order to avoid maintaining a separate language description. This entails the need for parsing system customizations to be expressed tersely — or for them to be cleanly encapsulated and accessed through a simple API.

- The system should, as much as possible, be devoid of counter-intuitive pitfalls. In general, the *simple* way should be the *right* way.

- The system should cover, using a limited number of built-in primitives, most common parsing use-cases. In particular, it should certainly be as expressive as PEGs and/or CFGs. It is important for the base framework to be expressive in order to avoid over-customization and wheel reinvention. At the same time, if the base system is too big, its subtlest components risk being ignored.

- The system's principles should be simple and easy to understand so that it is simple to extend and customize.

- The reification of the grammar (see above) should be easy to manipulate, so as to ease the development of other language-based tools.

These lists are by no means exhaustive — there are other desirable parser properties and even other desirable simplicity or flexibility properties.

Our selection emphasizes properties that are problematic for ad-hoc parsers or typical grammarware. As such, these are also the properties that are being traded off when selecting one or the other. While such a list is, naturally, somewhat subjective, we do believe it to be fairly non-controversial given the stated objectives.

It may seem that our criteria — which mention composing and creating custom parsers — are skewed towards parser combinators. While we do indeed believe that this way lies one of the possible answers, we do note that even recent state-of-the-art CFG parsers [72, 98] do make the grammar rule the unit of reuse and extension, and in a sense this was always the case, even if extension only took the forms of semantic actions.

As we tackle these different capabilities in this thesis, we will be careful to highlight and discuss the trade-offs introduced by our proposed solutions.

1.4 The Approach: Principled Procedural Parsing

In order to accomplish our goal of building a bridge between ad-hoc parsing and grammarware, we propose a somewhat unorthodox approach of the notion of *parsing*.

Instead of considering parsing as merely the derivation of a structured representation from an input stream based on some description of the language, **we approach parsing as a computation over the input stream, enriched with free-floating context**.

By *free-floating context*, we mean contextual information that is not explicitly passed between parser components, but is available to all. This context should be scoped to the parse, so that multiple inputs may be parsed in parallel without interfering with each other. As we shall see, we implement this context as parse-global state made available to parsing components during the parse. This context is further subdivided into specific sub-contexts associated to a specific key. We encourage the use of unique keys to avoid components accidentally interfering with each other.

Ad-hoc parsers could be seen as an example of the latter approach, where the free-floating context is simply program state being shared between components of a parser. However, it is difficult to deal with this state correctly in the presence of backtracking. Additionally, ad-hoc parsers suffer from all the pitfalls we outlined before.

Overwhelmingly, grammarware does not allow the use of free-floating context. When matching grammar rules, parsing tools only consider the (start of) the remainder of the input. In particular, the history of the rules that were previously matched — leading to the use of the current rule — is unavailable. Similarly, one does not usually have access to the start of the input or any data derived thereof.

If you cannot use contextual information to make parsing decisions, then this precludes context-sensitivity in your language a priori. That is not the only use case for contextual data. For instance, an understanding of how the current rule was reached can be helpful in generating better parsing error messages. It can also help extend the parsing framework with new useful parsers.

Immediately, it becomes apparent that we will need some kind of language to manipulate this state. We simply chose to embed parser specifications within an existing general-purpose host programming language, and allow the use of arbitrary host language code.[4] There are few good reasons not to make that choice. A smaller (external) Domain Specific Language (DSL) could make it easier to build proofs about the grammar, but there seems to be little interest in that, both in industry and in academia, and it remains to be seen what is in fact possible to prove with an extensible parser description — as opposed to a traditional closed-form formal grammar. On the other hand, using the host language immediately opens the possibility of implementing our approaches as a library within an existing general-purpose language — an attractive practical proposition. Parser implementations and framework extensions such as custom parsers and new grammar-derived tools can also be distributed as language components.

We name our approach — predicated on the manipulation of contextual information and the ability to inject arbitrary code into the parsing system — *Principled Procedural Parsing.*

We call the approach *procedural* because of the idea that we are going to issue instructions that manipulate the parsing context. The specification of a parser is not just a language description: it is the description of a computation — it guides the execution flow. The term *procedural* is used as a tongue-in-cheek counterpoint to *declarative* — even though we would still like to keep things as declarative as possible, whenever possible.[5] It is the recognition that to accomplish our stated objective, we deem it necessary — in fact, desirable — to expose this lower-level computation layer, instead of abstracting it away.

Furthermore, we say the approach is *principled*, for two reasons. First, the goal is still to parse some input stream, and as such our systems will be structured in a way that is input-centric. The goal of most parser components is still to match some input. Therefore, we will lay out some ground rules — principles — that govern how computations should be

[4] The use of custom code in parsers is sometimes called *semantic actions* in the literature. However, we go beyond the traditional purview of semantic actions by allowing the use of custom code in a way that is less restricted — yet safer — than usual.

[5] And in fact, an important part of the work we did, especially with regard to grammar reification, aims to recapture some of the declarativeness lost by allowing arbitrary extensions.

structured. Second, we need to ensure that context manipulations are safe. As we will see, this is not a given in the presence of backtracking, which may lead to making some context changes obsolete.

1.5 Autumn: Architecture of a Solution

Our quest for a flexible yet simple parsing lead to the development of a tool — *Autumn*. In this section, we briefly outline some fundamental features of Autumn and how they help implementing the principled procedural parsing approach.

> Autumn implements the *principled procedural parsing* approach as a Java 8 parsing library, which is available online at `https://github.com/norswap/autumn`. The repository contains links to the user manual (which include worked out examples) and the full Javadoc documentation for the Autumn API.

Our solution builds upon Parsing Expression Grammars (PEGs). PEGs (which are presented in more details in Section 2.4) can be seen as a formalization of top-down recursive-descent parsers — the usual approach used in writing an ad-hoc parser — but without the flexibility afforded by custom code.

Because of this, PEGs can be seen as the specification of a parsing control flow: the one realized by the corresponding ad-hoc parser. In fact, this interpretation turns out to be a viable implementation. The only widespread implementation variant is to memoize the result of matching a grammar rule at a given input position. This is a trivial transposition of the idea of memoizing the result of a function given a set of inputs, and so does not denature the "computational" nature of PEGs.

The missing puzzle piece is a way to reintroduce flexibility into the PEG formalism. For this, we need two things: first, the ability to run our own code within the framework of PEG, and second, a way to manipulate context safely.

The first thing is relatively straightforward. Parsing expression grammars are made up of small *primitive* parsing expressions (*e.g.*, character classes), which may be combined into larger parsing expressions using *combinators* (*e.g.*, sequences, choices and repetitions) — which may themselves be

further combined.[6] What we need to run our own code is simply the ability to define new primitive parsing expressions and new combinators — new parsers.

Within these custom parsers, we need to be able to manipulate the context. The problem here is that when faced with a choice, a PEG parser may speculatively try an alternative, and needs to backtrack if this alternative does not succeed. When backtracking happens, all changes made to the context during the speculative execution need to be reversed. As it turns out, we are able to do just that. We experimented with different approaches (more details later), but in the end, associating every context change with an *undo action* (a corresponding reverse change) turns out to be the simplest solution.

We note that the parsing system is still in charge of driving the flow of the parsing code,[7] based on the input and the behaviour defined by a grammar's parsers. Granted that the parsers satisfy some simple contracts, this enables the framework to ensure the required context safety guarantees.

We dubbed this approach to context-sensitive parsing *"Principled Stateful Parsing"* because in this case, the context is held within mutable state, whose integrity must be preserved in the presence of backtracking.

We improved another aspect of PEG parsing which we thought was sorely lacking — its ability to parse the syntax of infix (binary) expressions both efficiently and in a way that produces syntax trees with the correct associativity. We discuss multiple ways to solve these unexpectedly thorny issues.

We also dedicated a lot of thought and energy to *engineering aspects*: these aspects that are not directly related to parsing algorithms and their expressiveness, but are nevertheless crucial in building a practically useful parsing tool. These aspects include performance, error handling, grammar reification and traversal, debugging and grammar composition.

[6] Rules are merely named "pointers" to parsing expressions that may be referred to from within expressions.

[7] In the sense that it performs *inversion of control*: during parsing, Autumn is in charge and occasionally calls user-supplied code, not the other way around. Autumn is a *"library"* in the sense that it is a simple dependency that can be added to a Java program and called by it without needing to install supporting utilities; but it is also a *"framework"* because of inversion of control.

The engineering side of these aspects is seldom discussed, and we consider our treatment of them to be an important contribution of this work. We pay particular attention to the usability of the features we introduce, by trying to make the easy thing right, and the right thing easy.

Our hope is that the approach will first make it easier to perform some parsing task — especially those with specific or unusual requirements; and second, enable and inspire the creation of new — hopefully more palatable — parsing tools. The sort of parsing tool we advocate (and have built) give back to the user a measure of control over the parsing process, without incurring the tedium and complexity of an ad hoc parser.

All along this thesis, we will demonstrate how Autumn helps solve practical parsing problems. We invite readers to download the tool[8] and experiment with it.

1.6 Overview & Contributions

We now give an overview of what to expect from the rest of this thesis, outlining our various contributions and how they relate to each other.

Figure 1.1 contains a (partial) map of concepts and features that will be presented in the thesis. An arrow between a node A and B means that B *uses* A in some capacity — either B is built upon A, or A can be used in the realization of B. In the explanation that follows, we will refer to the nodes in this map by marking them in **bold**.

At the top of our hierarchy, we have an **extensible parser combinator system** inspired by the PEG formalism. Parser combinators and PEG are introduced in Section 2.3 and Section 2.4, respectively. Chapter 2 also presents additional background on parsing in general.

While parser combinator frameworks are relatively widespread, we emphasize the extensibility of our approach and its simple underlying model. The basic principles of Autumn are presented in Chapter 3, and in particular, we present the underlying model Section 3.2 and its embodiment in the base `Parser` class (which is inherited by parser[9] implementations)

[8] `https://github.com/norswap/autumn`

[9] Recall that we use the term *parser* to designate what would be called *parsing expression* or *parser combinator* in other frameworks. See the shaded box on page 9 for more details.

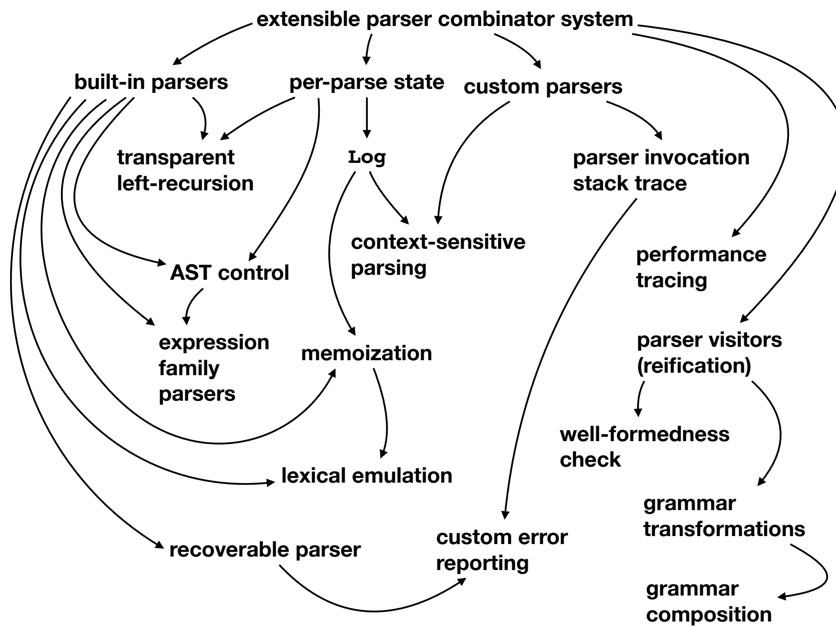


Figure 1.1: A (partial) map of concepts and features that will be presented in the thesis.

and the `Parse` class which encapsulates the state of a single parse.

Most features of Autumn are built upon — or extend — this base parser combinator system. Most obviously, we have the **built-in parsers** implementations, and the ability for users to define their own parsers (**custom parsers**). We note that all built-in parser implementations only use features available to the users, but we distinguish built-in and custom parsers to indicate when users are required to implement such a parser themselves.

As we made clear in this introductory chapter, the use of free-floating context, is a key feature of Autumn. The context is held within parse-specific state objects (**per-parse state**).

To achieve **context-sensitive parsing**, the other necessary piece is the **`Log`** data structure, which must be used to apply changes to these per-parse states. This data structure represents the source of truth for the parse state, while the actual content of the per-parse state objects is akin to a *materialized view* of the log. In that respect, the parse state is similar to a transactional database: changes are induced by parsers. If a parser fail, all the change induced by itself and all of its successful sub-parsers must be undone. The metaphor is imperfect however: transactional databases typically deal with multiple transactions, which may be concurrent. A parse is more similar to a hierarchy of nested transactions (one per parser), with the grammar's root corresponding to the outermost transaction.

This leads to our *principled stateful parsing* approach to **context-sensitive parsing**. Context sensitivity arises from making parsing decisions from the content of the log (*i.e.*, *the context* or *the state*). We formalize a series of primitive state manipulation operations that can be used to ensure that state manipulations are safe in the presence of backtracking — in other terms, to enforce transactional discipline. The approached is presented in Chapter 5. In particular, Section 5.6 explains how the formalization relates to the its implementation as the `Log` data structure.

A particular example of parse state that is emminently useful in practice is *the value stack*. This stack is used to build an Abstract Syntax Tree (AST) corresponding to a parse. Autumn offers built-in parsers that help construct an AST with ease (**AST control**). Since constructing an AST is a fundamental part of a parser's job, and doesn't necessarily require understanding the `Log` data structure, we present these capabilities in Section 3.3.

Beyond the traditional syntax-definition parsers (such as those corresponding to the PEG operators [27]) and the AST parsers, Autumn bundles other parsers to achieve various other goals. We present some of these parsers in the following paragraphs.

Parsing infix expression — in particular when mixing left and right associativity — is a perennial source of difficulties in parsing systems. Some of these difficulties are presented in Section 2.5. Specifically, with respect to our chosen approach, the PEG formalism does not support left-recursive grammar rules, and its semantics make an intuitive definition of associativity arduous. To alleviate these issues, we provide multiple means to define the syntax of infix expressions, implemented as built-in parsers. These solutions are presented in Chapter 4. In particular Section 4.5 introduces **transparent left-recursion** handling for Autumn, and Section 4.9 introduces the **expression family** combinators which help define mixed-associativity infix expression without explicit left-recursion while ensuring good parsing performance.

Autumn also enables customizing and improving error reporting (**custom error handling**). Its facilities for this purpose are described in Section 2.6. Beyond the facilities provided by the base framework, users can use custom parsers and custom parse states to store additional information. Additionally, Autumn includes built-in **recoverable parsers** which offers a fallback mechanism in case some part of the input can't be parsed in the proper way. All of this is presented in Section 6.4.

Autumn is also able to record **parser invocation stack traces**, indicating the chain of parser invocations leading to a particular event. By default, Autumn will associate such stack traces with parse errors, but users can also make use of them within their custom parsers.

Regarding performance concerns, the **memoization** and **lexical emulation** parsers (presented in Section 6.1.4 and Section 6.3, respectively) help tackle performance issues. Autumn is also able to trace the parse in order to generate detailed performance metrics, including the number of invocations of each parser, as well as the total time spent on these invocations (**performance tracing**).

As for reification — the requirement that the grammar be programmatically inspectable — we realize it by means of **parser visitors**, which realize a somewhat modified version of the visitor pattern [29]. This is presented in Section 6.5. Most notably, we use parser visitors to

generalize the PEG **well-formedness check** [27] to custom parsers, as well as enable **grammar transformations**, which in turn can be used to perform **grammar composition**. We explore that last use case in Section 6.7.

This thesis is supported by some of our prior publications. Chapter 4 greatly expands on our paper *"Parsing Expression Grammars Made Practical"* [57]. Chapter 5 covers our paper *"Taming Context-sensitive Languages with Principled Stateful Parsing"* [58], but provides a different realization of the same principles. Finally, in Section 6.4.4 we briefly explain how the ideas in our vision paper *"Red Shift: Procedural Shift-Reduce Parsing"* [55] can be applied to an extensible parser combinator framework like Autumn.

Chapter 2

Background

In this section, we review the background material to which we will refer in the rest of the thesis. In particular, we will dwell on the most popular parsing algorithms, including algorithms for Context Free Grammars (CFGs) and Parsing Expression Grammars (PEGs). We'll also discuss ad-hoc parsers, parser combinator frameworks, as well as the way parsers usually deal with infix expressions and errors.

2.1 Context Free Grammars (CFGs)

While our approach does not build on CFGs directly, they remain the most widespread vehicle of syntax specification. As such, it is important to understand them in order to contrast them with our own approaches. Similarly, previous work on CFG parser implementations is highly relevant to contextualize our contributions. We will first speak of the formalism itself, then briefly touch on its parsing algorithms.

2.1.1 The CFG Formalism

CFGs are one of the seminal forms of formal grammars — a way to define formal languages. They were introduced by Chomsky as part of his hierarchy of formal grammars. CFGs (Type-2 grammars) are strictly more expressive than regular expressions (Type-3 grammars) and strictly less expressive than context-sensitive grammars (Type-1 grammars). [14]

In formal language theory, a language is defined as a (possibly infinite) set of sentences. A sentence is a string of terminal symbols (terminals).

In the context of parsing, the set of terminals will usually be the set of admissible input characters or a predefined set of token objects. There is also a set of nonterminal symbols, which are used in the grammar's definition.

Formal grammars describe a set of production rules of the form `left` $\rightarrow$ `right`. For CFGs, the left-hand side is always a single nonterminal symbol (usually denoted by an uppercase word), while the right-hand side is a sequence of zero or more symbols (terminals or nonterminals both).

CFGs are sometimes said to be *generative*, because any sentence in the described language can be obtained by a sequence of rewrites based on the production rules.

Such a rewrite always starts with a distinguished nonterminal start symbol (usually noted S). Our initial candidate sentence is a string containing only S. At each step in the process, we can take any nonterminal occurring within our candidate sentence, and replace it with a matching right-hand side from one of the production rules. If at some point in this process no nonterminals are left in the candidate sentence, it is guaranteed to be a sentence of the language.

$$
\begin{aligned}
&S \rightarrow x \\
&S \rightarrow y \\
&S \rightarrow z \\
&S \rightarrow S + S \\
&S \rightarrow S - S \\
&S \rightarrow S\ * S \\
&S \rightarrow S\ /\ S \\
&S \rightarrow (S)
\end{aligned}
$$

Figure 2.1

The set of nonterminals is — by definition — the set of admissible rule left-hand sides. Formally, their role is purely mechanical in the process of generating language sentences, or — conversely — in establishing whether a sentence belongs to the language.

However, since grammars are usually written by humans, nonterminals will *tend* to correspond to well-known language constructs. As we will see, many tools exploit this approximation, notably to generate syntax trees.

Figure 2.1 shows an example of simple CFG for the language of arithmetic expressions over variables x, y and z.

2.1.2 CFG Parsing Algorithms

While the generative property of CFGs is elegant, it is not immediately helpful in the context of parsing, where we first have to recognize whether a given sentence belongs to a language. Hence, the need for dedicated parsing algorithms.

Historically, there have been three main types of CFG-based parsing algorithms: top-down parsers, bottom-up parsers and chart parsers.

In the current context, a *parser* is the code that performs the task of parsing, based on a grammar, and running a particular parsing algorithm. In the discussion that follows, we will often refer to the properties of *the X parsing algorithm* as the properties of *X parsers*.

As stated at the very start of this thesis: "Parsing is the process of analysing an input string in order to extract a structured representation of its content with respect to a specific language." Yet, algorithmically, the main challenge lies in validating whether the input sentence belongs to the language. Extracting the structure is more of an implementation concern, although it does sometimes have algorithm implications — especially when *ambiguity* is involved (see below).

We call *recognizer* an algorithm that only validates whether an input string conforms to a language defined by a grammar, without extracting its structure in the form of a parse tree.

Ambiguity occurs when a parser is able to generate multiple different parse trees for the same input. Said otherwise, when the grammar allows the same input to be matched in different ways; or formally, when the sentence can be generated using two sequences of rewrites that are distinct up to reordering.

Ambiguity is common in natural languages, but is undesirable in formal languages. As such, grammatical ambiguities have to be eliminated. It is possible to write a grammar such that it is not ambiguous in the first place. However, some grammatical idioms are so useful that some parsers allow the user to write ambiguous grammars, which are then explicitly disambiguated through *disambiguation rules* [1]. We note that the determining whether a CFG is ambiguous is undecidable in

general [81].[1]

2.1.3 Top-Down Parsers

Top-down CFG parsers start with a sequence containing only the starting nonterminal S and recursively replace the first nonterminal in the sequence by one of its associated right-hand sides, until the sequence becomes identical to the input.

These parsers are called *top-down* because they start from the "top" of the grammar (S) and progressively replace high-level nonterminals by their constituent parts.

The issue with our simplistic description is *choice*: a nonterminal may have multiple associated right-hand sides. Different top-down parsers will be mostly differentiated by how they deal with choice.

Let's first consider a naive top-down parser which remembers all the choices it makes, and stores these choices in a stack data-structure. During the execution of the parser, the sequence always has a prefix of zero or more terminals symbols. Replacing a nonterminal by one of its right-hand sides is guaranteed never to shrink this prefix, since only a nonterminal is removed. If, at some point, we notice that the terminal prefix of the sequence diverges from the same-length prefix of the input text, we can conclude that the parse cannot succeed with the choices we made. If this occurs, we need to *backtrack*: restore the state at the time of the last choice, and try another right-hand side. If all right-hand sides have been tried and are unsuccessful, we backtrack further to the previous choice.

Figure 2.2 shows the pseudo-code for a simple recognizer (not a parser, since a parse tree is not built) using a naive top-down algorithm. It works as explained previously, but we systematically shed the common (terminal) prefix of our sequence (`symbols`) and our input string (`input`).

The naive algorithm suffers from two big issues. First, it cannot handle the whole class of CFGs because it fails on left-recursive rules. A left-recursive rule is a rule whose right-hand side can be expanded to a sequence that will eventually start with the same nonterminal as the rule's left-hand side. The most obvious examples are *direct* left-recursive

[1] We will say more on ambiguity — and their relationship with the related problem of *prefix capture* — in Section 2.4.3.

```
1  function parse(symbols, input)
2
3      if (symbols is empty)
4          if (input is empty)
5              return accept
6          else
7              return reject
8
9      else if (symbols.head is a terminal)
10         if (input is empty || input.head != symbols.head)
11             return reject
12         else
13             return parse(symbols.tail, input.tail);
14
15     else if (symbols.head is a nonterminal)
16         for each alternative (symbols.head -> rhs)
17             if (parse(rhs + symbols.tail, input) == accept)
18                 return accept
19         return reject
20
21 parse(list(starting_symbol), input)
```

Figure 2.2: Pseudo-code for a naive top-down recursive CFG recognizer. The algorithm parses the input (a list of terminals) against a list of symbols (initially a list containing only the starting symbol of the CFG). This algorithm was adapted from a textbook. [60]

$$
\begin{aligned}
S &\rightarrow Xb \\
S &\rightarrow Xc \\
X &\rightarrow a \\
X &\rightarrow aX \\
X &\rightarrow aXX
\end{aligned}
$$

Figure 2.3: Simple non-left-recursive CFG that leads to exponential parsing times when using the naive top-down CFG recognizer algorithm. [87]

rules such as $As \to As\ a$, where the right-hand side starts with the left-hand side's nonterminal. When it encounters a left-recursive rule, the naive algorithm will loop forever, as it keeps replacing the same nonterminal again and again.

Second, the naive algorithm has worst-case exponential complexity. It is rather difficult to characterize the class of grammars that cause exponential run-times in practice, and the subject is not well-studied.[2] Figure 2.3 shows a simple non-left-recursive CFG that leads to exponential parsing times with the naive algorithm. This grammar matches a sequence of as followed by a single b or c. The problem in this case is that X is massively ambiguous: it's never clear whether the as should all be matched by a single match for X (second X rule) or split amongst two matches for X (third X rule). This dilemma occurs at every input position. In the unfortunate event that the input string ends with c, every combination of the second and third X rule for every input position has to be considered before the second S rule is even taken into consideration.

However, the naive algorithm is not as far-fetched as one may think, as it is almost exactly the one employed by Definite Clause Grammars (DCGs). DCGs are a mainstay feature of logic programming languages, most notably Prolog, for which it was developed [17]. DCG rules are translated into logical clauses which are executed by the logic engine.[3] It is also possible to include regular logic predicates into the rules, making the algorithm more extensible than any other presented in this section. Finally, it is possible to thread state through the grammar, a mechanism inspired from attribute grammars [52]. It can be used to generate ASTs, but can also be exploited to introduce context-sensitivity into the grammar. We will return to this notion in Section 5.2.2.

While the naive algorithm has worst-case exponential complexity, we do know classes of grammars which can be parsed with much better complexity bounds. For top-down parsers, the most famous class is that

[2] Such exponential grammars do seem to be very rare in practice, although other inefficiencies may exist. We explore this topic further — but for PEG grammars — in Section 6.1, the discussion of which may carry over to CFGs to some extent.

[3] Explaining the intricacies of logic programming and their relationship to parsing is out of the scope of this thesis; but if you would like to learn more about Prolog and DCGs, we recommend a short tutorial of our own [54] or the very complete *The Art of Prolog* [89].

of $LL(k)$[4] [80] grammars. These are grammars for which the parser can unambiguously select the correct right-hand side to use when expanding a nonterminal, if given k tokens of lookahead — *i.e.*, looking at k input tokens after the end of the common prefix between the input and our current sequence. k is always fixed in advance.

LL parsers have linear time complexity. However, the lookahead restriction is bothersome, and manifests itself as *first/first* or *first/follow* conflicts[5] within grammars that do not belong to the LL(k) class. Left-recursive grammars are not in LL(k) [80].

In order to bypass this limitation, alternatives building upon the LL parser machinery have been proposed.

$LL(*)$ [71] expands over LL(k) by falling back to backtracking only if a k-lookahead cannot unambiguously select a right-hand side. LL(*) has restricted backtracking compared to general CFG parsers, and so can only handle a subset of all non-left-recursive CFG grammars — but strictly more than LL(k). LL(*) backtracking is actually much closer to PEG parsers. The difference in backtracking between CFGs and PEGs will be explained in Section 2.4.

$ALL(*)$ (for Adaptative LL(*)) [72] is a further elaboration over LL(*). Whereas LL(*) performs k-lookahead through a *Deterministic Finite Automaton* (DFA) built ahead of time, ALL(*) constructs a DFA dynamically, based on the actual input. Whenever a portion of the input does not match the DFA, the DFA must be expanded. To find the correct right-hand side in that case, ALL(*) uses an *Augmented recursive Transition Network* (ATN), essentially a *Non-deterministic Finite Automaton* (NFA) augmented with a stack that can use the result of other DFAs or NFAs to make transitions. Parsing using a NFA is normally exponential, but because every NFA lookup at a given input position ends up cached in a DFA, the complexity stays polynomial. As a whole, ALL(*) has $O(n^4)$ worst-case complexity, albeit the run-time is almost always linear in practice.

---

[4] *LL* stands for *Left-to-right Leftmost derivation*, although this name fails to capture the essence of what LL grammars are about — it just means that nonterminals are expanded left-to-right in LL parsers.

[5] These conflicts occur when two right-hand sides can be reached with the same bounded lookahead sequence. *first/first* conflicts occur when those right-hand sides are for the same nonterminal. *first/follow* conflicts occur when they are for different nonterminals that can occur at the same input position.

Benchmarks show that ANTLR 4, the reference implementation of ALL(*), is faster than other general CFG parsers. However, it cannot handle non-direct left-recursive grammars. [72] This shows that the worst-case complexity is not always the most important factor when determining the performance of parsing algorithms.

Finally, *GLL* (for Generalized LL) [82] is a CFG parser that can handle any CFG in worst case $O(n^3)$ complexity. Like LL(*) and ALL(*), it also builds upon ideas from LL(k), but has a different strategy to deal with choice and avoid exponential run times. The GLL implementation is rather complex, but relies on a data structure called *Graph-Structured Stacks* (GSS), which is also used to implement bottom-up parsers.

In the naive algorithm, exponential run times occur when the same nonterminal is expanded into the same right-hand side over and over (at least in proportion to the input size) at the same position. The way to avoid this is to memoize the end results of these expansions. By *end result*, we mean the expansion into the right-hand side, and then the subsequent recursive expansions of the nonterminals therein, until only terminals remain. We only consider expansions that are compatible with the input string. There might be many such end results because of grammatical ambiguity.[6] By treating the set of these results as a single entity, one can first avoid recomputing them every time they are needed, but also construct bigger entities that are made up of smaller ones. The important property of these entities is how much input they can span, *i.e.*, the set comprising the size of the input matched by each result. In ALL(*), the memoization is fulfilled by the DFAs, while in GLL it is the role of the GSSs.

2.1.4 Bottom-Up Parsers

While top-down parsers work by recursively expanding nonterminals in the candidate sequence until it matches the input string, bottom-up parsers work the other way around. They start with the input string — the targeted sequence expansion — and recursively *reduce* sequences of symbols into nonterminals, until only the starting nonterminal S remains.

A naive implementation of the idea is even less efficient than a naive top-down parser, because at least the naive top-down parser is able to use the terminal prefix of the sequence to guide the search and back out

[6] In top-down algorithms, ambiguity occurs whenever multiple different choices in right-hand side selection lead to recognize the string as part of the language.

of dead-ends early.

Just like for top-down parsing, this efficiency issue gave rise to the discovery of sub-classes of CFGs that could be parsed efficiently. Chief amongst them is $LR(k)$[7] [50]. LR(k) actually defines two classes of languages, for $k = 0$ and $k \geq 1$. However, both of these classes include all deterministic context-free languages — which are languages that can be accepted by a *Deterministic Pushdown Automaton* (DPDA).[8] These languages are also unambiguous. LR parsers have linear time complexity. There are some variations (*SLR*, *LALR*, ...) with slightly different expressivity and memory usage characteristics, but for this overview we will stick to the generalities.

LR parsers are implemented as shift-reduce parsers. A shift-reduce parser is driven by precomputed tables, and uses a stack as its main data structure. An LR parser scans the input linearly, processing one terminal at a time. At each step of the parse, the parser looks at the symbols at the top of the stack, the current terminal, and the k subsequent terminals (*the lookahead*). Based on this, the parser consults its tables and takes the decision to either shift the terminal onto the stack, or to reduce some items on the top of the stack into a nonterminal — in which case the symbols on the top of the stack naturally match a rule's right-hand side. The parse ends when no more steps can be taken: either no actions match the current terminal and stack, or the whole input string was processed and no more reductions can be performed. The parse succeeds if only the starting symbol S is left on the stack.

The LR language classes are not incredibly intuitive, relating to the notion of determinism. For instance, palindrome grammars such as $A \rightarrow a\ A\ a\ \mid\ \epsilon$ [9] are not deterministic. Indeed, the parser needs to decide where to stop expanding A into the first alternative and into the second one (the empty string), *i.e.*, it needs to find the middle of the input. However, a shift-reduce parser cannot possibly determine this

[7] *LR* means Left-to-right Rightmost derivation. This refers to an implementation detail of LR parsers. As an approximation, we could say that, when faced with ambiguity, LR parsers prefer a right-associative interpretation of the input.

[8] It's important to note the difference between a deterministic *language* and a deterministic *grammar*. A deterministic language can always be described by one or more deterministic grammar, but may also be described non-deterministic grammars, that cannot be automatically translated into a DPDA.

[9] The pipe refers to choice, so $A \rightarrow a\ A\ a\ \mid\ \epsilon$ is equivalent to the two rules: $A \rightarrow aAa$ and $A \rightarrow \epsilon$. ϵ refers to the empty string.

with a single scan over the input.

We also note that every $LL(k)$ language is also a $LR(k)$ language. In practice, most programming language grammars are deterministic or nearly deterministic (cf. shaded box below). For this reason, LR parsing is the technique of choice for parsing tools of historical significance such as Yacc and GNU Bison. In these parsers, failure to supply the grammar of an LR-class language manifests in *shift/reduce* and *reduce/reduce* conflicts, respectively a failure to choose between a shift and reduce action, or between different reduce actions.

Are programming languages mostly deterministic?

It may seem like a bold claim that the grammars of most programming languages are mostly deterministic, and it is rather hard to show conclusively. We do however think that the claim is broadly true.

We can make a historical argument, that most grammars where written for *LL* or *LR* parsers, and hence deterministic. However, the causality could easily be reversed: languages were made deterministic because that's all the tools could handle.

A better argument is that we do not know useful programming language constructs that require (or are improved by) syntactic non-determinism.[10] For this to be true, we would need multiple programming constructs that can match an unbounded amount of input, and whose prefix are ambiguous (but not identical) such that they can only be distinguished by a bounded suffix. And that's only the easiest case — things can be worse, like in our palindrome example.

To clarify, let us a take an example that does not quite satisfy this criterion: postfix expressions. Consider the following expression which is valid in Java, C, etc: `(a[0][0][0])++` and `(a[0][0][0])--`. The part between parens could grow arbitrarily (there is no limit on how deep arrays can nest). Yet not until the postfix operator (`++` and `--`) do we know the correct construct. Of course, this case does not quite cut it because the prefix is identical in both cases: those are expression of lower precedence than the postfix operators.

Palindrome-like examples do not apply to programming languages: most bracketed constructs come in pairs (*e.g.*, `{}` or `[]`). Cases where the left

and right brackets are the same (*e.g.*, `""`) typically do not nest, probably because they would be too confusing for human programmers.

Another pitfall: as we will see later, many languages cannot be expressed as CFG (or PEG) at all. However, this is usually a problem of context-sensitivity, and the simplified version of the language described by a grammar is typically deterministic. This is still a problem, but one that requires a different cure, which we introduce in Chapter 5.

Later, a generalized LR parser — able to handle all CFGs — was also devised. It was named *GLR* (for Generalized LR) [94] and just like GLL, uses *Graph Structured Stacks* (GSSs). GLR antecedes GLL and pioneered the use of GSSs. In brief, the algorithm proceeds like an LR parser, but whenever a simple LR parser would encounter a conflict, the GLR parser forks the execution into multiple stacks. Down the line, the GLR parser is able to merge multiple congruent executions, and so to share common stack prefix and suffixes amongst multiple executions. In theory, GLR parsing can have $O(n^3)$ [48] complexity, however this comes with a constant factor so high as to be unusable, so the complexity of practical implementations is $O(n^{p+1})$, where p is the length of the longest right-hand side in the grammar [72]. However, the practical run time is dependent on the degree to which the grammar is deterministic. Deterministic grammars run in $O(n)$ time.

2.1.5 Chart Parsers

Chart parsers are general parsers capable of handling all CFGs that rely on a global data structure — the chart — to perform the parse. The approach centers on the use of dynamic programming to solve sub-problems, whose answers are stored in the chart. While we treat them apart from top-down and bottom-up parsers, chart parsers do in fact strongly exhibit top-down and/or bottom-up properties. In this section, we will briefly review the Earley and CYK parsing algorithms.

[10] Professor Peter van Roy pointed out to us that syntactic non-determinism can be useful in order to handle syntax errors. The ambiguity here comes from the fact that there are typically multiple ways to interpret an erroneous input. Nevertheless, that is a parser feature rather than a language construct — so our point stands. Such "ambiguous" error handling can in fact be implemented with custom parsers in Autumn.

The *Earley* algorithm [24] works by simulating a *Non-Deterministic Finite Automaton* (NDFA) that accepts the grammar. It does so by using a set of linear automata corresponding to grammar rules. In these small automata, the possible states are before each symbol in the right hand side sequence, or after the sequence. This is typically represented with a dot, *e.g.*, $[a\ b\ \bullet\ c]$ is a state for the rule $A \to a\ b\ c$.

For each input position, the algorithm stores a set of these states, along with the position at which the automata was entered. Initially, the algorithm starts with a state before each right-hand side of the starting S symbol, at input position 0. The algorithm then processes the input one terminal at a time, on the basis of all states at the previous input position. First, if the next symbol in any state is a nonterminal, a new state for all right-hand sides of that nonterminal is added at the same position (*prediction*). Second, if the next symbol of any state matches the next terminal, a copy of that state, shifted one symbol forward, is added to the next position (*scan*). Finally, if a state is positioned after all symbols in its sequence, all states at the automaton's starting position who have the rule's nonterminal as next symbol are copied to the next input position, shifted one symbol forward (*completion*). The algorithm avoids duplicating existing states.

The approach looks like a top-down/bottom-up hybrid. Entering new automata is done in a top-down fashion (via *prediction*), but automata skip nonterminals through reductions (*completion*). The Earley algorithm can be seen as a breadth-first version of the naive top-down algorithm, with state deduplication ensuring that the complexity stays polynomial. Conversely, GLR can be seen as an optimization of Earley, where the precomputed tables avoid going down dead ends, especially when the grammar is (mostly) deterministic. Earley's algorithm complexity is $O(n^3)$, and with a few modifications [62, 45] it can run in $O(n)$ over all LR(k) grammars. It is unclear whether GLR is faster than optimized Earley in practice, although that is the commonly held assumption.

The *CYK* algorithm [16] works on grammars written in Chomsky Normal Form (CNF) — all CFGs can be converted to CNF. All rules in a CNF grammar have one of two forms: either $A \to B\ C$ or $A \to a$. The algorithm considers all substrings of the input, starting first with all substrings of size one then incrementally increasing the size. Same-length substrings are processed left-to-right. For each substring, the algorithm then considers all ways the string can itself be partitioned into two substrings. Using this process, it progressively fills a global chart that

remembers which nonterminals matched at which input position, and how long the match was. The chart is initialized with all matches for $A \to a$ rules, which can be found trivially. Afterwards, each substring partition is a chance to find a length-specific match for a $A \to B\ C$ rule. Given that substrings are handled in increasing order of length, if B and C match, the information will already be held within the chart.

In summary, CYK is a relatively straightforward bottom-up dynamic programming algorithm. It has $\Theta(n^3)$ complexity, which makes it markedly worse than other general parsers for many practical (*e.g.*, deterministic) grammars.

2.2 Top-Down Recursive-Descent Ad-Hoc Parsers

We want to briefly touch on an often underacknowledged alternative to using CFGs and assorted parsing tools — and that is, writing a parser by hand directly. Such a parser is called *ad hoc* because it is custom-built for a single language — as opposed to parsing tools that either generate a parser from a grammar, or parse by interpreting a grammar on the fly.

Most ad-hoc parsers are *top-down recursive-descent parsers*. Such parsers are structured as a set of functions — roughly, one for each nonterminal in the grammar, recognizing that particular nonterminal. These functions can call each other, potentially recursively.

The semantics of such parsers — and their differences to that of CFGs — are rather interesting, but we will defer that discussion until we discuss Parsing Expression Grammars (PEGs) in Section 2.4, which are essentially a formalization of top-down recursive-descent parsers.

The only other type of ad-hoc parser that we are aware off are those used to parse infix expressions — which we will discuss in Section 2.5. Both types of ad-hoc parsers are typically used in tandem.

Why do people write parsers by hand? Arguably it's more difficult than writing a grammar (which you will probably need to write anyway — if only for reference purposes) and feeding it to a tool. And yet, most mainstream programming languages use ad-hoc parsers (cf. footnote on page 9)! Here, we recall a few reasons from Section 1.2, which we think explain the relative preponderance of ad-hoc parsers in serious language implementation:

- Ad-hoc parsers are more flexible. In particular, they allow for context-sensitive features in the language.
- Ad-hoc parsers allow emitting more descriptive error messages.
- Ad-hoc parsers can be faster than parsing tools.
- The implementers may have been ignorant about newer and better tools, or such tools might not have existed at the time the parser was first written.

We think that the three first reasons are quite compelling, and these are in fact some of the problems that we tackle in this thesis.

2.3 Parser Combinators

The parser combinator approach is the natural evolution of top-down recursive-descent ad-hoc parsing, as covered in the previous section.

In ad-hoc recursive-descent parsing, we typically write a function to recognize each syntactic element of interest — in the small (*e.g.*, a keyword, a number) as well as in the large (*e.g.*, a whole function definition).

The idea behind parser combinators stems from the recognition that there are typical patterns in how we want to *combine* functions recognizing smaller syntactic elements in order to produce a function that recognize a bigger syntactic element. Typical examples of such combinations include recognizing multiple elements in a sequence, or recognizing one element amongst multiple possibilities.

As such, a parser combinator is a higher-order function that combines one or multiple sub-functions in order to produce a new parsing function.

This does entail a standardization of the format of parsing functions. In order for a parser combinator to be universally applicable, all parsing functions must satisfy the same interface (*i.e.*, have the same type signature). For instance, we might define a parsing function as one that goes from the (remainder of) the input to a binary result (success or failure) along with a suffix of the input — the function having matched the missing prefix. There are other ways to represent this: could pass the whole input and make the current position in the input implicit, for instance. Some approaches [73, 63] propose using a wider set of results,

notably to distinguish between errors that should or should not cause backtracking. We could also include some by-products (typically parse trees) in the result.

The parser combinator approach represents a step away from the fully ad-hoc approach and towards a framework — imposing restrictions in order to foster component reuse. But it is an approach that is still firmly steeped in the code, as opposed from deriving from a more abstract formalism.[11]

The Parsing Expression Grammars (PEG) formalism — which is the object of the next section — draws on the notion of combinator, but brings it back into a well-circumscribed grammar formalism similar to CFG. As explained in Chapter 1, we ourselves endeavour to re-introduce the flexibility of arbitrary code into this paradigm.

Parser combinators are quite old — the first appearance of the concept (though not under that name) seems to be in Burge's 1975 *Recursive Programming Techniques* [12]. Burge uses an unimplemented language based on the lambda calculus, but his approach is surprisingly modern. He imagines multiple interpretations of his combinators, grounded in the parsing paradigms popular and (one must add) practical at the time (LL(1), LR).[12] The approach was later used to parse natural languages [28] and then programming languages [34].

The early history of parser combinators is tightly bound to that of functional programming languages, especially Haskell. This is especially true as it is shown that parser combinators can be expressed elegantly as monads [101]. Beyond elegant mathematics, this enables parser combinators to be able to encode context-sensitive grammatical features — albeit at a cost. This is explored further in Section 5.2.3.

Despite the academic hype, parser combinators saw at first little practical use, as noted by Leijen and Meijer [61] in 2001. The former is the author of Parsec [61] which became the first popular parser combinator library.

[11] This is blurrier a line than many people seem to realize. After all, programming languages *are* formal systems, and the point of standardized interfaces is precisely abstraction. In this thesis, we are very interested in straddling this line.

[12] Interestingly, parser combinators frameworks will largely keep the tradition to not backtrack by default, although it is generally allowed via some special combinator.

Today, most parsing system presented as *parser combinator frameworks* are actually PEG parsing libraries. This is not necessarily a misnomer, but has the important caveat that the set of available combinators is usually (though not always) fixed in advance and hence impossible to extend.

It's hard to explain why PEG parsing libraries became so popular relative to the "classical" combinator approach. In principle, any language that supports higher-order functions and lexical closures (or even approximations thereof, a class that includes C and Java after version 8[13]) is susceptible to implement the combinator approach. We could conjecture that PEG's similarity to existing formal grammars was partly responsible for its success. In any case, the explanation does not seem to be technical in nature. Interestingly, parser combinator libraries based on the CFG formalism also exist [38, 85], though these haven't seen significant traction yet.

We note that the *parsers* manipulated by combinators can be something else than functions, although they must necessarily include a functional component. We already said they could be a monad instance.[14] They could also be instances of an OO-style interface, in which case the combinator is more properly the constructor of the OO-style class implementing the interface.

Finally we note that Definite Clause Grammars (DCGs), which we mentioned in Section 2.1.3, are also an example of parser combinator framework, baked into the Prolog language. In Prolog, DCG clauses are a syntactic facility that maps the semantics of (non-left-recursive) CFGs onto that of the language. Because this mapping is transparent to the user, he may use what the language has to offer, resulting in both an increase in expressive power[15] and expressivity[15] when compared to the CFG formalism.

[13] Java version 8 introduces proper lambda-abstraction to the language.

[14] More precisely, using the Haskell terminology, the instance of a type for which an instance of the *Monad* typeclass exists.

[15] Expressive power being about what is possible to express (*i.e.*, the class of languages that can be represented), while expressivity is about expressing languages effectively (*i.e.*, ease of use, simplicity of formulation).

2.4 Parsing Expression Grammars (PEGs)

Parsing Expression Grammars (PEGs) are another variety of formal grammars introduced by Bryan Ford in 2004 [27], which turns out to be a rediscovery and improvement of parsing formalisms devised in the 70s, namely TS/TDPL and gTS/GTDPL [10].

In this chapter, we will introduce PEGs and carefully contrast then with CFGs. Whereas CFGs are generative — they describe a language and the grammar can be used to enumerate the set of sentences belonging to that a language via substitutions — PEGs are recognition-based: they describe a predicate indicating whether a sentence belongs to the language. In particular, PEGs define a language by specifying a top-down recursive-descent parser that recognizes them.

2.4.1 Expressions, Ordered Choice and Lookahead

PEGs differ from CFGs in two important respects. First, PEGs' productions (rules) are ordered. A PEG parser will try the right-hand sides for a nonterminal in order. If a right-hand side is recognized as a prefix of the remaining input, no other right-hand side (for the same nonterminal) will ever be tried at the same input position. For instance, assuming the rules $A \to a$, $A \to aa$ and $S \to Ab$, a CFG admits the string `aab` in the language, but a PEG does not. Since the first production for A succeeds at the start of the string, the second one is never even tried.

Second, a PEG may contain lookahead operators. The rules' right-hand sides may contains the terms $\&A$ or $!A$, respectively meaning that a prefix of the remainder of the input must or must not match A, but that this prefix is not to be consumed. For instance, given the rules $A \to a$, $A \to b$ and $S \to \&a\ A\ !a\ A$, the only admissible string in the language is `ab`.

PEG means Parsing Expression Grammar, because the rules' right-hand sides are expressions made out of symbols and operators. In reality, this is only a trivial difference to CFGs, because most rules using operators can be trivially desugared to equivalent rules that do not include them. Only the negative lookahead operator (noted `!`) is required. In fact, Ford showed that even this operator could be eliminated, but this requires a whole-grammar transformation [27]. Moreover, most CFG parsing systems allow specifiying CFGs with expressions in a similar manner. Table 2.1 lists all parsing expressions available in PEG as originally defined, along with their precedence. Table 2.2 lists all desugarings for

these same operators.

Expression	Name	Precedence
Nonterminal	Nonterminal	6
(<e>)	Parentheses	6
"string"	Literal String	6
[ab]	Character Class	6
^[ab]	Negated Character Class	6
[a-c]	Character Range	6
^[a-c]	Negated Character Range	6
_	Wildcard	6
<e>?	Optional	5
<e>*	Zero or More	5
<e>+	One or More	5
&<e>	Lookahead	4
!<e>	Forbid	4
<e1> ... <eN>	Sequence	2
<e1> | ... | <eN>	Ordered Choice	1

Table 2.1: List of the different kinds of parsing expressions along with their name and precedence.

In terms of expressivity, there are known languages that can be expressed as a PEG but not as a CFG (such as the language $a^n b^n c^n$ [16] [27]). There are also known languages that can be expressed as a CFG but have no known PEG grammar, though it has not been proved that such PEG grammars do not exist. The most salient example (in fact, the only salient) example is that of palindrome languages, *e.g.*, the language

[16] For the curious reader, the PEG grammar for this language is as follows:
$S \rightarrow (\&A\ !b)\ a+\ B$
$A \rightarrow a\ A?\ b$
$B \rightarrow b\ B?\ c$
See Table 2.1 and Table 2.2 for the meaning of the different operators.

Expression	**Desugaring**
Nonterminal	Nonterminal
(<e>)	$A \rightarrow$ <e>
"string"	$A \rightarrow s\ t\ r\ i\ n\ g$
[ab]	$A \rightarrow a$, $A \rightarrow b$
^[ab]	$A \rightarrow !a\ !b\ _$
[a-c]	$A \rightarrow a$, $A \rightarrow b$, $A \rightarrow c$
^[a-c]	$A \rightarrow !a\ !b\ !c\ _$
_	$A \rightarrow$ disjunction of all terminals
<e>?	$A \rightarrow$ <e> , $A \rightarrow \epsilon$
<e>*	$A \rightarrow$ <e> A , $A \rightarrow \epsilon$
<e>+	$A \rightarrow$ <e> A , $A \rightarrow$ <e>
&<e>	$A \rightarrow !B$, $B \rightarrow !C$, $C \rightarrow$ <e>
!<e>	$A \rightarrow !B$, $B \rightarrow$ <e>
<e1> ... <eN>	$A \rightarrow$ E1 ... EN , $E1 \rightarrow$ <e1> , ... , $EN \rightarrow$ <eN>
<e1> | ... | <eN>	$A \rightarrow$ <e1> , ... , $A \rightarrow$ <eN>

Table 2.2: Desugarings for the parsing expressions in Table 2.1. When the desugaring is a set of productions, the expression should be replaced with the name of the first production (denoted by A). Since expression are recursively nested, expansion is similarly recursive. The fully expanded form does not contain any operators besides negation (!).

described the CFG $A \to a\ A\ a \mid b\ A\ b \mid \epsilon$.[17,18] There does not seem to be practical language idioms that can be expressed with CFGs but not in PEGs.

2.4.2 PEGs and Recursive-Descent Parsers

```
1  // S  ::= AB c
2  // AB ::= A | B
3
4  function parseS (input, pos)
5      pos = parseAB(input, pos)
6      if (pos < 0)
7          return -1
8      if (input[pos] == 'c')
9          return pos + 1
10     return -1
11
12 function parseAB(input, pos)
13     pos = parseA()
14     if (pos >= 0)
15         return pos
16     return parseB()
17
18 parseS(input, 0)
```

Figure 2.4: Naive PEG recognizer pseudo-code implementation for the grammar fragment contained in the top comment. Each function returns the new input position after consuming a match, or -1 if it failed to match.

Because PEGs essentially formalize top-down recursive-descent parsers, they can be straightforwardly mapped to a function-based implementation, as shown in figure Figure 2.4. The key is that any parsing expression can be turned into a single function that calls functions generated for its nonterminals or sub-expressions.

[17]As a simple way to see what the problem might be, consider that the suffix of a palindrome may also be a palindrome. In fact, appending a palindrome to itself always yields a valid palindrome. Because of the single parse rule, we are unable to determine the middle of the palindrome in a simple manner.

[18]Adding context-sensitivity (cf. Chapter 5) solves the issues with palindromes easily: if we can recall the candidate prefix of the palindrome seen so far, we can ensure that the single parse rule does not preclude a match by mandating that any "inner palindrome" be followed by the reverse of the prefix.

This would not be possible with CFGs: if we assume that both A and B can match a prefix of the input, then in the case that the check for 'c' on line 9 would fail, `parseS` would need to explicitly try to match B, then re-try matching 'c', making the control flow much more complex.

2.4.3 The Single Parse Rule, Greed and (Lack of) Ambiguity

The consequence of the two differences between PEGs and CFGs are subtle but crucial. Ordered rules (or, using operators, *ordered choice*) lead to what we call the *single parse rule*: there is at most a single parse for a given nonterminal at a given input position. The single parse rule impacts the backtracking pattern of PEG parsers, as we will see in the next section.

PEG expressions are sometimes said to match *greedily*: they will alway match as much input as possible. For instance, the expression `a*` will always match as many `'a'`s as possible. This means that the expression `a*a` can never succeed on any input. This is a direct consequence of the single parse rule and makes perfect sense if you consider the desugaring of the Kleene star ($*$) operator in Table 2.2.

The single parse rule is both an advantage and an inconvenience. It is an advantage because PEGs are unambiguous[19] by construction: given a valid input, there is only a single correct parse. The disadvantage is illustrated by our previous example that the PEG with rules $A \to a$, $A \to aa$ and $S \to Ab$ does not accept the string `aab`. Roman Redziejowski calls this *prefix capture* or *language hiding* [78]. In this simplistic example, the first rule for A captures the `a` prefix, and “hides” the second rule for A.

In a certain sense, ambiguity and prefix capture are two sides of the same coin, and you cannot have both together. It could be argued that it is quite easy to let a CFG parser disambiguate on the basis of rule ordering, thus avoiding both prefix capture and ambiguity by mimicking PEG only when an actual ambiguity is detected. However, the situation for the end user remains the same: he still has to search for potential ambiguities (like for regular CFGs) and to carefully order the grammar rules (like for PEGs).

[19] Ambiguity is described at the start of Section 2.1.2.

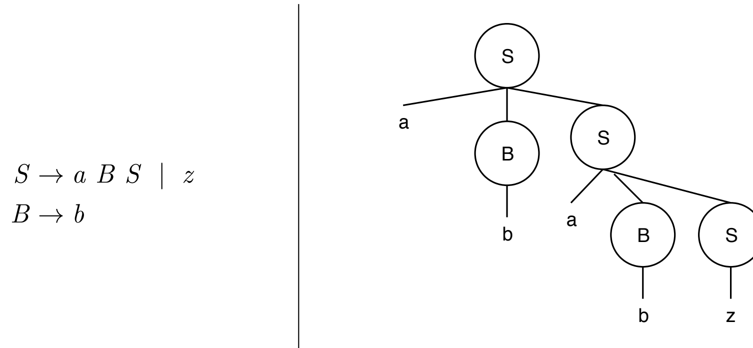


Figure 2.5: Given the grammar on the left, the tree on the right represents the nonterminal expansion of input "ababz".

Finally, we note that determining whether a PEG has prefix capture, or whether a CFG is ambiguous are undecidable problems. However, algorithms exist that can find potential violations [81]. Unfortunately, besides the issue of false positives, the approach relies on involved whole-grammar transformations.

2.4.4 The PEG Algorithm

Top-down recursive-descent parsers are sometimes said to have a *parse stack*. Roughly, the parse stack consists of all nonterminals on which the algorithm may eventually try to backtrack by trying an alternative production. The parse stack only includes nonterminals that have not been fully expanded yet.[20] Because of the single parse rule, these are the only nonterminals for which we are allowed to try alternative productions.

One can also see nonterminal expansions as trees, where each node is a symbol. Nonterminals have for children all the symbols that their expansion generated. This is pictured in Figure 2.5. If we adopt such a view, then the parse stack of a PEG parser is always a path from a symbol towards the root of the expansion tree (the starting nonterminal S). PEG parsers only backtrack vertically in the tree, along this path. In Figure 2.5, while parsing the second "b", the PEG parse stack would be $[B, S, S]$.

CFG parsers, on the other hand, must consider the alternative expansions of nonterminals that have already been fully expanded. The parse stack of

[20]Given the top-down algorithm, this synonymously means the nonterminals that have not been fully matched to part of the input yet.

a naive top-down CFG parser (such as the one listed in Figure 2.2) includes all nonterminals that have been expanded (fully or not). PEG parsers backtrack only vertically, but CFG parsers also backtrack laterally. In Figure 2.5, while parsing the second "b", the naive CFG parse stack would be $[B, S, B, S]$, where the second B corresponds to the first (leftmost) expansion of the B rule.

Figure 2.6 shows pseudo-code for a simple PEG recognizer (using the minimal form with only negative lookahead). This code is very similar to the naive top-down CFG recognizer shown in Figure 2.2. Because they are so close, we highlighted the key differences using asterisks ($*$).

Just like the naive CFG recognizer, the PEG recognizer takes a list of symbols (initially only the starting symbol S) and the input (a list of terminals). We also systematically shed the common prefix of our sequence and our input string. However, the PEG parser returns two values. In addition to a verdict (`accept`, `reject`) that indicates whether a prefix of the input matched the list of symbols, it also returns the (unmatched) rest of the input.

The new algorithm adds a clause to the if-statement to deal with the not-predicate (negative lookahead). But the most significant changes are on lines 24 and 26: instead of recursing on the rest of the symbols, we only recurse on the expansion of the first nonterminal. If that succeeds, only then do we recurse on the rest of the input. This basically enforces the single parse rule: the nonterminal is matched to an alternative, and backtracking won't be able to change it like it could in the CFG parser.

We can now precise the notion of parse stack: it corresponds to the function call stack in these algorithms. For the CFG recognizer, there is one function call on the call stack per nonterminal expansion. In the PEG parser, however, the call stack only contains function calls for nonterminals that have not been fully expanded yet.[21]

2.4.5 Packrat Parsing

The simple PEG parsing algorithm presented before has worst-case exponential complexity. In practice, exponential run times almost never happen, but other inefficiencies might occur. We discuss the topic further in Section 6.1, which includes the presentation of a particularly worrying

[21] In both cases, there are also functions call for matching terminals, but those are of no interest to us.

```
1  function parse (symbols, input)
2
3      if (symbols is empty)
4          if (input is empty)
5              return [accept, input]
6          else
7              return [reject, input]
8
9      else if (symbols.head is a terminal)
10         if (input is empty || input.head != symbols.head)
11             return [reject, input]
12         else
13             return parse(symbols.tail, input.tail)
14
15 *   else if (symbols.head is a not-predicate)
16 *       [result, _] = parse(symbols.head.operand, input)
17 *       if (result == accept)
18 *           return [reject, input]
19 *       else
20 *           return [accept, input]
21
22     else if (symbols.head is a nonterminal)
23         for each alternative (symbols.head -> rhs)
24 *           [result, input_leftover] = parse(rhs, input)
25             if (result == accept)
26 *               return parse(symbols.tail, input_leftover)
27         return [refuse, input]
28
29 parse(list(starting_symbol), input)
```

Figure 2.6: Pseudo-code for a PEG top-down recursive recognizer, using the minimal PEG formalism. Notable differences with the algorithm from Figure 2.2 have been indicated with asterisks ($*$).

inefficiency that we discovered in a common grammatical idiom.

One of the initial draws of PEGs was another algorithm devised to parse them with $O(n)$ complexity. The algorithm, dubbed *packrat parsing* [26] simply consists of memoizing the result of each parsing expression invocation. Indeed, the single parse rule guarantees that there is at most a single parse for a given parsing expression at a given input position. Using this technique, each parsing expression can be invoked at most once at each input position — and since the number of parsing expressions in a grammar is fixed, the algorithm is linear in the size of the input.

Packrat parsing is not perfect however. Naive packrat parsing may consume a lot of memory. Storing and retrieving the memoized matches has a high overhead, which makes packrat parsing less attractive than its theoretical properties would suggest.

In particular, packrat parsing performs poorly compared with linear-time CFG parsers and naive PEG parsers, when using a grammar designed specifically for those parsers. Those grammars are designed to limit backtracking, and as such benefit very little from memoization [8].

Another pitfall of packrat parsing is that it interacts poorly with parse state. If the result of a parsing expression depends on something else than the input position, then it cannot be memoized (alternatively, the dependency must be identified and become part of the memoized identity). This also recursively affects all parsing expressions using the offending expression.

2.5 Expression Parsing

If there is one area that is problematic in programming language parsing, it is probably parsing expressions made out of operators and values (that are to serve as operands), especially when infix operators are involved. The prototypical example is arithmetic expressions on assignable variables, which feature a mix of binary infix, prefix and postfix operators; as well as some operator overloading: `+` and `-` are both infix and prefix operators, `++` is both prefix and postfix. There are other examples: the syntax of regular expressions and PEGs, the syntax of types in some languages, ...

Expressions are hard to parse because they are inherently ambiguous. Should `(1 + 2 + 3)` parse as `((1 + 2) + 3)` or as `(1 + (2 + 3))`? This corresponds to the notion of associativity, and the two proposed answers

are respectively left- and right-associative. Similarly, should `(1 * 2 + 3)` parse as `((1 * 2) + 3)` or as `(1 * (2 + 3))`? This corresponds to the notion of precedence. In the first proposed answer, `*` has more precedence than `+`, while the contrary is true in the second answer.

To get an unambiguous expression syntax, you need a set of values, operators and their associated precedence and associativity, as well as some additional restrictions. Typically, operators with the same precedence should have the same associativity, and be of the same kind (infix, prefix, postfix), lest some ambiguity remains.

Practical expression syntax has to contend with some additional difficulties. Typically, an escape hatch for precedence is provided under the form of parens operator (`()`). There might also be additional syntactic restrictions on the value that some operators may take as operands.

Another difficulty: Infix operators are not necessarily binary — they could be ternary, or more. For instance, C-like languages often have a conditional ternary operator of the form [`<cond> ? <value1> : <value2>`]. Such operators are sometimes called *mixfix* operators. These can cause additional ambiguities, but the issue is often avoided by making sure that the operator parts (*e.g.*, `?` and `:`) preclude "middle-associativity", leaving just left- and right-associativity as options and letting the infix operator be treated as a binary operator. For instance, `(a ? b : c ? d : e)` can be interpreted as `((a ? b : c) ? d : e)` or `(a ? b : (c ? d : e))`, but on the other hand, `(a ? (b : c ? d) : e)` is syntactically invalid. Provided there are no further ambiguities with other operators, a sufficient condition to preclude middle-associativity this is to pick distinct symbols for each part of the mixfix operator.

Putting all this together, we can see why expression parsing is such a thorny problem. Over the years, a few dedicated algorithms have been devised. Those are in fact necessary in order to parse expressions alongside a top-down recursive descent parser.

The first such algorithm is Dijkstra's shunting yard algorithm [22]. The algorithm works a bit like a shift-reduce parser, excepted it is driven by tables of operator precedence and associativity rather than by parse tables derived from a grammar. Whenever the algorithm finds an operator of lower-precedence that the previous one, it performs a reduction on this higher-precedence operator and its operands.

Another algorithm is *Top Down Operator Precedence* or TDOP for short, devised by Vaughan Pratt [75]. Implementations of the algorithm are sometimes also called *Pratt parsers* or *precedence climbers*. Here, the algorithm is implemented as a function that is responsible to parse one (sub-)expression and calls itself recursively to parse its sub-expressions. The trick is to use lookahead to find the next operator and — based on its precedence or associativity — decide whether to call the function recursively to parse the expression tree rooted at that operator, or to use the preceding value as the right-hand side of the previous operator, and use the resulting tree as the left-hand side of the operator ahead.

It's interesting to note that the shunting yard algorithm and TDOP are essentially equivalent: the shunting yard algorithm simply replaces the recursion from TDOP with an explicit stack.

Of course, expressions can be parsed with regular grammar-driven parsers. In particular, LR(1), LALR, and all general parsers will accept the *classical expression encoding*, which is shown in Figure 2.7a.

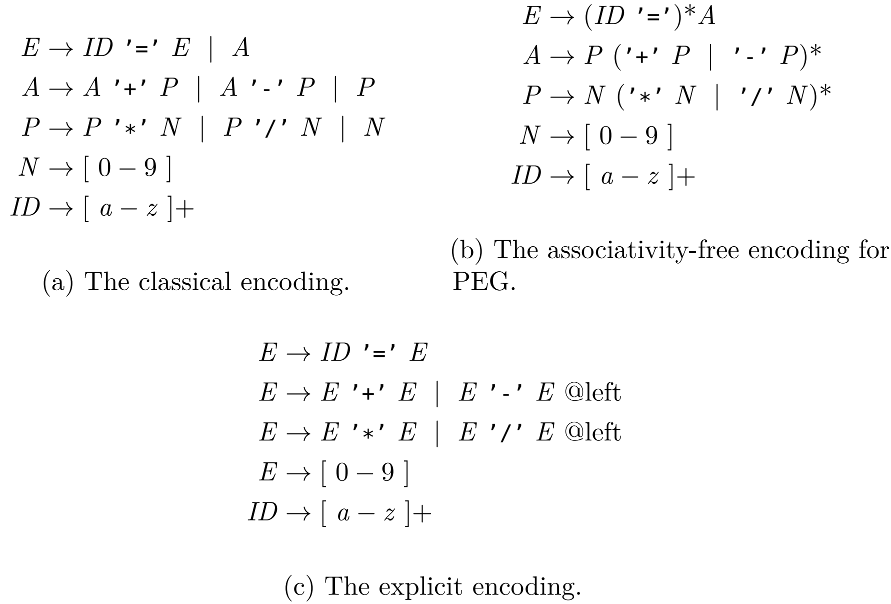
The classical encoding encodes precedence by associating each precedence level with a nonterminal, and referring to these nonterminals from the nonterminal representing the previous (lower) precedence level. Associativity is encoded by either direct left- or right-recursion.

LL does not allow the classical encoding, because it is left-recursive. Neither do classical PEG parsers, but PEG parsers with left-recursion support will (cf. Chapter 4).

Instead, LL and PEG parsers often adopt an alternate encoding, which destroys the associativity information. The operators and operands at the same level of precedence are parsed as a list of terms, and the correct associativity is reclaimed by post-processing the generated parse tree. This encoding is shown (for PEG) in Figure 2.7b.

Alternatively, LL and PEG parsers can go hybrid and embed one of the previously mentioned expression parsing algorithms. To cite Jeffrey Kegler's *Parsing: a timeline* [47]:

> But recursive descent does have a huge advantage, one which, despite its severe limitations, will save it from obsolescence time and again. Hand-written recursive descent is essentially calling subroutines. Adding custom modification to recursive descent is



(a) The classical encoding.

(b) The associativity-free encoding for PEG.

(c) The explicit encoding.

Figure 2.7: Three ways to encode an expression language with right-associative assignments and left-associative arithmetic operators.

very straight-forward.

Some systems [72] allow encoding associativity and precedence explicitly, or implicitly using rules ordering. In this encoding, a single nonterminal can be used for all expressions. This is shown in Figure 2.7c: the rules are listed in order of increasing precedence, and an explicit annotation indicates that sums and products should be left-associative.

Finally, some programming languages (*e.g.*, Scala, Haskell, Swift and Prolog) enable their users to define custom operators, sometimes alongside with their precedence and associativity (*e.g.*, Haskell, Swift). Such a scheme cannot be encoded in a grammar; but expression parsing algorithms can support it, through dynamic modification of the precedence and associativity tables that guide the algorithms.

2.6 Error Reporting

This section investigates the literature on error-reporting strategies. The goal of this investigation was to identify potential strategies that could be implemented Autumn. Section 6.4 will discuss feasible error-reporting and

error-recovery strategies for Autumn, which includes both adaptations of classical techniques presented in this section, as well as novel ideas.

In a parser, an *error* occurs whenever the input cannot be matched against the language it is supposed to conform to. Absent a bug in the grammar, this happens because some things — or potentially, many things — in the input are wrong, probably as a result of a mistake made by the programmer. The role of error reporting is then to help the programmer to spot the mistake and possibly correct it. From now on, we will use both *error* and *mistake* interchangeably.

The notion of *mistake* is imbued with intent. It is impossible to say with certainty what constitutes a mistake. In fact, code that parses may still contain mistakes — which in that case simply resulted in code that was still syntactically valid. When error-reporting, we are thus always trying to point out the most likely mistake(s) made by the programmer that lead to the input being syntactically invalid. Anything we will do will necessarily be heuristic in nature.

2.6.1 Overview

In order to report an error, the parser must first pinpoint the location of the error, and then generate an error message that informs the user about the location and the potential mistake that was made here.

Pinpointing an error is trivial when using deterministic parsers (and hence, deterministic languages). These parsers run linearly on the input, and so it suffices to wait until they become stuck and use that location. In general CFG and PEG parsers, things are subtler, and implementers often resort to the *furthest error heuristic*, which we will discuss in Section 2.6.2.

Generating a good error message is hard. Broadly, existing parsers fall into two categories: those that report errors in terms of expected or unexpected terminals at the error site; and those that manually associate custom error messages with specific error scenarios.[22] In the second case, the context available to generate the error message is often limited, and this is a major reason why many languages use ad-hoc parsers (cf. footnote on page 9) — as they make it easier to generate informative error messages. There is preciously little research on how to generate

[22]As there are too many possible errors to encode all of them, this approach usually falls back to reporting (un)expected tokens.

better error messages, and the techniques are mostly concerned with associating errors with custom messages. We will review such techniques in Section 2.6.3.

To the best of our knowledge, there are no approaches that automatically report errors in terms of higher-level concepts (*i.e.*, nonterminals instead of terminals). At best, one will be given the content of the parse stack or trace. This observation was one of the main motivation behind our *Red Shift parsing* [55] approach. Red Shift parsing is explained in Section 6.4, which will also explain how the idea can be adapted in Autumn.

So far, we have only talked about reporting a single error. Realistically, it's going to happen that the programmer has made multiple successive mistakes. It could be interesting to report mistakes past the first one.[23] We report on the rich literature on *error recovery* in Section 2.6.4.

2.6.2 The Furthest Error Heuristic

In a top-down parser, *potential mistakes* are easy to find. Remember (Section 2.1.3) that a top-down parser starts from the starting symbol and progressively expands nonterminals while matching the terminal prefix of our expanded sequence against the input. Whenever a mismatch between the terminal prefix and the input occurs, we have a potential mistake.

Even correct inputs will generate a lot of potential mistakes, because of choice: it is likely that most right-hand sides for a given nonterminal expansion do not match the input. Whenever the parse fails, however, we do need to report something, and there are way too many potential mistakes to report all of them.

The most popular heuristic is to report the *furthest error*, *i.e.*, the one which occurs at the furthest position in the input. In practice, this heuristic works very well [25], as programming languages tend to be highly deterministic.

Somewhat surprisingly, the same thing works well for bottom-up parsers (Section 2.1.4). The reason is that these parsers look at the stack to determine what right-hand side to use, and so are guided by the previous

[23] Personally, we are not convinced this is super interesting from a user experience perspective, but we are not aware of any user study on the topic, and many mainstream compilers do report multiple errors.

reductions living on the stack. In fact, simple LR parsers do not backtrack and so the first error is always also the furthest error. For general LR parsers like GLR, it suffices to pick the furthest error amongst the divergent executions encoded in the Graph Structured Stacks (GSSs).

2.6.3 Associating Errors With Custom Messages

Example-Based Error Messages

Mapping parse errors to error messages can be done through *error productions* encoded in a grammar.[24]

However, Clinton Jeffery [39] notes that adding explicit recovery rules in an LR grammar makes the grammar less readable and reusable. He proposes writing examples of syntactic errors, along with a corresponding error message. These examples are then mapped to (erroneous) LR parse states; and the parts of the error message that refer to the particulars of the example are made generic, in order to accommodate similar errors. For instance, an identifier from the example appearing in the error message will act as a placeholder for any identifier that might occur in that context. The parser is modified so that, whenever the parse state in question is encountered, it can emit an adapted version of the error message. The approach enables incremental improvements of error messages by encoding common error cases separately from the grammar itself. This method was used in the Go language compiler for a while [18], but it was phased out in favor of ad hoc error handling.

Pottier [74] expands on Jeffery's approach by proposing an algorithm that is able to generate a minimal input sentence for every LR error state, hence making it possible to maintain a *complete* collection of *(error, message)* pairs. The messages still have to be written manually.

Labeled Errors in PEGs

Unlike for CFGs, literature on error reporting in PEGs is relatively scarce. In general, implementers have been relatively content to stick with the furthest error heuristic. The only significant alternative proposal was made by Maidl *et al.* [63]. They add labeled failures to the PEG formalism: each parsing expression invocation succeeds consuming some input or fails with a label. The default failure label is `fail` which preserves the default semantics. Other (custom) labels are propagated up the parse

[24] These may also be used for error recovery, as we will see in a moment.

stack, but can be caught through a specially-annotated choice expression. By annotating expressions that should not fail with a special label (for instance, in many languages an expression must always follow the `if` keyword), it becomes possible to more accurately pinpoint errors without relying on the furthest error heuristic. Moreover the mechanism can be used to supply custom error messages, in the fashion of error productions. The system can be seen as a variant of the *PEG cut operator* as devised by Mizushima [65], which was inspired by the Prolog cut operator.

2.6.4 Error Recovery

In the presence of a syntactically erroneous input, all parsers will eventually become stuck short of their ultimate goal (matching the input to the grammar), unable to take further processing steps. Most parsers proceed linearly along the input (possibly going back if necessary, *e.g.*, via backtracking). Because of this, they will become stuck at the furthest error position — typically, once all potential right-hand side expansions (and combinations thereof) have been tried. At this point, if we hope to discover further errors in the input, it is necessary to correct — or at least recover from — the error.

We will distinguish between correcting and non-correcting error-recovery schemes. Error correction consists of changing the input in order to make it syntactically valid — or more accurately, to let the parser process past the current error position. Besides making it possible to report multiple errors, this is also valuable in its own right — the correction can be suggested to the programmer. Often — but not always — these corrections are based on the insertion or deletion of terminals at or around the error position, as predicted by failed nonterminal expansions. [21]

Non-correcting error recovery is an alternative approach: instead of correcting the error, we seek to skip past it, and to *resynchronize* the parser to the input.

The seminal "dragon book" [3] presents four major families of error-recovery methods. We present them here briefly. Our presentation is also informed by the review of error-recovery schemes for LR parsers published by Degano *et al.* in 1995 [21]. It offers a comprehensive overview of the topic up to that date. More recent developments will be discussed later. LL parsers have been historically less used, and error recovery in those parsers has been less studied; but broadly, the same principles can be applied.

Panic-Mode Recovery

In panic-mode recovery, we skip terminals from the input until we encounter one that belongs to a set of *synchronizing symbols*. Typically those will be closing parens (*e.g.*, `}` or `)`) or other delimiters (*e.g.*, `;` or `,`). The big question here is which nonterminal to consider as failed, *i.e.*, where to resume our expansion process. This can either be encoded manually[25] (by using recovery rules, see in a moment), or heuristically. One heuristic that works well is considering the nonterminal that precedes the occurrence of the synchronization symbol in a right-hand side as failed. In LR, this can sometimes be achieved by popping symbols from the stack [21].

Phrase-Level Recovery

This is a form of error correction performed in two steps. In the first step, a context (a chunk of the input, *a phrase*) containing the error is isolated. In the second step, a correction of the phrase is attempted and validated.

Isolating the context is done by gathering symbols on the right (in the remainder of the input) according to a predetermined strategy. It is also possible to collect context on the left — in LR parsers this would mean on the stack. The latter is often avoided, as it requires *unparsing* nonterminals into their terminal components. Parsing these terminals might have entailed the execution of semantic actions, which subsequently need to be undone.

In the second step, a correction is attempted. Typically, these corrections will be predicated on the parser's knowledge of previously failed nonterminal expansions: the phrase is modified so that a previously-failed nonterminal expansion may now succeed. As multiple corrections may be possible, the best one must be selected. An effective criterion is to use the correction that maximizes the input consumed until the next furthest error position — or, if any, the first correction that lets the parse succeed.

Historically, trying all possible corrections was both complex and costly. Instead, parsers often limited themselves to *local recovery* techniques, which only modify the phrase by adding or removing symbols at the start of the remainder of the input, typically putting a bound on the number of such operations — often even to just one substitution, insertion or

[25] This is notably a possibility in the GNU Bison LALR/GLR parser.

deletion. These historical algorithms also allow changing the left-context, but only for terminals that have not been reduced into a nonterminal yet. The most famous algorithm to perform such corrections is the Burke-Fisher error repair algorithm [13], which considers only the error symbol and the terminal part of the left-context, bounded to a fixed number k.

The more general forms of phrase-level recovery are often called *region recovery* instead. These occupy the middle spot between local and global recovery methods (see later). There are a lot of simple but effective heuristics that can be used to delineate regions of interest, such as using matching pairs of parens (*e.g.*, `()` or `{}`), or even indentation [19]. Note that, somewhat counterintuitively, these methods do not usually outperform the local approaches, as noted by Degano *et al.* [21].

Additionally, phrase-level recovery methods (as well as other methods we will see later) are liable to loop indefinitely in some cases. This must be detected by effectively bounding the size of the correction, and falling back on a secondary recovery method (such as panic mode) when the bound is reached.

While we mostly focus on LR parsers here, the same strategy is straightforwardly applicable to LL and other top-down parsers. The major difference is that if unparsing occurs, it must modify the parse tree under construction instead of the parse stack.

Error Productions

Sometimes also called *error-recovery rules*, this strategy consists of annotating grammar rules with hints on how to handle an error once it is encountered. This could consist of instructions for the other error-recovery strategies, such as which symbol to skip to (panic-mode recovery), or which symbol to insert (phrase-level recovery); but it could also consist of more detailed instructions of what to do with the input and the rest of the parse state.

An example of interest is the Marpa [45] Earley parser (cf. Section 2.1.5), which enables a technique called *ruby slippers*.[26] The idea is to associate a semantic action to some grammar rules, which enables the generation of virtual tokens. This can be used as an error-correction mechanism, by generating missing tokens; or as a way to enable some context-sensitive

[26] A reference to *The Wizard of Oz* where the heroin uses magical ruby slippers to be transported back to our world.

features, such as semicolon insertion or layout-sensitive parsing. [46]

Global Correction

The idea of this technique is to find the minimal correction that can be applied to the whole input in order to make it syntactically correct. Because this requires a whole reparse of the input in order to validate each candidate correction, this technique is very expensive and not used in practice. However, the minimally-corrected input does provide a nice benchmark against which to compare other forms of error correction.[27]

Non-Classical Approaches

We now step beyond the classical error-recovery approaches to mention a few exotic methods and recent developments.

Richter's Non-Correcting Approach

One of the most promising error-recovery approaches is that developed by Helmut Richter. In his paper [79], he advocates a non-correcting recovery approach on a *suffix analysis* of the input string. In brief, the suffix analysis of a string is a sequence i_x of indices in the string such that $[i_{j-1} : i_j]$[28] is not a subword (a substring) of a sentence in the language and i_j marks the first conflicting symbol.

This method has very good empirical results. It particularly shines when source code has been mangled due to large erroneous cuts or pastes — cases where error corrections are often impossible.

Unfortunately, the method relies heavily on being able to determine if a subword of the input is a subword of the language. The naive ways of doing this are very expensive, but the operation can be made tractable if we have a recognizer for the corresponding suffix language — the language containing all the suffixes of the sentences in the original language. This recognizer must have the *correct prefix property*, meaning it should be able to indicate the longest prefix of the input that is a suffix of a sentence in the language; the reason being that a subword is precisely the prefix of a sentence's suffix. It is possible to automatically derive a grammar for the suffix language from the original grammar, but the resulting grammar might lose properties of interest, such as determinism. At the time the

[27]Note however that the minimal correction is not always the best correction.

[28]i_{j-1} and i_j are both indices in our sequence. i_{j-1} should not be confused with $i_j - 1$.

approach was proposed, that made it unpractical, and to the best of our knowledge, it was never implemented.

Semi-Parsing

Semi-parsing is a set of techniques aimed at parsing only part of an input — in some case, specifically its *correct* parts. While the goal is not strictly speaking error reporting, these methods can be employed — by reversal — as techniques for error-recovery. Vadim Zaytsev wrote a good survey of the domain [108]. Some techniques of interest include island grammars [66] and its derivatives lake and bridge grammars [68].

An island grammar precisely specifies the syntax of only part of the input (the *island*) which is surrounded by input specified loosely, or not at all (the *water*). In lake grammars, the islands are allowed to contain pockets of water. In bridge grammars, the relationship between islands (in particular in terms of *scopes*) is uncovered using salient indicators found in the water (*reefs*).

Automatic Derivation of Recovery Rules

More recently, Visser *et al.* have worked on integrating various earlier techniques and making them robust to modern parsing use cases, such as grammar composition [19]. Their approach automatically derives recovery rules based on the grammar, using ideas from island grammars [66]. The indentation of input files is used to improve the quality of error recoveries inside nested structures.

The automatically generated recovery rules allow skipping unrecognized "tokens"[29], inserting certain literals such as curly braces (`{}`) and semi-colons, and closing comments and string literals. These are heuristical in nature — based on common programming language syntax — and remind us of some phrase-based recovery approaches. Furthermore, the generation of these recovery rules are guided by a set of heuristics. For instance, braces and parens are only inserted in order to balance the count of opening and closing braces or parens. It is furthermore possible to customize these rules and heuristics.

[29] Quoted because Spoofax, the framework implementing the approach is scannerless: it does not feature a separate tokenization step. Nevertheless, the approximation holds.

2.7 Further Related Work

In this chapter, we covered broad background information on parsing techniques and various associated concepts (ambiguity, determinism, greed, backtracking, ...). We also included a survey of error-reporting and error-recovery techniques.

This chapter does not constitute an exhaustive related work for the research exposed in this thesis. Relevant state-of-the-art research will be cited next to our related contributions in each chapter. In particular, Section 4.11 presents related work on left-recursion and infix expression parsing in PEG; while Section 5.2 covers the state of the art in context-sensitive parsing.

Chapter 3

Autumn's Basics

Now that we have caught up on the background in the field of parsing, and in PEG and combinator parsing in particular (cf. Section 2.4 and Section 2.3), we want to introduce the basics behind the inner workings of our *Autumn* parsing tool.[1]

Autumn is the vehicle we use to demonstrate the various parsing improvements presented in this thesis. Moreover, it was also our means to inquire about the engineering of parsing tools, and now demonstrates how the various improvements we introduce — but also various techniques previously presented in the literature — can be built on top of the same core framework, and can interact with one another.

In particular, Autumn is a parser combinator framework written in Java that can be seen as an extensible superset of PEG. Autumn's set of built-in parsers can be extended by the user[2] using arbitrary Java code. Autumn supports left-recursion (Section 4.5), associativity selection for infix operators (Section 4.9), context-sensitive parsing through recall (Chapter 5), memoization (Section 6.1.4), optional seamless lexing (tok-

---

[1]Autumn is available online at `https://github.com/norswap/autumn`

[2]Let it be said once and for all that when we say "the user" we mean the programmer who wants to add a parsing component to his program — and is expected to write a grammar to achieve this. There are a few other interesting actors: a selfsame programmer that would use a grammar written by someone else, or a programmer that would use a library with a parsing component and be confronted, for instance, with parse errors. We'll refer to these specific use cases where appropriate.

enization, Section 6.3), and can parse both text strings and lists of objects. Autumn also has built-in support debugging and tracing (Section 6.6).

All this is achieved with a code base that is both small (less than 5000 significant lines of Java code) and well-documented: over 3000 lines of comments, which include a full Javadoc reference of the public API. A full user manual is also available online.

One of the reasons that things are able to fit together so well is that Autumn evolved constantly over the course of the thesis, undergoing no less than three full rewrites for a total of four versions. Particular attention was paid to the user experience of the tool, and some aspects of the API were constantly improved in that respect. For instance, the formulation of left- and right-associative operators' syntax became markedly less awkward and more consistent over time. In the remainder of the thesis, we will try to distill some of the lessons gleaned from Autumn's evolution.

The current chapter serves to make the reader familiar with Autumn's syntax, and to introduce its fundamental principles regarding basic parsing operations, namely recognizing text using simple PEG-style grammars and generating an abstract syntax tree.

This chapter will not review the principles of parser combinators and PEG, so it is recommended to have read Section 2.3 and Section 2.4 beforehand.

Many details — especially regarding the parsing improvements presented in this thesis — will be given in subsequent chapters.

3.1 Introductory Example

As an introduction to Autumn, let us explain a simple but non-trivial grammar. At first we will only define a recognizer, but then we will extend it to produce an AST in Section 3.3. In particular, this section will familiarize yourself with Autumns's Domain Specific Language (DSL) used to define grammars, as well as the meaning of some common combinators.

The language for which we will write a grammar is JSON (Javascript Object Notation), according to the specification available on the official

JSON website.[3]

```
 1  Integer      ::= 0 | [1-9] [0-9]*
 2  Fractional   ::= '.' [0-9]+
 3  Exponent     ::= [eE] [+-]? Integer
 4  Number       ::= '-'? Integer Fractional? Exponent?
 5  HexDigit     ::= [0-9] | [a-f] | [A-F]
 6  StringChar   ::= !["\] ![\u0000-\u001F] .
 7                 | \ [\/bfnrt]
 8                 | "\u" HexDigit HexDigit HexDigit HexDigit
 9  String       ::= '"' StringChar* '"'
10  Value        ::= String
11                 | Number
12                 | Object
13                 | Array
14                 | "true"
15                 | "false"
16                 | "null"
17  Pair         ::= String ':' Value
18  Object       ::= '{' (Pair (',' Pair)*)? '}'
19  Array        ::= '[' (Value (',' Value)*)? ']'
20  Document     ::= Value
```

Figure 3.1: EBNF grammar of the JSON language.

Figure 3.1 displays the JSON grammar using the EBNF notation for CFGs [37]. In case your are not familiar with the notation, here is a description of the grammar in English:

- An integer is 0 or a sequence of one or more digits that does not start with 0.
- The fractional part of a number is a dot followed by a string of digits.
- The exponent part of a number is 'e' or 'E' optionally followed by '+' or '-', followed by an integer.
- A number optionally starts with '-', then has an integer, then an optional fractional part and an optional exponent part.
- - A string character is any Unicode character, but not a double quote

[3] https://www.json.org

("), a backslash (\), nor anything in the range `[\u0000-\u001F]` (which are control characters). Alternatively it can also be a backslash followed by another backslash, '/', 'b', 'f', 'n', 'r' or 't' (named character escapes), or the a backslash followed by 'u' then four hexadecimal digits.

- A hexadecimal digit is a letter between '0' and '9' or between 'a' and 'f', or between 'A' and 'F'.

- A string is zero or more string characters enclosed between double quotes (").

- A value is a string, a number, an object (see below), an array (see below), or the words "true", "false" or "null".

- A pair is a string followed by a colon, followed by a value.

- An object is a (possibly empty) sequence of pairs, separated by commas and enclosed between curly brackets ({}).

- An array is a (possibly empty) sequence of values, separated by commas and enclosed between square brackets ([]).

- A JSON document is comprised of a single JSON value.

However, this EBNF grammar still does not fully specify the language. We still need to specify where whitespace is allowed. In JSON, whitespace is allowed after all brackets, commas, colons and values. Whitespace is comprised of spaces, tabs, newlines (\n) and carriage returns (\r).

The listing below shows the Autumn version of the JSON grammar. The correspondence with the EBNF grammar is pretty direct, although some changes are required. The only meaningful grammatical difference is that the `integer` rule is simpler, owing to PEG's ordered choice semantics. Regarding the notation and other peculiarities, a full commentary follows.

```
1 import norswap.autumn.Autumn;
2 import norswap.autumn.DSL;
3 import norswap.autumn.ParseOptions;
4 import norswap.autumn.ParseResult;
5
```

```
6  public final class JSON extends DSL
7  {
8      { ws = usual_whitespace; }
9
10     public rule integer = choice(
11         character('0'),
12         digit.at_least(1));
13
14     public rule fractional =
15         seq(character('.'), digit.at_least(1));
16
17     public rule exponent =
18         seq(set("eE"), set("+-").opt(), integer);
19
20     public rule number =
21         seq(character('-').opt(), integer,
22             fractional.opt(), exponent.opt())
23         .word();
24
25     public rule string_char = choice(
26         seq(
27             set('"', '\\').not(),
28             range('\u0000', '\u001F').not(),
29             any),
30         seq(character('\\'), set("\\/bfnrt")),
31         seq(str("\\u"), hex_digit, hex_digit, hex_digit, hex_digit));
32
33     public rule string =
34         seq(character('"'), string_char.at_least(0), character('"'))
35         .word();
36
37     public rule value = lazy(() -> choice(
38         string,
39         number,
40         this.object,
41         this.array,
42         "true",
43         "false",
44         "null"));
45
46     public rule pair =
47         seq(string, ":", value);
48
49     public rule object =
50         seq("{", pair.sep(0, ","), "}");
51
52     public rule array =
53         seq("[", value.sep(0, ","), "]");
```

```
54
55     public rule root = seq(ws, value);
56
57     { make_rule_names(); }
58
59     public ParseResult parse (String input) {
60         return Autumn.parse(root, input, ParseOptions.get());
61     }
62 }
```

3.1.1 `DSL`, `rule`, parsers and combinators

First, notice we inherit from `DSL`. `DSL` (for Domain Specific Language) is a base class that contains a slew of methods which we will use to define our grammar.

`DSL` also defines the `rule` class, which represents a rule in our grammar.

In reality, `rule` is merely a wrapper around the more fundamental `Parser` class. It defines methods that help construct new rules (hence, parsers). So for instance, in the `integer` rule, you have `digit.at_least(1)`. `digit` is a rule predefined in `DSL` (as is `hex_digit`, both by virtue of being ubiquitous in programming languages), and `at_least` is a method in the `rule` class that returns a new `rule`. Here, the `integer` rule matches as many repetitions of `digit` as possible, with a minimum of one.

This is a pretty common way to build up objects in object-oriented programming — it is known as *the builder pattern* [29].

In practice we will call those things that have type `rule` or `Parser` *"parsers"*. Parsers can be combined into bigger parsers, such as in `digit.at_least(1)`. This returns a rule wrapping a parser with type `Repeat` (a subclass of `Parser`). We say that `digit` is a *sub-parser* (or *child parser*) of digit.at_least(1).

We also say that `at_least` is a *parser combinator* (or *combinator* for short). A parser combinator takes one or multiple parsers as argument(s) (in our example, just `digit`) and combines or wraps them into a higher-level parser.

In practice, we will reserve the word *"rule"* for parsers that are assigned to a field in our grammar — and hence prefixed with the type `rule`.

The `{ make_rule_names(); }` bit is an instance initializer that, for each `rule` (actually each `Parser`) that has been assigned to a field, assigns the rule a printable name corresponding to the name of that field. This makes for much more pleasant error output.

3.1.2 Whitespace Handling & String Literals

Consider the `{ ws = usual_whitespace; }` initializer at the top of the grammar. `ws` is a field of `DSL` that designates the rule to be used for parsing whitespace (if it is `null` — which it is by default — then no special whitespace handling is performed). Here we assign it the predefined `usual_whitespace` rule, which conveniently matches that of JSON.

Where does this whitespace come into play? In all parsers created by a combinator called `word`. The `word(String)` version returns a parser that matches the specified string and any subsequent whitespace. The `rule#word()`[4] version matches what the receiver matches, followed by any whitespace.

You will notice that some of the combinators (*e.g.*, `choice` in the `value` rule) are passed string literals directly. These string literals are implicitly converted into parsers by applying them the `word(String)` method.

It is also possible to use the `ws` rule directly, as we do in the last (`root`) rule, because we want to match whitespace before our JSON value as well.

We note that as a rule, `ws` must always succeed — it should be able to succeed while matching no input, which is typically achieved by wrapping a simple whitespace rule with `rule#at_least(0)`. For instance, `usual_whitespace` is defined as `set(" \t\n\r").at_least(0)`.

3.1.3 `lazy` and `sep`

You should be able to tell what most of the methods do by comparison with our previous descriptions of the grammar (in EBNF and English). There are a couple of methods that may appear more mysterious, however.

First, there is that `lazy` method taking a lambda function in the `value` rule, and how we qualified `object` and `array` with `this` in that rule.

---

[4]The `X#y` notation denotes that `y` is a field or method of class `X`.

`lazy` returns a parser that will be initialized when first used, based on the lambda that it was passed. The reason we need `lazy` is that there is recursion in the grammar: an array may contain values, but a value may itself be an array! Because we use fields to store our rules, a rule's definition cannot reference fields that are defined after: when the rule is initialized, these fields will not have been initialized yet!

And that's why we use `lazy` to defer the initialization process. `lazy` still produces a rule that can be referred from `array`, but avoids capturing the value of `array`, which is not initialized at that point in time. Even within a lambda function, Java has a syntactic restriction on forward reference, and we thus need to prefix `array` and `object` with the `this` qualifier.

The other method that is slightly different is `sep`. For instance in the `array` rule, `value.sep(0, word(","))` means "a sequence of zero or more values, separated by commas".

3.1.4 Launching the Parse

The `parse` method shows how one can initiate a parse over a string by using the `Autumn.parse` entry point with a default set of options (`ParseOptions.get()`). The method returns a `ParseResult`, which amongst other things indicates whether the parse was successful (`ParseResult#success`), and if so, whether it matched the whole input (`ParseResult#full_match`), or otherwise the furthest position to which the parse could progress before encountering an error (`ParseResult#error_position`).

3.2 `Parser` and `Parse`

Let us now look at the general principles of how parsers are implemented. This revolves around two key classes: `Parser` and `Parse`.

We mentioned the `Parser` class in the previous section. Instances of this class represent parsing expressions, or *parsers*, capable of recognizing some input.

More precisely, a parser is at core a function that, given the remaining input, succeeds or fails at matching a prefix of this remaining input.

Each kind of parser is a subclass of `Parser` which overrides its `boolean`

`Parser#doparse(Parse)` method. `doparse` is a protected method used to implement `boolean Parser#parse(Parse)`, which is the method that triggers the parser. The reason for the separation of both methods is that `parse` takes care of some bookkeeping automatically.

Together, these methods fulfill the function of parsing, and do so by reading and modifying a `Parse` object. As the name implies, this represents a "parse" over an input. A `Parse` contains, amongst other things, the input, the current position within the input, the position of the furthest error encountered so far and a stack used to build an AST (cf. Section 3.3).

In Autumn's current implementation, parses admit two different types of input, either a textual `String` or a list of objects.[5]

A parser checks if it matches the input by calling *sub-parsers*, or by direct comparison against characters or objects via the methods `Parse#char_at(index)` or `Parse#object_at(index)`.

`doparse` must return `true` if it matched some input, in which case it must set `Parse#pos` past the input that was matched. Otherwise, it must return `false` — `parse` will take care to reset `Parse#pos` to its initial value.

On top of `doparse`, `parse` adds automatic management of three pieces of data held in `Parse`:

- `Parse#pos` — by capturing its initial value and automatically reverting back to it if `doparse` returns `false`.
- `Parse#error` — the position of the furthest parser error encountered. This is used in error reporting and will be explained later. We also capture the stack trace for the furthest error.
- `Parse#log` — the data structure used to manage stateful context. This will be explained in detail in Chapter 5.

Autumn supplies one `Parser` implementation per parsing expression in

[5]It is conceivable to add support for different types of input in the future. Of particular interest are input streams: inputs that are initially incomplete, such as inputs delivered over a network connection. The big pitfall here is that since our algorithm backtracks, we will ultimately need to buffer the whole input anyway — therefore it is unclear whether the added complexity of supporting input streams is worth it in practice.

```java
import norswap.autumn.Parse;
import norswap.autumn.Parser;
import java.util.Arrays;
import java.util.Collections;
import java.util.List;

 public final class Sequence extends Parser
 {
    private final Parser[] children;

    public Sequence (Parser... children) {
        this.children = children;
    }

    @Override public boolean doparse (Parse parse) {
        for (Parser child: children)
            if (!child.parse(parse))
                return false;
        return true;
    }

    @Override public List<Parser> children() {
        return Collections.unmodifiableList(Arrays.asList(children));
    }

    // ... (elided)
}
```

Figure 3.2: Elided implementation of the `Sequence` class representing parsing expressions formed by the sequential combinator.

```java
import norswap.autumn.Parse;
import norswap.autumn.Parser;
import java.util.Arrays;
import java.util.Collections;
import java.util.List;

public final class Choice extends Parser
{
    private final Parser[] children;

    public Choice (Parser... children) {
        this.children = children;
    }

    @Override public boolean doparse (Parse parse) {
        for (Parser child: children)
            if (child.parse(parse))
                return true;
        return false;
    }

    @Override public List<Parser> children() {
        return Collections.unmodifiableList(Arrays.asList(children));
    }

    // ... (elided)
}
```

Figure 3.3: Elided implementation of the `Choice` class representing parsing expressions formed by the choice combinator.

Table 2.1, and then some additional ones besides.

As an example, Figure 3.2 and Figure 3.3 show a slightly elided version of the implementations of the `Sequence` and `Choice` classes, which implement two of the most common parsing combinators: sequencing and choice.

The code also includes an implementation for the `Parser#children()` method, which returns a list of the sub-parsers used by the parser. This method is used to enable traversal of the parser graph (cf. Section 6.5).

Notice how the approach used by our implementation is markedly different from that of Figure 2.6. That algorithm assumed that each parsing expression had been fully desugared using the logic of Table 2.2. In Autumn, by contrast, each type of parsing expression is given its own implementation. This makes it easier to define custom parsers, and makes debugging easier, as parser classes will appear in stack traces (cf. Section 6.6.1).

3.3 Building An Abstract Syntax Tree (AST)

In Section 3.1, we saw an example of Autumn grammar for the JSON language. However, this grammar simply specified a recognizer: it did not produce any kind of parse tree.

In this section, we revisit the example and show how to produce an Abstract Syntax Tree (AST) for the JSON language. The updated code follows.

```
1  import norswap.autumn.Autumn;
2  import norswap.autumn.DSL;
3  import norswap.autumn.ParseOptions;
4  import norswap.autumn.ParseResult;
5  import java.util.Arrays;
6  import java.util.stream.Collectors;
7
8  public final class JSON extends DSL
9  {
10     { ws = usual_whitespace; }
11
12     public rule integer = choice(
13         character('0'),
14         digit.at_least(1));
```

```
15
16      public rule fractional =
17          seq(character('.'), digit.at_least(1));
18
19      public rule exponent =
20          seq(set("eE"), set("+-").opt(), integer);
21
22      public rule number =
23          seq(character('-').opt(), integer,
24              fractional.opt(), exponent.opt())
25          .push(with_string((p,xs,str) -> Double.parseDouble(str)))
26          .word();
27
28      public rule string_char = choice(
29          seq(
30              set('"', '\\').not(),
31              range('\u0000', '\u001F').not(),
32              any),
33          seq(character('\\'), set("\\/bfnrt")),
34          seq(str("\\u"), hex_digit, hex_digit, hex_digit, hex_digit));
35
36      public rule string =
37          seq(character('"'), string_char.at_least(0), character('"'))
38          .push(with_string(
39              (p,xs,str) -> str.substring(1, str.length() - 1)))
40          .word();
41
42      public rule value = lazy(() -> choice(
43          string,
44          number,
45          this.object,
46          this.array,
47          word("true")  .as_val(true),
48          word("false") .as_val(false),
49          word("null")  .as_val(null)));
50
51      public rule pair =
52          seq(string, ":", value)
53          .push(xs -> xs);
54
55      public rule object =
56          seq("{", pair.sep(0, ","), "}")
57          .push(xs -> Arrays.stream((Object[][]) xs)
58                      .collect(Collectors.toMap(
59                                  x -> (String) x[0],
60                                  x -> x[1])));
61
62      public rule array =
```

```
63            seq("[", value.sep(0, ","), "]")
64            .collect()
65            .as_list(Object.class);
66
67        public rule root = seq(ws, value);
68
69        { make_rule_names(); }
70
71        public ParseResult parse (String input) {
72            return Autumn.parse(root, input, ParseOptions.get());
73        }
74    }
```

3.3.1 AST-Building Combinators

There are relatively few changes compared to the original version from page page 64, namely the additions of calls to methods `push(...)`, `as_val(...)` and `collect().as_list(...)`.

The basic principle behind AST creation in Autumn is that there exists a value stack (accessible via `Parse#stack`) on which parsers can push or pop items (often AST nodes).

Typically, a parser will pop the nodes pushed on the stack by its children, aggregate them into a bigger node, and push that on the stack.

But in fact, it is not the parser itself that will do this, but a new parser — of class `Collect` — created (in this example) by the AST-constructing combinators `push(...)`, `as_val(...)` and `collect().as_list(...)`. The `Collect` parser wraps the parser on which the combinator was called and takes care of the AST construction functionality.

In our JSON example, a simple case is that of the `as_val` combinators. The parsers returned by those, when their sub-parser is successful, simply push the parameter of the method on the value stack.

The `push` combinator, in its simple form (without `with_string`) takes a function of one parameter. This parameter, which we denote `xs` (for "the Xs"), designates an array of items pushed on the stack by the children of the parser, and popped by the parser. In the `pair` rule we simply push this array itself (the array object, not all its individual items) back on the stack! It will contain as first item a string (the key) and as second item a JSON value (the value mapped to the key).

`with_string` takes a function of three parameters. `xs` is as discussed previously, `p` is an instance of `Parse` (always unused in this grammar) and `str` is the string matched by the parser. Essentially `with_string` indicates we want to do something using the matched string — and so use a 3-parameters lambda instead of the single-parameter lambda that `push` normally accepts. Implementation-wise, `with_string` repackages the 3-parameters function into the interface expected by `push`.

In the `number` rule, we parse the number represented by `str` and push it on the stack. In the `string` rule, we cut off the double quotes and push the resulting string onto the stack.[6]

In the `object` rule, we do something a bit more technical. `xs` is still the array of items pushed on stack by sub-parsers, which in this case means that each item is an array pushed by the `pair` rule. Therefore we can cast `xs` to type `Object[][]` and stream it. We use `Collectors.toMap` from the standard library to isolate the key and the value from each sub-array.

Finally, in the `array` rule, `collect().as_list(...)` collects all items pushed on the stack by sub-parsers into a list whose parameter type is given by the class parameter (here it is `Object`), and pushes that list on the stack.

The `collect()` part actually creates a builder object than can be used to build `Collect` instances with some advanced options. The only AST combinators defined in class `rule` are those that cannot be customized (like `as_val(...)`) or shorthands for frequently used combinators (currently only for `push(...)`, which is equivalent to `collect().push(...)`). This avoids cluttering `rule` with options and validation logic that is only relevant to AST construction.

You can learn more about the various options for `Collect` in Autumn's documentation. We will however discuss one particular customization option that is often used in practice.

This option is called `lookback` and enables augmenting the `xs` parameter with items from the stack that were not pushed there by children of the

[6]Beware that this truncated string is not really the represented string. It may still contain character escapes (*e.g.*, '\n') that have not been processed. We elided the "unescaping" logic here, but the recommended solution is to isolate it in a helper method and call this method on the truncated string.

`Collect` parser. This is useful for rules that always act as a suffix to other rules that do push items on the stack. For instance, `parser.collect().lookback(3).push(xs -> ...)` means: collect all items pushed by `parser` on the stack, as well as three more items, pop them off the stack and pass them to `push` as the `xs` parameter.

3.3.2 Value Stack as Context

The value stack on which we push AST nodes is a form of state. Whenever we backtrack over our parsers, we must make sure they do not leave nodes on the value stack that are no longer part of the current interpretation of the input.

As it happens, this is exactly the same kind of logic we have to apply to any kind of *stateful context* in the presence of backtracking.

Therefore, the value stack will be handled by the same logic as other cases of stateful context — it is merely one case of particular interest.[7]

The logic behind the handling of stateful context is explained in detail in Chapter 5.

3.4 Beyond Basics

Many advanced features of Autumn, built on top of the basic principles presented in chapter, will be presented later down the line as they become relevant. This section maps out the locations where these features (in particular, their Autumn realization) are introduced.

- Section 4.5.2 introduces the `left_recursive` combinator which supports transparent left-recursion handling.
- Section 4.8.3 introduces the `left_fold` and `right_fold` combinators that enable *folding* a `push` parser over a list of terms.
- Section 4.9 introduces the `left_expression` and `right_expression` combinators, which enable defining families of (infix, prefix, postfix) expressions grouped by precedence level and is our recommended

[7] Unlike many other types of context however, we typically do not make parsing decision (*i.e.*, decisions about the input to match) based on the value stack. However this is still possible — and legal — in Autumn, but typically not needed and none of the built-in parsers do so.

way to define expression syntax.

- Section 5.6 introduces the `Log` data structure used to manage context, and how it can be used for context-sensitive parsing in practice.

- Section 5.6.3 introduces the `ParseState` class, which is a key way to make state use (including context) safer by isolating it for each parse.

- Section 6.1.4 presents Autumn's support for memoization through the `memo` combinator.

- Section 6.3 presents Autumn's support lexical analysis emulation via the `token` and `token_choice` combinators.

- Section 6.4.5 introduces the `Bounded` parser, which can be used restrict the input that a parser can match to that match by another parser.

- Section 6.5 presents Autumn's support for grammar traversal through implementations of `ParserWalker`, and for defining operations specialized per-parser-kind through implementations of `ParserVisitor`.

- Section 6.5.5 explains how visitors and walkers built into Autumn can be used to check for grammar well-formedness, while Section 6.5.6 explains how the `CopyVisitor` visitor/walker can be extended to perform grammar transformations.

- Finally, Section 6.6 introduces Autumn's support for debugging via reporting of the stack of parser invocation, and for performance tracing by recording a set of performance metrics.

3.5 Java 8 Grammar

In order to show a more substantial example, we have reproduced our full Autumn grammar for Java 8 in Appendix A. The grammar does not use every Autumn feature, but shows what a real grammar defining non-trivial syntax might look like.

Chapter 4

Infix Expression Parsing in PEGs and Combinators

As outlined in Chapter 1, the goal of the thesis is the creation of a new parsing system that strike a delicate balance between simplicity and expressivity.

The main avenue through which we propose to reach this goal is by letting the user extend the grammar formalism in well-circumscribed ways. In particular, we want the user to be able to define new parsers, which should conform to a standardized interface (cf. Section 2.3 and Section 3.2).

On the one hand we're basing ourselves on the semantics of a well-understood and effective parsing system (PEG). On the other hand we enable extensibility in a way that feels naturally integrated in the system. Each parser may have its own semantics, but from the perspective of the system it acts as a black box.

We think these extension possibilities have been under-exploited, and this thesis shows that we can use them to build new features. In particular, we are interested in features that improve the expressivity of the parsing system, or that solve pain points in the combinator approach.

Since they are typically implemented fairly similarly, both PEG and combinator parsing frameworks share the same pitfalls. Most of these pitfalls revolve around defining the syntax of expressions, and the perennial

issues of associativity and precedence. In this chapter, we take the first step towards transcending the PEG and combinator approaches by seeing how we can use the strength of the paradigm to solve its own pitfalls. Hence, this chapter is both a new contribution on infix expression parsing within the PEG/combinator paradigm, and a validation of the power of our *procedural parsing* approach in general.

In what follows, we will refer to *"PEG and combinator parsing frameworks"* as just *"PEG"*. Since the definition of custom parsing expressions / parsers is at the heart of our approach, let it be is clear that this refers to *PEG-style semantics* — *i.e.*, the *single parse rule* and *exclusively vertical backtracking*, as explained in Section 2.4; and not to a myopic vision of PEG as strictly the set of operators originally described by Ford [27].

In particular, this chapter discusses left-recursion handling in PEG, since the most notorious problem of these systems is their inability to handle left-recursive rules. The PEG formalism even explicitly excludes left-recursion. The reason is simple: in top-down recursive-descent parsers (which PEG formalizes), left-recursion causes infinite looping, as shown in Figure 4.1.

```
1 // S ::= S a | a
2
3 function parseS (input, pos)
4     pos2 = parseS(input, pos) // infinite loop
5     if (pos2 >= 0)
6         pos = pos2
7     if (input[pos] == 'a')
8         return pos + 1
9     return -1
```

Figure 4.1: Naive implementation of the left-recursive PEG grammar shown in the top comment. The function loops infinitely upon invocation.

4.1 Outline

One may question why we even need left-recursion in the first place. Truthfully, one may usually dispense with it, except in one common case: defining the syntax of left-associative infix expressions.[1] We detail this

[1]In CFGs, left recursion is also commonly used to encode repetitions. In PEG, this is easily replaced by uses of the repetition operators $*$ and +.

case in Section 4.2.

There are multiple ways to tackle the problem of left-recursion. The approach that may seem simplest at first glance is to define the semantics of left-recursion in PEG and to handle all left-recursion transparently.[2] But for this, the semantics of left-recursion in PEG must first be clarified (Section 4.3). As we also wish to be able to select the associativity of expressions that are both left- and right-recursive, we also need to pay attention to the particularly thorny issue of defining the semantics of left-associativity (Section 4.4). With this understanding in place, we give an algorithm that implements the established semantics (Section 4.5) assuming that the left-recursive expressions have been identified. We further explain how to automatically identify these left-recursive occurrences (Section 4.6). A related approach builds upon our algorithm to specify both associativity and precedence explicitly, yielding a new combinator named *expression cluster* (Section 4.7).

We then depart from the idea of transparency (*i.e.*, having explicit left-recursive references in the grammar), and show it is still possible to define syntactic constructs equivalent to the left-recursive ones, while retaining the ability to generate left-associative trees. To do so, we propose a parsing combinator that is analogous to a *left-folding* higher-order function (Section 4.8). Finally, we show that the left-folding approach still suffers from a couple minor practical issues — especially when multiple operators must be defined at the same precedence level, and propose new combinators that enable defining a family of expression at the same precedence level (Section 4.9). We conclude the chapter with a discussion contrasting the different approaches (Section 4.10).

Beyond the various solutions to the problem of left-recursion and left-associativity in PEGs, this chapter makes various other contributions. We precisely circumscribe the issues related to the problem of left-recursion and left-associativity — in particular under the constraint of maintaining the *procedural* character of the system (in which new combinators can be added seamlessly, and the execution model is simple). We discuss what it actually means to be left-recursive and left-associative — a discussion that is too often brushed aside. We propose a pragmatic understanding of left-associativity in PEG that helps retain its procedural

[2]We stress that *transparently* does not mean *automatically* — it might be necessary to explicitly annotate left-recursive occurrences. We do discuss *automatic* handling in Section 4.6.

character (Section 4.4). Finally, we validate the power of our procedural approach by showing it is sufficient to implement multiple solutions to these particularly thorny issues.

Part of the content of this chapter covers our 2015 paper *"Parsing Expression Grammars Made Practical"* [57] (concentrated in Section 4.2, Section 4.6 and Section 4.7), while the rest is original.

4.2 Grammatical Encodings of Expression Syntax

The most common and important use of left-recursion by far is in the definition of infix expression syntax. The way such syntax is often conveyed to humans is through the use of the *precedence* and *associativity* of operators. *Precedence* defines how tightly an operator "binds" its operands compared to other operators, while *associativity* defines if operators with the same precedence group to the left or to the right. See Section 2.5 for more details, examples, and a discussion of corner cases.

CFG and PEG, however, do not include the notion of precedence and associativity. Some tools do expose the notion, typically either to desugar it to the regular grammar notation, or to guide a lower-level process. For instance, the YACC (LALR) parser generator [43] uses associativity and precedence annotations to resolve shift-reduce conflicts.

Figure 4.2 shows five PEGs that define an arithmetic language with multiplication, division, addition and subtraction. Grammars 4.2a and 4.2b are the only two supported by the original PEG formalism. Grammar 4.2c requires left-recursive semantics. Grammar 4.2d uses special annotations to indicate precedence and associativity, changing the semantics of the grammar accordingly (we present one realization of this idea — *expression clusters* — in Section 4.7). Grammar 4.2e only uses associativity annotations, relying on the traditional *layering* approach to specify precedence. We could call these five grammars different *encodings* of the expression syntax.

While all grammars in Figure 4.2 describe the same language, they do not generate the same syntax tree. They all encode the correct precedence, but while grammars 4.2c, 4.2d and 4.2e are left-associative, grammar 4.2a is right-associative, and grammar 4.2b is neither (an expression's parse tree is composed of its leftmost operand followed by an optional list of suffixes).

$S \to P$ '+' S | P '−' S | P
$P \to N$ '∗' P | N '/' P | N
$N \to [0-9]+$

(a) Layered, Right-Associative

$S \to P($ '+' $S)*$ | $P($ '−' $S)*$ | S
$P \to N($ '∗' $P)*$ | $N($ '/' $P)*$ | S
$N \to [0-9]+$

(b) Idiomatic

$S \to S$ '+' P | S '−' P | P
$P \to P$ '∗' N | P '/' N | N
$N \to [0-9]+$

(c) Layered, Left-Associative

$E \to$
  E '+' E `@+ @left`,
  E '−' E,
  E '∗' E `@+ @left`,
  E '/' E,
  [0-9]+ `@+`

(d) Annotations

$S \to S$ '+' S | S '−' S | P `@left`
$P \to P$ '∗' P | P '/' P | N `@left`
$N \to [0-9]+$

(e) Layered + Associativity Annotations

Figure 4.2: Five PEGs defining a minimal arithmetic language.

Worryingly, arithmetic is traditionally defined as left-associative, but only the layered right-associative (4.2a) and idiomatic (4.2b) grammars can be written in standard PEG. This is the single best argument that PEG needs a way to handle left-recursion or at least to define left-associative syntax.

Needless to say, infix operators are a big deal in programming languages. Besides being overwhelmingly used to define the syntax of arithmetic expressions, they are also commonly used within the syntax of types (*e.g.*, type unions, disjunction, type parameters, type bounds) and in specialized languages (*e.g.*, regular expressions, PEG — cf. Table 2.1). Many general programming languages (*e.g.*, C++, Scala, Haskell) also offer the possibility to define custom infix operators, which are often used in the definition of domain specific languages (DSLs).

4.3 The Semantics of Left-Recursion

In order to realize transparent[3] left-recursion, we have to extend the PEG formalism with a semantics for it, then to find a means to implement it. Some of the simplicity of PEG is lost in the process, but — crucially — the simple parser interface described in Section 2.3 can be preserved.

The semantics of left-recursion are awkward to describe, because they break the *single-parse rule*: left-recursive rules can have multiple (nested) matches at the same input position. The function-call metaphor also breaks down as the naive implementation begets infinite recursion, as previously shown in Figure 4.1.

It is perhaps easiest to describe the desired behaviour using a crude operational semantics — by describing the left-recursion matching algorithm. The original algorithm was devised by Warth et. al for their OMeta parsing framework [105] as a modification of the packrat memoization mechanism (cf. Section 2.4.5), but the basic idea behind the solution remained similar in further improvements of the algorithm.

It goes as follows: when we first try to match a left-recursive parser, we block all left-recursive invocations of that parser. The result is a first match called *the seed*. The next step is to iteratively grow the seed.

[3] By *transparent* handling of left-recursion we mean a way to handle left-recursion that does not entail a fundamental reorganization or rewrite of the grammar. At most, left-recursive rules should be marked explicitly.

This is done by resetting the input position and attempting to match the parser again — except this time left-recursive invocations match the same input prefix as the previous seed. This process is repeated until the seed stops growing, at which point it becomes the final match for the parser.

For instance, consider the grammar $A \to Aa \mid b$ with the input "baa". Here is what happens over the run of the algorithm:[4]

1. The first alternative is tried. Left-recursion is blocked, so it fails. The second alternative succeeds. The new seed is "b".

2. The first alternative is tried. Left-recursion matches the seed (the result of the previous iteration — "b"). The new seed is "ba".

3. The first alternative is tried. Left-recursion matches the seed ("ba"). The new seed is "baa".

4. The first alternative is tried. Left-recursion matches the seed ("baa"). However, no further "a" can be matched. The second alternative fails as well. We cannot grow the seed, so the previous seed ("baa") is the final match.

So far, so good, but things get more complicated if we introduce syntax trees. In the discussion to follow, we will consider concrete syntax trees: where each parser creates a node in the syntax tree. However, the discussion is equally valid for more abstract syntax trees.

Given syntax trees then, one issue remains. What to do for rules that are both left- and right-recursive[5], such as $S \to S + S \mid x$. If we use parens to delimitate the syntax tree, the input string $x + x + x$ could be parsed as $((x + x) + x)$ or as $(x + (x + x))$. The first interpretation is said to be left-associative, the second right-associative. Which one should be selected?

The algorithm presented above will always favor a right-associative parse: after obtaining the initial seed, the right-recursion consumes the rest of

[4]If you are interested in a pseudo-code formulation of the algorithm, refer to Figure 4.3, which displays an algorithm that subsumes this simple left-recursion matching algorithm, adding support for associativity selection.

[5]We shall sometimes designate these occurrences as *dual-recursive*.

the matchable input, and the seed is not allowed to grow any further.

The question lead to some debate. Laurence Tratt [96] argues that only the left-associative option makes sense, keeping in with PEG's greedy nature. However, OMeta [105], which uses the algorithm above, always selects the right-associative interpretation.

Our own opinion is that this is just a matter of convention, and it is desirable to be able to choose the desired associativity. This, in turn, means we need to define specific semantics for the left-associative case (as the above algorithm naturally behaves right-associatively).

But we will also admit that we do not know any case that *requires* dual left- and right-recursion. It seems one can always get the desired associativity via layering (cf. Figure 4.2). Yet, there is still value in permitting associativity selection for dual recursion. Dual recursion might occur unforeseen, when performing grammar composition or grammar refactorings. Given that, it might help to preemptively clarify the desired semantics (left- or right-associative). This also helps when converting legacy grammars to Autumn (potentially with the assistance of tools): dual recursion elimination becomes one less difficulty to care about.

There is only one issue. Defining the semantics of left-associativity is, it turns outs, much more difficult than one would expect.

4.4 The Semantics of Left-Associativity

The platonic ideal of associativity is that it is a tie-breaker when there is an ambiguity in the way to match a given input: associativity defines whether we recurse on the right or on the left.

A few remarks. First, associativity does not cover all possible interpretations of an ambiguous input. Consider the CFG (not PEG!): $S \to S{+}S \mid x$ with input $x{+}x{+}x{+}x$. There are 5 possible interpretations: $(x+(x+(x+x)))$ (right-associative), $(((x+x)+x)+x)$ (left-associative), $((x+x)+(x+x))$, $(x+((x+x)+x))$, and $(x+(x+(x+x))+x)$. The last three interpretations can not be designated via associativity. But who in their right-mind would want such a behaviour?

Second, PEG is naturally unambiguous. This is why we need a separate left-associative semantics in the first place. It also means the tie-breaking interpretation is not very tractable. We need to proactively (before the

parse) find out where such “ambiguities” might occur.

4.4.1 Naive Formulation

There is one formulation of left-associative semantics that is very simple and works well on infix rules like $S \to S + S \mid x$. The formulation is **“a left-associative rule may not recurse within a right-recursion”**. For instance, within the aforementioned rule, the second S on the right-hand side can only ever match x.

This formulation has two problems, however.

First, it turns out that it is impossible to tell right-recursion apart from other forms of non-left recursion (also known as *middle recursion*). For instance, in $S \to S(S)S \mid x$, the middle S should be allowed to recurse freely.

Second, it means we also forbid legitimate right-recursion. For instance, in $A \to AA \mid bA \mid a$, the bA alternative is sometimes unambiguous and should be allowed to right-recurse — but as formulated, it can only ever match ba but not, for instance, bba.

4.4.2 Ambiguous Recursion

It turns out that our two problems from the previous sub-section share the same root cause.

Namely, both left- and right-recursion can be *hidden*: only occur on certain inputs. For instance, $S \to a?\ S\ c \mid b$ has hidden left-recursion: it is left-recursive on input bbc, but not on inputs abc or $aabcc$. Hidden recursion also occurs through ordered choice: $S \to aSc \mid bS \mid b$ is not right-recursive on input abc but is on input bb.[6]

The consequence is that we cannot *in general* tell whether a recursive occurrence is an instance of left- or right-recursion.

Of course, some rules are always unambiguous, for instance in $S \to S\ (S)\ S \mid x$, the middle S is never a left- or right-recursion.

[6]Left- and right-recursion can also be *indirect*, *i.e.*, reached through a chain of rules (parsers) rather than appearing immediately on the right-hand side. This complicates analysis slightly, but is not a limiting factor.

We could define the semantics of left-associativity by specifying the desired behaviour for the cases where recursive occurrences are unambiguous (*i.e.*, provably (non-)left- or (non-)right-recursive). We think this is a user experience mistake, however. Indeed, instead of having a simple rule that is always applicable, we now have to split the semantics in three parts: the semantics given ambiguous recursion, the semantics given unambiguous recursion, and the rules that statically determine whether a recursive occurrence is ambiguous or not. This turns an already tricky notion into something truly complex.

An alternative could be to check for ambiguity at run-time. This is relatively simple for left-recursion, but arduous for right-recursion — requiring some form of implicit lookahead parsing. We discarded this possibility out of hand, because it reintroduces a limited form of lateral backtracking to the PEG-like formalism (cf. Section 2.4.4). This significantly complexifies the semantics and opens up a lot of possibilities to run into patterns of poor performance due to the additional backtracking.

4.4.3 Restating the Problem

At this point, we hope to have established that the problem does not have any obvious elegant solution — the left-associative interpretation of a PEG rule that is both left- and right-recursive is intractable in general.

This seems like a strong claim, so let us re-contextualize and recapitulate the issue.

We take as a given that we want to work with an execution-based (operational) PEG semantics, without lateral backtracking. These semantics seem to be a big reason why PEG has been successful in practice: PEG parsing frameworks are easy to implement, and the parallel between their semantics and a function call stack makes them friendly to preexisting programmer intuition. Moreover, we build heavily on this property, most notably the context-sensitive machinery in Chapter 5.

Under this constraint, it is hard to conceive how to tackle left-recursion without relying on iterative expansion (“growing the seed”) — or said otherwise, without temporarily switching from top-down parsing to a

limited form bottom-up mode.[7,8] Given the iterative semantics, the presence of hidden left- and right-recursion means that recursion is ambiguous in general, and so we cannot determine which instances of right-recursion should be blocked, and which should be allowed to recurse freely.

4.4.4 A Pragmatic Way Out

So we seem at an impasse. Assuming we are bent on given left-associativity a semantics, what do we do in practice?

We compromise, and then, we cheat.

We decided to keep the simplistic interpretation we gave at the start of this section: "a left-associative rule may not recurse within a right-recursion". Let's recall that this works very nicely with infix binary expression that are both left- and right-recursive (*e.g.*, $S \to S + S \mid x$), which was the main reason why we wanted explicit associativity selection in the first place.

But we can do better than that by allowing the user to manually re-introduce recursion within a right-recursive invocation, when he has himself determined that it is the desired semantics. We do so via an "escape hatch" operator. In Autumn, the associated parser bears the name `GuardedRecursion`.

Let's denote the application of this operator on parser X as $[X]$, then we can solve our previous problematic examples by writing $S \to S\ ([S])\ S \mid x$ and $S \to AA \mid b\ [A] \mid a$.

The result is both simple and flexible. By leaving a little bit of the interpretation to the user (but only for the less common case of letting right-recursion run in parts of left-associative rules) we avoid foisting

[7] This is the weakest part of the argument. But it must be said that no alternative to iterative expansion has yet been found despite significant scrutiny. Since we work under the constraint of keeping the semantics straightforward, it seems to be a fair working assumption that a left-recursion handling scheme with a simple operational semantics is not readily attainable.

[8] A counter-argument here might be that we could "just" expand the bottom-up semantics to resolve the ambiguous recursions. While this angle might yield interesting results, it means the whole semantics must be duplicated, once again running counter to our constraint.

upon him the much more significant overhead of a complex semantics, while retaining the same expressivity.

4.4.5 Related Work

In what is the most relevant paper to our discussion of the semantics of left-associativity [64], Medeiros *et al.* propose an axiomatic semantics of PEG, where they handle associativity by assigning precedence levels to nonterminals. A given nonterminal can recurse **at the same position** via nonterminal occurrences that have a similar or higher precedence level, but not a lower one. They give the example of the rule $E \to E_1 + E_2 \mid n$ for left-associative addition. Because E_2 cannot recurse through E_1, it will only ever match n.

The remaining questions are how to assign these precedence levels, and what semantics does the assignment produce.

Unfortunately, Medeiros *et al.* only present a manual version and seem to suggest that this assignment is a simply tooling issue [64]. They say — after speaking of making left-associativity the default:

> The disadvantage is that this makes specifying precedence levels with right-associative operators a little harder, but this is a user interface issue: a tool could use a YACC-style interface, with `%left` and `%right` precedence directives and the order these directives appear giving the relative precedence, with the tool assigning precedence levels automatically.

As we have already established, it is not as simple as that: given the existence of hidden left- and right-recursion, it is impossible to pinpoint actual left- and right-recursion in advance of knowing the actual input.

Second, we argue that the semantics can be confusing.

Consider the (admittedly pathological) grammar $A \to A_1 A_2 \mid aA_2 \mid a$. Given the semantics proposed by Medeiros *et al.*, this will always produce right-associative matches: *e.g.*, $(a, (a, a))$ for input aaa. In fact, there is not a single way to reassign the precedence levels in this grammar that would make it behave left-associatively. While it is true that no one in their right mind would write such a rule — if left-associative behaviour is desired, the second alternative is useless! — but rules with a similar structure might occur when composing rules that make sense locally.

We have experimented with Medeiros *et al.*'s semantics, and we believe it would be possible to alter it such that there always exists a precedence level assignment that produces the same associativity as the semantics we ourselves propose. No single simple change will suffice to do so, however. Most notably, it is not enough to either prohibit recursion into nonterminals of the same precedence level, or to preserve the precedence level for each nonterminal being matched, irrespective of input position.

From a user experience perspective, we think our approach foists less complexity on the user, and makes the decisions more intuitive by making the choices the user has to make closer to the object-level, namely (a) is this rule left- or right-associative, and (b) are there instances of non-left recursion that should be allowed to recurse?

4.5 Transparent Left-Recursion Handling in PEG

Having established the semantics of left recursion with associativity selection in the two previous sections (4.3 and 4.4), we are now ready to present the algorithm that implements these semantics (Section 4.5.1), discuss how it can be used in practice (Section 4.5.2) and talk about some inherent performance issues, and what can be done about them (Section 4.5.3).

Let us first briefly summarize the semantics we established in the previous sections. We proceed by iterative expansion ("growing the seed"). Upon first invocation of a left-recursive rule, left-recursion is disabled, then we keep re-invoking the rule at the same input position, using the match from the previous iteration as result for left-recursive invocations, until the match stops growing. By default, this will yield right-associative syntax trees. For rules explicitly marked as left-associative, all instances of non-potentially-left recursion are blocked from recursing more than once. This can be circumvented by the use of an escape hatch operator.

4.5.1 The Left-Recursive Algorithm

Figure 4.3 shows our left-recursion handling algorithm, including provisions for left-associative rules. The algorithm applies to a *left-recursive parser*, which is a type of parser that wraps a parser that is *logically*

```
1  seeds = {}
2  recursions = {}
3  parse expr (a left-recursive parser) at position:
4      if seeds[position] [expr] exists then
5          return seeds[position][expr]
6      else if recursions[position] [expr] == 2 then
7          return failure
8      else if recursions[position] [expr] == 1 then
9          recursions[position][expr] = 2
10         result = parse(expr.operand)
11         recursions[position][expr] = 1
12         return result
13     current = failure
14     seeds[position][expr] = failure
15     if expr is left-associative then
16         recursions[position][expr] = 1
17     repeat
18         result = parse(expr.operand)
19         if result consumed more input than current then
20             current = result
21             seeds[position][expr] = result
22         else
23             remove seeds[position][expr]
24             if expr is left-associative then
25                 remove recursions[position][expr]
26             return current
```

Figure 4.3: Left-recursion and associativity handling algorithm.

left-recursive. This wrapper can be inserted manually or automatically,[9] as will be discussed in Section 4.6. We will refer to the wrapped parser as *the operand* of the wrapper.

The algorithm maintains two global data structures. First, a map (`seeds`) from *(position, parser)* pairs to parse results. Second, a map (`recursions`) from the same kind of pairs to an integer that designates the current *recursion level*, used to avoid right-recursion in left-associative parses. A parse result represents any data generated by invoking a parser at an input position, including the syntax tree constructed and the amount of input consumed. The parse results stored in our map are our *seeds* — temporary results that can "grow" in a bottom-up fashion. Note that our data structures are global, so that they persist between (recursive) invocations of the algorithm. The parsing algorithm for other parsers need not be concerned with them.

Let us first ignore left-associative parsers (*i.e.*, uses of the `recursions` data structure). When invoking a parser at a given position, the algorithm starts by looking for a seed matching the pair, returning it if present. If not, it immediately adds a special seed that signals failure. We then invoke the operand, update the seed, and repeat until the seed stops growing. The idea is simple: on the first go, all left-recursion is blocked by the failure seed, and the result is our base case. Each subsequent invocation allows one additional left-recursion, until we have matched all the input that could be. For rules that are both left- and right-recursive, the first left-recursion will cause the right-recursion to kick in. Because of PEG's greedy nature, the right-recursion consumes the rest of the input that can be matched, leaving nothing for further left-recursions. The result is a right-associative parse.

Things are only slightly different in the left-associative case. When starting the parse, we set the recursion level for the *(position, parser)* pair to 1. This will cause any non-left recursion to bump the counter to 2, after which any deeper recursion will be blocked. The purpose of this counter is simply to let a left-associative rule right-recurse only once — as opposed to blocking right-recursion entirely, which would not work for most rules that are both left- and right-associative such as

[9] Though Autumn does not support automatic insertions, for reasons that are explained in Section 4.6.

$S \to S + S \mid a$![10] Given that right-recursion is now limited, the loop still grows the seed, ensuring a left-associative parse.

A few notes about the algorithm. First, if used in conjunction with packrat parsing, it requires memoization to be disabled while the left-recursive parser is being invoked. This is a limitation the algorithms shares with its predecessors: otherwise, we might memoize a temporary result. In theory, it is possible to memoize sub-parser invocation that are not involved in the left-recursive loop, but it is unclear if the benefits are worth the bookkeeping costs.

Second, and as explained in Section 4.4, middle-recursion is blocked along with real right-recursion. To enable it, Autumn does supply an escape hatch operator (`GuardedRecursion`) which allows the guarded parser to recurse freely. Conceptually, this is achieved by hiding the recursion count in the `recursions` map, for the duration of the guarded parser's invocation.

4.5.2 Transparent Left-Recursion in Autumn

Figure 4.4 shows how the algorithm may be used in Autumn, through the `left_recursive` combinator (which enacts the default behaviour: right-associative for dual-recursive rules) and the `left_recursive_left_assoc` combinator (left-associative for dual-recursive rules). The code fragment uses these combinators to define a simple arithmetic expression syntax with parentheses and a ternary (conditional) operator.[11]

Please do not use this code as a model, as it suffers from a performance flaw that we shall discuss in the next section.

The first rule defines a "primary expression" that could be either a parenthesized expression, or a number (rule `number`, not shown). Parentheses let us escape precedence, and so we want to refer to the root of the expression hierarchy (`expr`). Since this is a forward recursive reference (`expr` is defined after `primary_expr` and both rules are mutually recursive), we

[10] We actually made this mistake in the algorithm published in our previous paper [57]. The mistake stemmed from how the left-recursive parser was not originally wrapping the choice parser corresponding to the left-recursive rule, but left-recursive alternates of that parser.

[11] We're only dealing with syntax here, but if the semantics bother you can assume that the ternary operator evaluates to its third operand if its first operand evaluates to zero, and to its second operand otherwise.

```
rule primary_expr = choice(
    seq("(", lazy(() -> this.expr).guarded(), ")")
        .push(xs -> new ParenExpr($(xs,0))),
    number);

rule mult_expr = left_recursive_left_assoc(self ->
    choice(
      seq(self, "*", self).push(xs -> new MulExpr($(xs,0), $(xs,1))),
      seq(self, "/", self).push(xs -> new DivExpr($(xs,0), $(xs,1))),
      primary_expr));

rule add_expr = left_recursive_left_assoc(self ->
    choice(
      seq(self, "+", self).push(xs -> new AddExpr($(xs,0), $(xs,1))),
      seq(self, "-", self).push(xs -> new SubExpr($(xs,0), $(xs,1))),
      mult_expr));

rule expr = left_recursive(self ->
    choice(
      seq(self, "?", self, ":", self)
          .push(xs -> new TernaryExpr($(xs,0), $(xs,1), $(xs,2))),
      add_expr));
```

Figure 4.4: Using left-recursive parsing combinators in Autumn to define the syntax of arithmetic expressions with parentheses and a ternary (conditional) operator.

resort to the method described in Section 3.1.3 and wrap the reference in a `lazy` parser).

Right after that, the `guarded()` combinator is applied in order to create an instance of the `GuardedRecursion` parser, our escape hatch operator. This is required because `primary_expr` is called from left-associative rules (`mult_expr` and `add_expr`) which block non-left recursion as per our algorithm (cf. Figure 4.3). Without the escape hatch, it becomes impossible to parse the input $1 + (2 + 3)$.

Finally, the AST produced by the parenthesized expression is retrieved and wrapped in a `ParenExpr` node, which is pushed onto the value stack via the `push` combinator (refer to Section 3.3.1 for more details).

Moving on, `mult_expr` and `add_expr` are fairly similar rules that each express two left-associative infix operators at the same level of precedence.

As you can see, we explicitly mark the rules as left-recursive (and potentially left-associative). Section 4.6 will discuss how it is possible to automatically mark parsers as left-recursive, and why we rejected the approach — though we do automatically check for unhandled left-recursion by default.

You may also notice that these rules are recursive, yet they do not use `lazy`. For brevity's sake in case of direct recursion, Autumn automates the whole setup by passing this lazy parser to the combinator directly (as the `self` parameters in our listing).[12] Indirect recursion via `lazy` is still supported.

Rule `expr` defines an expression to be a ternary expression or an additive expression. It is left-recursive, but not left-associative, as the ternary operator is right-associative.

Finally, we note that we could actually avoid the dual recursion fairly easily: just replace the right `self` operands with the lower-precedence rule (*e.g.*, `primary_expr` within `mult_expr`). Then we can replace `left_recursive_left_assoc` with simply `left_recursive` (cf. the discussion at the end of Section 4.3).

[12] Autumn also has the `recursive` combinator to help tersely define recursive rules that do not require the left-recursive behaviour.

4.5.3 Performance Woes

So, we now have a solution that supports left-recursion with a fairly straightforward semantics that is well-defined for every possible case. For the pesky dual left- and right-recursive case, we support associativity selection. Is this the solution we have been looking for all along?

Alas, no. We can show that the code in Figure 4.4 exhibits undesirable performance overheads.

The issue occurs whenever a left-recursive parser matches its non-recursive part — in Figure 4.4, those are always the last alternative of the `choice` combinator. In those cases, the algorithm will always call the non-recursive part twice: first during the first iteration, as it will become the initial seed; and then during the second iteration, when we will find that the match did not grow. By itself, this seems benign, but the issue compounds when left-recursive parsers are stacked over one another, as is the case in our example, and is very often the case for infix operators. Whenever N left-recursive parsers are chained in this way, the first non-left-recursive parser in the chain will be called 2^N times when we just want to match what it matches. So in Figure 4.4, using the `expr` rule to match a number will cause the `number` rule to be invoked eight times! Real operator hierarchies run much deeper. For instance, Java and C have about 10 contiguous levels of precedence that are left-associative, adding up to a slowdown by a whopping factor of 1024 in those cases. The complexity is somewhat amortized for longer expressions, but the cost remains prohibitively high.

There are a couple ways in which this issue can be fixed. The first is to memoize the non-left-recursive part of each left-recursive parser. This is a tricky proposition, because we have to be sure that this part is not involved in a left-recursive loop. As we have already established in Section 4.4.2, this is impossible to know in general because of hidden left-recursion. So the responsibility will fall squarely on the shoulders of the user.

Another option is to exploit the memoization mechanism inherent to our left-recursion handling algorithm (*i.e.*, the seed storage) to avoid reparsing the initial seed. For instance, here is how we would rewrite the `add_expr` rule from Figure 4.4:

```
1 rule add_expr = left_recursive_left_assoc(self ->
2     choice(
3       seq(self, "+", self).push(xs -> new AddExpr($(xs,0), $(xs,1))),
4       seq(self, "-", self).push(xs -> new SubExpr($(xs,0), $(xs,1))),
5       self,
6       mult_expr));
```

The only change is the lone `self` as third choice alternative. Upon the first iteration of the algorithm, it will fail and a `mult_expr` will be matched. On the second iteration (assuming no operators are present), it will match and reuse the cached seed, hence completing the algorithm without causing a second invocation of `mult_expr`. Just like the memoization trick, this cannot be automated in general because of hidden left-recursion.

Clearly, we would be better served with a way to define left-associative parsers that is not as risky.[13] We will come back to this problem in Section 4.9, after investigating two other avenues that can teach us a lot about infix expression parsing (Sections 4.7 and 4.8).

But for now, we will investigate a problem closely related to transparent left-recursion handling, namely automatic left-recursion discovery.

4.6 Automatic Left-Recursion Discovery

The algorithm in Figure 4.3 is used for parsing left-recursive parsers. These parsers wrap a sub-parser that is logically left-recursive (*i.e.*, if left unhandled, it's invocation will lead to infinite recursion). However, we did not yet discuss how to identify logically left-recursive parsers. This is the object of this section.

We note that this possibility has been largely ignored in the literature. OMeta [105] and Medeiros *et al.*'s approach [64] detect left-recursion "as it occurs", which entails overheads that can be avoided by detecting left-recursion in advance. The Katahdin [83] language, which pioneers some of the techniques we perfected in our algorithm, requires left-recursive grammar rules to be annotated explicitly and otherwise blocks left-recursion by default — left-recursion will always fail in an unmarked rule.

[13] Not to mention, that would be more in line with our goal of *simplicity* as described in Section 1.3 — the easy way should be the *right* way.

If we define a relationship *FIRST(x, y)* such that y is a direct sub-parser of x [14] that can be invoked at the same position as x, then we can derive a directed graph from each grammar where the nodes are the parsers, and the edges are given by the *FIRST* relationship. In this graph, any loop designates an instance of left-recursion.

In practice, computing the *FIRST* relationship will require computing a predicate *NULLABLE(x)* that holds only if a parser x can succeed without matching any input.

Theoretically, each parser in the loop is left-recursive. In practice, it is better to "break up" the loop by designating a single parser as the left-recursive one, subject to the algorithm in Figure 4.3.

We do note that the algorithm does support more than one parser being marked as left-recursive, or even all of them. This does come with a pretty important caveat, however: the parsers will do unnecessary extra work in trying to grow all the seeds (one per left-recursive parser), sometimes consequently so.

To illustrate this phenomenon, look at Figure 4.5. The rule A matches one or more repetitions of the string "dcba". Let's assume all rules are marked as left-recursive and hence subject to our left-recursion handling algorithm. If we invoke A on the input string "dcba", it will invoke B twice: once to obtain the initial seed, and a second time to confirm the seed cannot grow. B, in turn, invokes C twice, for the same reasons. Similarly, C invokes D twice. In total, C is thus called four times, and D eight times. In general, if there are $L > 1$ left-recursive parsers in a left-recursive cycle, the parse will slow down by a constant factor of 2^{L-2} when compared to the case where only a single parser is marked as left-recursive. The ability to grow the seed (*e.g.*, if our input was "dcbadcba") does not impact this factor.

We do note that this slowdown does not cause the theoretical run time to become exponential in the input size, as the slowdown factor is a constant that depends on the grammar but not on the size of the input.[15] Nevertheless, for big left-recursive cycles, the slowdown can become

[14] A direct sub-parser of aparser p is any parser that p may invoke directly from its `doparse` method.

[15] One notable other cause for a slowdown of the same nature will be explored in Section 6.1.2.

$$
\begin{aligned}
A &\rightarrow B\ a \\
B &\rightarrow C\ b \\
C &\rightarrow D\ c \\
D &\rightarrow A\ d\ \mid\ d
\end{aligned}
$$

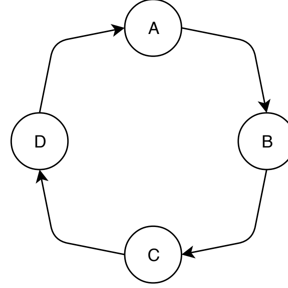


Figure 4.5: A grammar made out of a single left-recursive cycle (represented on the right). With starting symbol A, the grammar is equivalent to `(d c b a)+`, and is suceptible to slowdown using our left-recursive handling algorithm, if multiple rules are marked as left-recursive.

$$
\begin{aligned}
A &\rightarrow A\ c\ \mid\ B\ a \\
B &\rightarrow A\ b\ \mid\ B\ b\ \mid\ b
\end{aligned}
$$

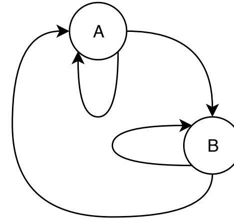


Figure 4.6: A grammar with three left-recursive cycles (represented on the right), two of which are nested in the third one. With starting symbol A, the grammar is equivalent to `(b+ a c*)+`.

consequent, hence the preference for breaking left-recursive cycles by marking a single parser as left-recursive.

Despite this caveat, we note that the ability of the algorithm to handle multiple left-recursive nodes within a single cycle is essential, because some parsers may be part of multiple left-recursive cycles, and it might not be possible to break those cycles using only nodes that are not shared amongst multiple cycles. The simplest example is that of embedded left-recursion: whenever all the parsers in a left-recursive cycle are also part of a bigger left-recursive cycle. This is illustrated in Figure 4.6.

Left-recursion detection boils down to cycle[16] detection in directed graphs.

[16]A cycle is a *trail* (a path) in the graph with no repeated vertices excepted that the first and the last vertices are the same. This is sometimes called a *simple cycle*, *simple circuit* or *elementary circuit*, depending on the authors.

This can be accomplished with Johnson's algorithm [42].[17] Johnson's algorithm has complexity $O((v + e) * (c + 1))$ where v is the number of vertices, e is the number of edges, and c is the number of cycles in the graph. In theory, the number of cycles can be exponential in the size of the graph. However, for grammars it is usually safe to assume the existence of a constant number of cycles.

Johnson's algorithm is relatively complex to implement — it is essentially a dynamic programming algorithm built on top of Tarjan's strongly connected components algorithm [90]. It is also relevant that while our *FIRST* relationship forms a de-facto graph, it is not programatically available in a form that offers all the niceties of a real graph data structure. It is of course possible to reify this graph, it just is additional work.

Fortunately, we do not actually need to find *all* cycles. It is sufficient that we are able to identify nodes to be marked as left-recursive, such that there is not any cycle left in the graph that does not contain a left-recursive node. Algorithmically, this problem is a whole lot simpler: it suffices to perform a depth-first traversal of the graph starting from each grammar rule. We cut off the visit when encountering a cycle (*i.e.*, a node already on the current trail, which will be marked as left-recursive); or when encountering a node that was already visited (*i.e.*, removed from the trail after the part of the graph reachable through it was entirely visited). That algorithm has complexity $O(v + e)$.

In general, identifying left-recursive nodes has two possible purposes. The first is to automatically insert left-recursive parsers into the grammar. The second is to warn the users that unhandled left-recursion has been found and that the grammar is consequently not well-formed [27].

Autumn currently does not allow the automatic insertion of left-recursive parsers — nor does it allow grammar rewriting in general (*i.e.*, *in-place* mutation of the parser graph). From our experience with previous prototypes, we did find out that grammar rewrites do make grammar debugging more arduous. They also hamper extensibility. Consider for instance what should be done with the *FIRST* and *NULLABLE* relationships in case of a grammar rewrite — should we compute them before or after the rewrite, or maybe both? In that case, is there a way to incrementalize their computation? Does any of this complexity need

[17] Not to be confused with the more popular algorithm from the same author whose purpose is to find all shortest paths between pairs of vertices in a directed graph.

to be pushed onto the user?

Autumn does however allow grammar *transformations* which create a modified copy of the original grammar. This will be presented in Section 6.5.6.

Regarding left-recursion, we will see later that we do not think transparent left-recursion handling is necessarily the best solution. Given that — using the detection algorithm described in this section — we can protect the users from the harmful effect of unmanaged left-recursion, we do not think that automatic insertion of left-recursive parsers is a necessary feature.

The implementation of the left-recursion detection algorithm poses some other interesting — but not strictly algorithmic — challenges. The first one is how to compute the *FIRST* and *NULLABLE* relationships when users can implement their own custom parsers using a general programming language. There are no magical solutions here: the relationships have to be defined for each type of parser, and if custom parsers are allowed, it falls to its author to define the relationships for the new parser. The second challenge is in how to walk and modify the grammar (*i.e.*, the parser graph). In Autumn, we tackle these challenges through an extensible *visitor pattern* [29] architecture which we describe in Section 6.5. This enables defining the *FIRST* and *NULLABLE* relationships separately from parsing semantics, and reusing the grammar traversal logic for multiple endeavours. Section 6.5.5 will present the built-in visitor used to compute the two relationships.

4.7 Expression Clusters

Expression clusters — which we initially proposed in our 2015 paper [57] — are a new combinator used to express the syntax of infix expressions. The idea is to express the syntax of all operators within a single combinator (hence *cluster*).

Expression clusters are somewhat analogous to choice operators, where additionally the order of the terms and explicit annotations are used to enforce the proper precedence and associativity semantics. The terms (*i.e.*, the sub-parsers) are grouped by precedence, with the annotation `@+` indicating an increase in precedence.

Figure Figure 4.7 repeats the grammar already shown in Figure 4.2d

$E \rightarrow$
E '+' E `@+ @left`,
E '−' E,
E '∗' E `@+ @left`,
E '/' E,
[0-9]+ `@+`

Figure 4.7: A grammar using annotations to specify precedence and associativity within a single rule, which can be implemented by the expression cluster combinator (this grammar also appears as Figure 4.2d).

(in Section 4.2 on infix expression encodings). In this grammar, the `+` and `-` operators have the same precedence, which is lower than that of the `*` and `/` operators. The `@left` or `@right` annotations indicate the associativity at a given precedence level. Notice how the cluster references itself recursively at all references levels, including in left- and right-recursive positions.

An algorithm for the expression cluster parser is given in Figure 4.8. This algorithm builds upon that of Figure 4.3. We now maintain a map from cluster expressions to their *current precedence*. We iterate over all the precedence groups in our cluster, in decreasing order of precedence (from most binding to least binding). For each group, we verify that the group's precedence is not lower than the current precedence. If not, the current precedence is updated to that of the group. We then iterate over the sub-parsers in the group, trying to grow our seed. After growing the seed, we retry all sub-parsers in the group *from the beginning*. Note that we can do away with our `recursions` map from Figure 4.8: left-associativity is handled via the precedence check. For left-associative groups, we increment the precedence by one, forbidding recursive entry in the group. Upon finishing the invocation, we remove the current precedence mapping only if the invocation was not recursive: if it was, another invocation is still making use of the precedence.

The biggest advantage of expression clusters is their "clean look": there is no need to encode precedence via the user of "layered" rules as in grammars 4.2a and 4.2c of Figure 4.2. The usual concepts of precedence and associativity are made explicit. Precedences are also scoped by expression cluster, meaning it is possible to define multiple independent expressions syntaxes without sharing a single precedence table (something

```
1  seeds = {}
2  precedences = {}
3  parse expr (a cluster parser) at position:
4      if  seeds[position] [expr]  exists then
5          return seeds[position][expr]
6      current = failure
7      seeds[position][expr] = failure
8      min_precedence = precedences[expr] if defined, else 0
9      loop: for group in expr.groups (in decreasing precedence order)
        do
10         if group.precedence < min_precedence then
11             break
12         precedences[expr] = group.precedence +
13             group.left_associative ? 1 : 0
14         for op in group.ops do
15             result = parse(op)
16             if result consumed more input than current then
17                 current = result
18                 seeds[position][expr] = result
19                 goto loop
20     remove seeds[position][expr]
21     if there is no other ongoing invocation of expr then
22         remove precedences[expr]
23     return current
```

Figure 4.8: Expression cluster parsing algorithm.

layered approaches also do, however).

On the other hand, the reality of grammars is sometimes more messy than a simple precedence scheme can handle. For instance, some operators may restrict the range of syntactically valid operands. It is not for instance possible (and intrinsically difficult) to skip precedence levels. Another example is supplied by Java, which in its 8th version added anonymous functions (*lambdas*) to the language. A lambda can form a whole expression, or appear on the right of assignment or conditional (ternary) expressions — but not on their left! This cannot easily be represented with expression clusters, who depend on iteratively growing the *left-hand* side seed![18]

In practice, it is also useful to label the different terms of a cluster with a name, if only for debugging and error-reporting purpose. Annotations for AST constructions may also need to be added. As expression clusters gets more spread out (and their DSL representation more and more complicated), the aesthetic argument starts making less and less sense, to the point where the layered construction is often easier to understand, because more explicit.

Like transparent left-recursion handling, the algorithm also does not play nice with memoization, and it also requires an escape hatch to parse non-left recursion within left-associative operator's operands.

For these reasons, we cut expression clusters out of Autumn, mostly at the profit of a new combinator used to define a collection of operators at a given precedence level, which will be introduced in Section 4.9.

We will however point out that expression clusters do not suffer from the performance pitfalls highlighted in Section 4.5.3. The reason is that since they encompass all the chained left-recursive operators, it essentially fuses their seed expansion logic. As such, the worst case becomes that a primary expression will be parsed twice, which is not so worrisome.[19]

---

[18] It should be said we think this particular case is a mistake in grammar design — it would have been better to make lambdas a *primary expression* (of lower precedence than all the operators). The grammar in the Java specification already allows illegal expressions such as `3 + String.class` or `1?2:3`. Just because a language constraint can be expressed syntactically does not always mean it should. Nevertheless, this is a nice example of the expresiveness limitations of expression clusters.

[19] As long as you do not stack multiple expression clusters, which seems like an unlikely degenerate case, but it is feasible.

Expression clusters also teach us something genuinely valuable about parsing infix expressions, namely that it is practical to parse multiple operators in a truly bottom-up fashion (*i.e.*, starting with the highest-precedence operator) using iterative expansion. We shall return to that idea in Section 4.9.

4.8 Left-Associativity via Left-Folding

So far we have presented two new combinators that enable left-recursion within a PEG-like framework. One rather general (transparent left-recursion handling, Section 4.5), and one rather specialized (expression clusters, Section 4.7).

Both share two woes: the need to use an escape hatch operator when specifying left-associativity, and incompatibility with memoizing combinators. They also have issues of their own: a tendency to cause performance issues for transparent handling, as well as a tendency to become too large, inflexible and bloated for expression clusters.

But handling left-recursion is not an aim in itself. What we really want is to be able to express the syntax of infix expressions — and crucially, to get a syntax tree with the proper associativity out of them.

Perhaps we could borrow a page from some parsing tools which let you specify operators using some "idiomatic" encoding (either explicitly, or by rewriting the grammar for you) — then let you get a syntax tree with the proper associativity by *rewriting* the original tree [31, 72]. For instance, using the PEG idiomatic encoding (see Figure 4.2b), you would write $E \to M(`+\text{'}\ M)*$ instead of $E \to E\ `+\text{'}\ M \mid M$.

Using such a solution, we could avoid the downsides of explicit left-recursion handling. And since Autumn is inherently extensible, we do not even need to resort to grammar or syntax tree rewrites — we can write a new combinator that will provide the equivalent behaviour. The rest of this section explores how we can do this.

4.8.1 Syntax Trees and Left Folds

As the notion of associativity is intimately related to that of syntax trees, talk about potential solutions necessitates a little discussion of how (abstract) syntax trees are usually constructed in Autumn. The basics of syntax tree construction are covered in Section 3.3. We briefly recall

some principles, which will be helpful in the construction of our new combinator.

Autumn uses a stack data structure to construct syntax trees, which we call *value stack*. Some parser combinators are able to push and pop nodes on the value stack. Special handling is necessary to undo changes to this stack in case of backtracking. This is a more powerful model than is usual, and will be further discussed in Chapter 5. Autumn lets users write these combinators explicitly, which allows them to directly generate an *abstract* syntax tree instead of a mere parse tree.[20]

In general, the behaviour of a parser regarding the parse tree can be modeled as a $N^* \to N^*$ function; where N is the set of syntax tree nodes. The input of the function is the ordered set of syntax tree nodes emitted by the direct sub-parser of the parser. The function returns the ordered set of parse tree node that the parser *emits*. Let's call this *the syntax tree reduction function* or *STR function* for short. In Autumn, the input nodes are typically popped from the value stack, while the output nodes are pushed on it.

The default behaviour of a STR function is to simply pass the nodes generated by the direct sub-parsers as is. The most common non-default behaviour is to use all of the nodes emitted by the sub-parsers to build a single higher-level node — for instance combining multiple statements into a single compound statement, or multiple field and method declaration into a single class declaration.

Under these conditions, it becomes possible to conceive of a STR function that produces a left-associative tree for a non left-recursive parser. Such techniques sometimes involve rewriting parts of an existing tree, in which case they're called *tree rewriting*. Their use is not uncommon [31, 72]. But with our proposed solution, rewriting is not actually necessary.

Any left-recursive rule can be split into (a) the union of its non-left-recursive parts (corresponding to the possible initial seeds in Section 4.5) — which we will call S, and (b) the union of the suffixes of its left-recursive parts (corresponding to possible "growths" of the seed) — which we will call R. Therefore, left-recursive rules can be expressed as $X \to S\ R \mid S$ and consequently rewritten as $X \to S\ R*$.

[20]See shaded box on page 9.

$$S \rightarrow S \text{ `}*\text{' } N \mid S \text{ `/' } N \mid N$$
$$N \rightarrow [0-9]+$$

$$S \rightarrow N \; (\text{`}*\text{' } N \mid \text{ `/' } N) *$$
$$N \rightarrow [0-9]+$$

Figure 4.9: Two PEG grammars for the language of integer multiplication and division. The first grammar is left-recursive, while the second is in the $S\ R*$ form discussed in this section.

Figure 4.9 shows two PEG grammars for a simple arithmetic language with multiplication and division: the natural left-recursive form, and the corresponding $S\ R*$ form.

If we assume that the S part yields one node, while the $R*$ yields a list with one node per successful invocation of R, then the input to our STR function is a list containing these two items.

To produce a left-associative abstract syntax tree from the $(S, R*)$ input, we propose using a *left-fold*. Figure 4.10 shows the definition for a left-fold function. The result is initialized to the `initial` parameter, then built up incrementally by calling function `f` once for each item in the list, using the previous result as first parameter. For instance, `fold_left(0, [1, 2, 3], f)` will yield the same result as `f(f(f(0, 1), 2), 3)`.

```
1 function fold_left (initial, list, f)
2     r = initial
3     for it in list
4         r = f(r, it)
5     return r
```

Figure 4.10: Pseudo-code implementation of a left-fold function.

We want to use the initial seed node (S) as `initial` parameter and the list of R nodes for `list`. The function `f` has to be supplied by the user.

4.8.2 Left-Folding by Hand

Using Autumn, we can define a new parser combinator that takes two sub-parsers: one corresponding to S and one to R, as well as a function to feed to `fold_left`.

Figure 4.11 shows how to use a left-fold to generate a left-associative AST in the Autumn Java DSL for a grammar similar to that shown at the bottom of Figure 4.9. This is not the preferred way to implement operators in Autumn, as we will see shortly, but the example has didactic value.

We now briefly explain Figure 4.11. For a basic understanding of the Autumn framework, please refer to Chapter 3. The code snippet starts with the Java equivalent of the left-fold function from Figure 4.10. It is followed by the grammar rule `mult_op` which is a choice between the multiplication and division operator. Matching the given operator causes a corresponding AST node to be pushed on the stack, via the `as_val` combinator.

The grammar rule `mult_suffix` corresponds to the $(‘*’\ N \ |\ ‘/’\ N)$ part of the bottom grammar in Figure 4.9. Recall that the `push` combinator takes a function, whose result it will push on the value stack. The function's parameter is an array of nodes emitted by the parser on which `push` is applied (in this case, a sequence with two sub-parsers). These nodes have been popped from the value stack before being passed to the function. Here the function simply returns the node array.

The next rule, `mult_suffixes` corresponds to the $(‘*’\ N \ |\ ‘/’\ N)^*$ part of our grammar. The `repeat(0)` parse combinator takes the place of the Kleene star operator (0 denotes the minimum amount of repetitions). The resulting parser collects all the pairs generated by calls to the `mult_suffix` in an array, which is pushed on the value stack.

The final grammar rule, `mult_expr`, corresponds to the whole S rule of our grammar: $N\ (‘*’\ N \ |\ ‘/’\ N)^*$. `push` appears again and this time we call our left-fold function. The `$` function is a convenience offered by Autumn that auto-casts an item to the inferred expected type (one-parameter form), or indexes an array and similarly auto-casts the retrieved element (two-parameters form). `$(xs,0)` is a node for the integer matched by `integer_literal`, while `$(xs,1)` is the list of pairs pushed by `mult_suffixes`. The function passed to the fold simply unpacks a pair and constructs a `BinaryExpression` node, whose constructor takes

```
<T, U> T fold_left (T initial, U[] list, BiFunction<T, U, T> f) {
    T r = initial;
    for (U it: list) r = f.apply(r, it);
    return r;
}

rule mult_op = choice(
    STAR    .as_val(MULTIPLY),
    DIV     .as_val(DIVIDE));

rule mult_suffix =
    seq(mult_op, integer_literal)
    .push(xs -> xs);

rule mult_suffixes =
    mult_suffix.repeat(0)
    .push(xs -> xs);

rule mult_expr =
    seq(integer_literal, mult_suffixes)
    .push(xs ->
        fold_left($(xs,0), $(xs,1), (left, pair) -> {
          Object[] arr = $(pair);
          return BinaryExpression.mk($(arr[0]), $(left), $(arr[1]));
        }));
```

Figure 4.11: Using the Autumn Java DSL with the left-folding technique to generate left-associative trees for multiplicative arithmetic expressions.

an operator, a left-hand-side node and a right-hand-side node (in that order).

Depending on your perspective, this can either seem very elegant or very ugly. It is a very small amount of code to solve our problem, built on top of very general primitives. On the other hand, this code looks downright verbose and ugly in the middle of what we hope to be a mostly declarative grammar specification.

Fortunately, we can simply abstract the pattern away, by writing a new combinator that performs the work for us.

```
rule primary_expr = choice(
    seq("(", lazy(() -> this.expr), ")")
        .push(xs -> new ParenExpr($(xs,0))),
    number);

rule mult_op = choice(
    STAR    .as_val(MULTIPLY),
    DIV     .as_val(DIVIDE));

rule mult_expr = left_fold(
    primary_expr, mult_op, xs -> xs[1] == MULTIPLY
        ? new MulExpression($(xs,0), $(xs,2))
        : new DivExpression($(xs,0), $(xs,2)));

rule add_op = choice(
    PLUS    .as_val(ADD),
    SUB     .as_val(SUBTRACT));

rule add_expr = left_fold(
    mult_expr, add_op, xs -> xs[1] == ADD
        ? new AddExpression($(xs,0), $(xs,2))
        : new SubExpression($(xs,0), $(xs,2)));

rule expr = right_fold(
    add_expr, seq("?", add_expr, ":"),
        xs -> new TernaryExpr($(xs,0), $(xs,1), $(xs,2)));
```

Figure 4.12: Using Autumn's folding combinators to define a grammar with left-associative arithmetic operators, parenthesized expressions and a right-associative ternary operator.

```
StackAction.Push binary_push =
    (p,xs) -> new BinaryExpression($(xs,1), $(xs,0), $(xs,2));

rule mult_expr = left_fold(
    prefix_expr, mult_op, binary_push);

rule add_expr = left_fold(
    mult_expr, add_op, binary_push);
```

Figure 4.13: A refactoring of Figure 4.12 that assumes that all binary operator share the same node class.

4.8.3 Left-Folding in Autumn

Figure 4.12 shows how we would actually define the same arithmetic syntax as in Figure 4.4 via folding in Autumn.

We use the `left_fold` parsing combinator, which takes a sub-parser that is responsible for matching the left- and right-hand sides, as well as a sub-parser to match the operators; and automates the setup of Figure 4.11. The final argument is a stack action that corresponds to the action passed to the folding function.

We also use a `right_fold` combinator that builds right-associative parse trees. This is not necessary — it is easy to build right-associative trees using the standard combinators — but it makes for a pleasant symmetry in grammar definitions.

We support more forms of the `left_fold` and `right_fold` combinators. It is possible to specify different parsers for the left and right operands (given the left-/right-associative interpretation, the right-/left-hand-side parser will be used at most once). It is also possible to specify that an operator needs to be matched — in Figure 4.12, `add_expr` can match a single integer, without operators.

We could make the code slightly more elegant if we assumed that all binary operators shared the same node class. Then we could also factor out the stack action used to create that node — a small nicety that using an embedded DSL affords us. Figure 4.13 shows what the change would look like.

Is this the be-all end-all of left-associativity then? We are indeed getting

close to something simple and declarative, but we have to mention two small issues.

First, it is slightly unpleasant to have to factor out the operators into a choice of their own, then to dispatch on an operator value pushed on the stack in order to generate the correct node. This setup becomes even uglier if one tries to mix postfix operators with infix operators within a left fold (or equivalently prefix operators within a right fold).

Second, the `right_fold` combinator, just like transparent left-recursion, can easily be used in grammatical patterns that yield a significant slow-down.

Imagine a right-associative operator defined as $(L \text{ ‘}*\text{’}) * R$ (in the right-folding case, it is the prefix that is repeated). If L happens to be the same parser as R — as is in fact commonly the case — and the operator is not present, we will end up calling the parser twice: first as a potential prefix, then as the right-hand side. If you stack N right-associative on top of one another, then the problem compounds to give us once again slowdown by a factor of 2^N.

We made it so that the combinator will automatically detect when its left- and right-hand side are the same, so that it can locally memoize its results in that case. But it is still easy to run into the problem when both sides share sub-parsers — for instance if $R \to L \mid X$. We actually experienced this particular issue firsthand when refactoring our Java grammar to use `left_fold` and `right_fold`.

These two small issues lead to the definition of yet another pair of better, safer combinators, which are the object of the next section.

4.9 Defining Expression Families

What we would really like, ultimately, is to offer a one-size-fits(-almost)-all solution for defining the syntax of expressions — not only single operators, but whole *families* of expressions — or *algebra*, to the more mathematically-inclined.

In particular, learning from our previous attempts, we would like to avoid problematic performance scenarios that are easy to trigger by accident (like our `left_recursive` and `right_fold` combinators), and having to perform advanced grammar refactorings in order to define more than one

operator (as was the case in Figure 4.12).

We previously discussed what goes into the definition of expression syntax in Section 2.5 on expression parsing. Let us recall some points we made, which will guide our proposal.

Expression families will naturally include our usual nemeses, infix expressions — but also prefix and postfix expressions. Note that infix expressions do not only include binary expressions but also mixfix expressions of higher arity, *e.g.*, C and Java's ternary operator `a ? b : c`. These can be treated in the same way as binary expressions — as long as their constituent operators are distinct. Using identical operators (*e.g.*, `a : b : c`) leads to a new kind of associativity, namely middle-associativity — *i.e.*, interpreting `a : b : c : d : e` as `a : (b : c : d) : e`. We will see how we can in fact handle this pretty easily.

An expression family is typically divided into precedence levels. This is not the only possible model, but it is by far the most common. Programming languages syntax do admit exceptions to this model — for instance parentheses usually reset precedence, or there may be constraints on the operands of some operators — but it is general and well-understood enough to form the basis of our explanation and implementation.

One hard constraint that we shall impose is that all operators at a given precedence level must have the same associativity. Postfix operators are naturally left-associative and prefix operators are naturally right-associative, so we can mix infix and unary operators — and we definitely want to, as this happens in practice.[21]

In real programming languages, left- and right-associativity operators do not tend to mix at the same precedence level. Yet one can construct examples that do not seem totally far-fetched. Consider the grammar $A \rightarrow A + A \mid -A \mid a$, marked as left-associative. We want to parse $a + a + a$ as $+(+(a, a), a)$ (left-associative) but $- - a$ as $-(-(a))$ (right-associative). And because the operators are actually unambiguous, both mix seamlessly: $a + a + -a + a + a$ parses as $+(a, -(+(+(a, a), a)))$. We are strongly accustomed to unary operators binding more tightly than

[21]For instance, the `instanceof` operator in Java has the same precedence as ordering operators (`>=`, `<`, ...) but is effectively postfix since its right-hand side is a type and not an expression.

binary operators,[22] but it is hard to argue that anything is *fundamentally* wrong with this example.

Nevertheless, we do disallow these associativity mismatches within our new combinators, as they would add a lot of complexity for something that no one tries to do in practice. As a proof of the flexibility of our system, we will however show later how to re-introduce them.

Irrespective of associativity matching, operators may also be ambiguous: consider for instance a single operator used both in infix and postfix position. In this case, we fallback on PEG's ordered choice semantics: whatever was defined first takes precedence.

In order to build our new combinators, we want to synthesize the virtues of our two previous proposals. Transparent left-recursion handling (Section 4.5) makes it easy to combine multiple operators at the same precedence level: just make it a choice between left-recursive rules (cf. Figure 4.4). On the other hand, it is susceptible to poor performance when chaining multiple left-recursive combinators together — manual tuning is required to reclaim normal performance. Left-folding (Section 4.8), on the other hand, does not suffer the performance penalty[23] but requires left-factoring the expressions whose operator share a precedence level.

As already discussed at the end of Section 4.5.3, the issues with our implementation of transparent left-recursion handling cannot be fixed automatically. But the issues of folding are less fundamental, and amenable to a fix. We want a way to specify operators and AST construction logic without having to factor out the operators and then perform some kind of dispatch within the associated stack action (as is done in Figure 4.12). We also want to discourage the use of different operands for right-folding in order to prevent its performance pitfall.

We simply chose to name our new combinators `left_expression` and `right_expression`. Figure 4.14 shows how the DSL for these new combinators can be used in practice, to define the same syntax as Figure 4.4 and Figure 4.12. This ends up looking very much like the transparent solution, but without the performance issues. Under the wraps however, the logic is similar to that used by our folding combinators.

[22] And our combinators will actually enable that, but only when the associativity is the same.

[23] Though its right-folding dual sometimes does, as explained in Section 4.8.3.

```
rule primary_expr = choice(
    seq("(", lazy(() -> this.expr), ")")
        .push(xs -> new ParenExpr($(xs,0))),
    number);

rule mult_expr = left_expression()
    .operand(primary_expr)
    .operator("*", xs -> new MulExpr($(xs,0), $(xs,1)))
    .operator("/", xs -> new DivExpr($(xs,0), $(xs,1)))
    .get();

rule add_expr = left_expression()
    .operand(mult_expr)
    .operator("+", xs -> new AddExpr($(xs,0), $(xs,1)))
    .operator("-", xs -> new SubExpr($(xs,0), $(xs,1)))
    .get();

rule expr = right_expression()
    .operand(add_expr)
    .operator(seq("?", add_expr, ":"),
        xs -> new TernaryExpr($(xs,0), $(xs,1), $(xs,2))
    .get();
```

Figure 4.14: Using the Autumn DSL's `left` combinator to generate left-associative trees for multiplicative and additive arithmetic expressions.

The parsing expression used to parse the operands is specified using `.operand(...)`, but there is also a way to specify separate left and right operands (`.left(...)` and `.right(...)`) — but only for `left_expression`. For `right_expression` we supply the uglily-named methods `._maybe_slow_left(...)` and `._maybe_slow_right(...)` to nonetheless force specifying different operators if you know what you are doing and have taken steps to avoid the performance penalty (*e.g.*, using memoization).

As you may see, each rule corresponds to a level of precedence. The relationship between these levels is defined via simple layering via `operand` (and its variants that we just discussed).

Implementation-wise, the combinators work essentially in the same way as in Figure 4.12, with the difference that we now have separate stack actions for each operator.

We could have made it so that our DSL would have returned instances of `LeftFold` or `RightFold`, the implementation classes for our folding parsing expressions. However, we preferred to create a new kind of parsing expressions. The main reason is keeping the debugging output straightforward: using the translation approach would have made a couple of parsing expressions appear (choice between operators, suffix sequences) that the user did not specify. Using our new `LeftExpression` and `RightExpression` classes, the situation is simpler: the parsing expression has all operands and operators as direct sub-parsers. As an optimization, however, whenever there is only a single operator or suffix/prefix defined, a folding parser is returned — since no intermediate parsers are required in that case.

As promised earlier, we show how to achieve middle-associativity for mixfix operators of arity > 2 is fairly easy to achieve. For our ternary operator, for instance, it suffices to replace the middle operand by a recursive reference:

```
1 rule expr = right_expression()
2     .operand(add_expr)
3     .operator(seq("?", lazy(() -> this.expr), ":"),
4         xs -> new TernaryExpr($(xs,0), $(xs,1), $(xs,2))
5     .get();
```

As for mixing associativity, it is similarly simple to achieve with recursion. Below is a grammar implementing the left-associative version of rule $A \to A + A \mid -A \mid a$. You should be able to convince yourself that it does indeed implement the mixed-associativity semantics we discussed previously.

```
1 rule minus_expr =
2     seq("-", lazy(() -> this.expr))
3     .push(xs -> new MinuxExpr($(xs,0)));
4
5 rule expr = left_expression()
6     .operand(choice("a", minus_expr))
7     .operator("+", xs -> new PlusExpr($(xs,0), $(xs,1)))
8     .get();
```

4.10 Discussion

We presented no less than four ways to define the syntax of expressions, all able to create left-associative parse trees, a capability made difficult by the very semantics of the original PEG formalism.

The first of these solutions — transparent left recursion handling (Section 4.5) — aims to let the user write grammars as naturally as possible, and then annotate occurrences of left-recursion with a new parser combinator (`left_recursive`). In order to do this, we had to tackle the thorny question of what it meant for a PEG to be left-recursive (Section 4.3) and left-associative (Section 4.4). While the solution seems nice in isolation, it can lead to problematic performance when deployed to define the syntax of an expression family.

We also detailed how we can identify unhandled left recursion (*i.e.*, missing `left_recursion`) statically in Section 4.6. That capability could also be used to automatically insert the combinator, though we opted not to do so.

The second solution — expression clusters (Section 4.7) — let us define the syntax of whole expression families in a single rule. While we ultimately ruled it impractical and it is no longer a part of the Autumn library, it was a bridge of sorts between transparent left-recursion handling (whose algorithm it is based on) and our fourth solution.

Our third solution — folding combinators (Section 4.8) — offered a simple and clean way to specify left- and right-associative operators by folding a stack action over a list of suffixes (left fold) or prefixes (right fold). We particularly focused on left folds, as left-associativity is the problem to be solved. Left-folds are simple and handy, but they require a bit of contorsion to define the syntax of multiple operators at the same precedence level.

Finally, our last solution — expression family parsing (Section 4.9) — synthesized our previous approaches by supplying a way to define multiple operators at the same precedence level. It is implemented like the folding approach, but the declaration of precedence levels looks much cleaner.

Quite clearly, we like that final solution a lot. It is really hard to misuse and covers all the common use-cases (and then some less common too, through judicious use of recursion).

The other two solutions implemented in Autumn (transparent left recursion and folding) nevertheless have a place.

Transparent left recursion is particularly handy for porting grammars that were not designed with Autumn in mind. Though in that case, it is necessary to be careful about tuning the grammar so as to avoid bad performance.

Folding can still be used for left or right-associative constructs that are not part of an expression family. They can also look quite clean whenever all the operators at the same level of precedence share a single node representation (as showcased in Figure 4.13).

We do note that since Autumn checks for unhandled left-recursion by default, we're also not too worried about users accidentally introducing left-recursion. The error will be caught and then easily corrected.

Finally, we note that, in a certain sense, the "edge" that folding has over the transparent approach is that within folding we explicitly identify the higher-precedence operand — which is the equivalent of the "non-left-recursive part" (*i.e.*, potential seeds) in transparent handling. If we could identify that part, we could automatically memoize it!

4.11 Related Work

Feature-wise, some works have paved the way for full left-recursion, associativity and precedence handling within the PEG paradigm.

OMeta [106] is a tool for pattern matching over arbitrary data types. It was the first tool to implement left-recursion for PEGs [105], albeit allowing only right-associative parses.

Katahdin [83] is a language whose syntax and semantics are mutable at run-time. It pioneers some of the techniques we successfully improved upon — most notably the idea of blocking right-recursion to enable left-associative parses — but is not a parsing tool per se. Compared to Katahdin, our algorithm is much more widely applicable, as it can handle hidden left-recursion and middle-recursion (cf. Section 4.4). Katahdin's algorithm is also tangled with the handling of operator precedence, as well as the packrat store. Our algorithm makes the data structures explicit (even though it could still be backed by the packrat store). We also note that Seaton did not produce a formal description of the algorithm beyond the code of Katahdin itself.

Medeiros *et al.* [64] have, independently of us, devised their own approach to left-recursion in PEG. They approach the problem from a different direction, presenting first an axiomatic semantics of PEG which is augmented to support left-recursion, and then an operational semantics for a virtual PEG *parsing machine*, which can be used as a basis for implementation. Their approach also supports a form of associativity selection which is described and discussed in Section 4.4.5. Their approach is notably used in *IronMeta* — a port of OMeta to C#. However, IronMeta does not support associativity selection.

Our approach was to take the PEG formalism (and possible extensions thereof) and enrich it with user-friendly, expressive and safe support for infix expression parsing. Another plausible approach would be to start from the CFG formalism (which has an intuitive story for associativity selection, as stated in Section 4.4) and implement it as a parser combinator framework. CFG-based parser combinator frameworks exist [38, 85] but do not enable the definition of custom combinators.

Chapter 5

Principled Stateful Parsing

In Chapter 1, we made the case for improving parsing systems' flexibility and expressivity. In particular, we highlighted the fact that most parsing formalisms and tools do not support the definition of languages with context-sensitive features.

In this chapter, we start by giving some intuition as to what these context-sensitive language features are useful for (Section 5.1). Next, we present the state-of-the-art solutions in context-sensitive parsing (Section 5.2) and show that they lack the property of *context transparency*: they make grammars hard to write, maintain and compose by hardwiring context through the entire grammar (Section 5.3).

Instead, we approach context-sensitive parsing through the idea that parsers may *recall* previously matched input (or data derived therefrom) in order to make parsing decisions. We make use of mutable *parse state* to enable this form of recall.

More specifically, we introduce *principled stateful parsing* as a new transactional discipline that makes state changes transparent to parsing mechanisms such as backtracking and memoization. To enforce this discipline, users specify parsers using formally specified primitive state manipulation operations (Section 5.5).

Finally, we show how a parsing system can support principled stateful parsing (Section 5.6) as well as examples of how the approach can be used in practice. Our approach builds on top of the combinator-based

procedural approach we have been advocating, and introduces context-sensitivity via a log of reversible change as the last ingredient to achieve our vision for principled procedural parsing (cf. Section 1.4).

Most of the content of this chapter has been reworked from our 2016 *"Taming Context-Sensitive Languages with Principled Stateful Parsing"* [58] paper, while Section 5.6 on implementation and operationalization is entirely new.

5.1 Context-Sensitive Parsing

It might not be immediately evident what we mean by *context-sensitive language features*. The usual way to characterize this class of features is by contrasting Chomsky's Context-Free Grammars (CFGs) with his Context-Sensitive Grammars (CSGs) [14]. However, this is not precisely what we have in mind here. Instead, we propose to characterize context-sensitivity through the notion of *recall*, which we present in Section 5.1.1. We then briefly explain why Chomsky's CSGs are unhelpful to tackle the problem in Section 5.1.2. Finally, we outline how the parser combinator approach is particularly amenable to recall-based context-sensitive parsing in Section 5.1.3.

5.1.1 Recall and Context-Sensitive Features

We propose to approach context sensitivity through the notion of *recall*, *i.e.*, the ability to accept sentences based on relationships between some of their parts. This is more easily understood in parsing terms as the capability to make parsing decisions based on previously matched input.

Here are a couple of examples of such recall-based context-sensitive language features:

- In C, in order to determine whether the statement `x*y;` is the product of x by y, or rather the declaration of a variable y which is a pointer to type x, one must analyze the type definitions preceding the statement.

- In Haskell and Standard ML, programmers can introduce operators with custom precedence and associativity. The parser needs to interpret these definitions in order to be able to parse the remainder of the input.

- Since Python has significant indentation, a Python parser needs to detect when the indentation level increases or decreases.

- In XML, opening tags must be matched with corresponding closing tags. For instance, `<foo></foo>` is valid while `<foo></bar>` is not. As such, an XML parser must memorize the names of open tags, at arbitrary levels of nesting.

- Many network protocols, including TCP, make use of length-delimited fields whose length is not known in advance but indicated by a length field that precedes them.

Most parsing formalisms and tools cannot adequately handle these syntactic peculiarities, leading to all sorts of hacks, and to the rejection of parsing tools altogether, causing developers to write ad-hoc parsers by hand. There are a few exceptions however, which we review in Section 5.2. We will then explain the key properties almost all these solutions lack, namely *context transparency*, in Section 5.3.

5.1.2 Context-Sensitive Grammars

The term *context-sensitive* is often associated with Chomsky's *Context Sensitive Grammars* (CSGs) [14], a grammar formalism that is strictly more expressive than CFGs, by virtue of allowing rules to be bounded by a "context", *i.e.*, strings of symbols to appear around the nonterminal to be expanded.

Nevertheless, CSGs are only of little help, due to the intricate coding that they require[1]. A CSG is made of rewrite rules $\alpha X \beta \rightarrow \alpha \gamma \beta$ where α, β and γ are strings of mixed terminals and nonterminals. These rules must be non-contracting: $\alpha \gamma \beta$, as a string of symbols, must not be shorter than $\alpha X \beta$.[2] As a matter of fact, these grammars were never meant to describe programming languages, but natural languages, where the shape of the rules make much more sense. In particular, it is difficult to encode *recall* constraints: for instance, requiring the same string to appear at two different locations in the sentence (assuming the string is not fixed in advance).

[1]The same holds for a large body of work on *mildly context-sensitive grammars*. [44]

[2]In reality, CSG rules are not required to be non-contracting, but non-contracting grammars and CSG describe the same set of languages, [15] and non-contracting grammars are easier to operationalize in practice.

As a result, parsing with CSGs has seldom been applied to programming languages. Writing CSGs can prove challenging; for instance the grammar for the language $a^n b^n c^n$ — exemplifying a relatively simple form of recall — is notoriously tricky [32].

In fact, CSGs have not had much applications in natural parsing either. The issue is that parsing CSGs is computationally intractable, as the problem as been proven to be PSPACE-complete [53]. PSPACE — being the set of problems that can be solved using a polynomial amount a space — includes NP, the set of problems solvable in nondeterministic polynomial time. The existence of a deterministic polynomial time parsing algorithm for CSGs would therefore imply P = NP, which we know to be *quite unlikely*.

5.1.3 Context-Sensitivity & Parser Combinators

Our work builds on top of the parser combinator approach to enable *recall*. We allow users to write parsers which can manipulate mutable state. However, unlike most parsing tools that allow state modifications (*e.g.*, ANTLR [72], Rats! [31]), we are *principled* about state use. Indeed, it is relatively easy to mess up the state if the parsing tool does not take special provisions to maintain its integrity.

General parsing algorithms typically do not proceed linearly. They either explore multiple choices in parallel (typical of general CFG parsing algorithms); or speculatively try alternatives when faced with a choice, then backtrack if this alternative does not succeed (typical of PEG parsing algorithms).

In the case of parallel exploration, multiple copies of the state need to be kept, and potentially merged together. While this is certainly possible, it does not play nicely with the semantics of most programming languages. State has to be defined in a way that enables it to be forked and merged as required. This can be achieved through the use of specialized data structures.

Our work focuses on fixing the speculative execution scenario. When backtracking happens, all changes made to the state during the speculative execution need to be reversed. Parsers may also memoize the result of a speculative execution. In a stateful model, these results need to include the state changes incurred by the execution. As will be explained in Section 5.4, we satisfy these requirements by introducing primitive operations to manipulate mutable state in a principled way. All that is

required from the user is to provide state changes as a pair of functions: one that applies the change, and one that undoes it.

It would be disingenuous to suggest that we did compare context-sensitivity-handling solutions for both approaches (speculative and parallel execution) on equal terms. Having made a predetermination that the combinator approach would be conducive to our objectives, we endeavoured to build context-sensitivity on top of it.

Nevertheless, we do sincerely believe that the speculative approach is more advantageous in terms of extensibility and user experience, at least in the context of *recall*. The user (grammar author) ultimately has to implement these state changes himself, and supplying changes along with an undo function is easier than writing routines to fork and merge complex data structures.

5.2 State of The Art

As the problem of context-sensitive features in programming languages is not new, it is not surprising that several solutions have been proposed. We review these solutions in order to better put our contributions in perspective. We do not purport to review the entire body of work on context-sensitive parsing, but only the approaches closest to our goal. In particular, we left out the literature on context-sensitive lexical analysis (*e.g.*, [99, 5]) which by definition only handles a small subset of all context sensitivity issues.

5.2.1 Backtracking Semantic Actions

Parsing with backtracking semantic actions [93] is an approach developed by Thurston and Cordy that builds upon a modified form of backtracking LR algorithm[3] with reversible semantic actions. Upon backtracking, state changes are reversed. Two important restrictions apply: state changes can only occur during reductions, and the state can only affect the parse through semantic conditions that trigger backtracking.

[3] A backtracking LR parser is a LR parser that, upon encountering a conflict, tries one possible action and subsequently backtracks to the other actions if the choice leads to a parse failure. This is different from GLR, which is a LR parser that explores conflicting actions in parallel. Backtracking LR is generally eschewed in favour of GLR which has much more reasonable bounds and practical run time on nondeterministic grammars. See Section 2.1.4 for more details on LR and GLR.

The backtracking semantic actions approach is employed in the Colm [92] source transformation language developed by Thurston as his PhD thesis project.

The major difference between backtracking semantic actions and our approach is that it builds upon bottom-up parsing while our approach builds upon top-down recursive-descent parsing. Backtracking semantic actions can only run semantic actions and predicates after reductions. This is already quite powerful — it can be used to implement all the examples from Section 5.1.1 except Haskell/SML — but it does not allow defining custom parsing operators that make use of the state.

For instance, our approach could handle a language that enables programs to extend the language's own grammar (such as Haskell and SML). This is impossible using only semantic predicates.[4] In essence, the semantic predicates allow for disambiguation between productions, but cannot match data directly. As predicates run upon reduction, they also cannot be used to *guard* a nonterminal, preventing the parser from attempting a match that will necessarily be discarded.

To reverse semantic actions that get backtracked over, the Colm implementation allows either manual specification of undo actions, or the automatic computation of the undo procedure (*reverse execution*). This is made possible by the fact that the language runs on a custom virtual machine, which is able to record reverse instructions as it executes a semantic action. In contrast, Autumn runs as a Java DSL and we require the user to specify the undo actions manually, although with some added facilities for capturing state to be restored via lambda capture, as well as abstracting away and composing the undo logic. These points are further elaborated in Section 5.6.3.

We consider backtracking semantic actions [93] to be the safest and most convenient system for context-sensitive parsing among those presented in this section. In a sense, it is the bottom-up mirror of our approach — as both employ backtracking and undo actions. The differences can largely be explained by the underlying systems — our system being geared towards pervasive extensibility, while LR is harder to steer and extend. We also enable state handling beyond change reversal: for

[4] However, in his Colm [92] language, the author pairs this algorithm with a lexer that can be augmented with token-generation actions. The lexer can then be used to solve the issue by collaborating with the parser.

instance Autumn enables capturing the set of state changes produced by a parser. This is quite useful in order to support features such as memoization, or capturing a set of settings and declarations that have to be brought back "in scope" later.

5.2.2 Data-Dependent Grammars

Jim *et al.* [41] proposed data-dependent grammars, a formalism which permits context sensitivity by allowing rules to be parameterized by semantic values. A parameterized nonterminal appearing on the right-hand side of a rule acts as a form of function call that also returns a semantic value. These semantic values are computed by *semantic actions* written in a general-purpose programming language. There are also *semantic predicates* which can make rule alternatives succeed or fail depending on a semantic value.

Data-dependent grammars can be compiled to a format accepted by a target parsing tool, which must support fairly general semantic actions. In subsequent work [40], the authors introduced a new kind of automaton that can be used to implement parsers recognizing data-dependent grammars. These techniques are put to work in a tool called Yakker.

Data-dependent grammars, though theoretically compelling, suffer from usability issues. The value-passing model means that the parse state needs to be threaded throughout the grammar. Making a rule dependent on a new semantic value means that all rules through which this rule is reachable might need to be modified to pass this value around. Maintainability-wise, this is far from ideal. Moreover, it harms composability, as a rule must be aware of all states it has to pass through.

Afroozeh and Izmaylova [2] show how advanced parser features such as lexical disambiguation filters, operator precedence, significant indentation and conditional preprocessor directives can be translated to data-dependent grammars. Quite clearly, the task is non-trivial and one comes away with the feeling that dependent grammars are better suited as an elegant calculus to be targeted by parsing tool writers rather than as a paradigm that fits the needs of tool users. The machinery implementing the formalism is also distinctively non-trivial, involving a multi-stage transformation into a continuation routine or into a new kind of automaton. In contrast, our approach is conceptually simpler and can be layered on top of a general-purpose programming language.

Finally, we note that the much older Definite Clause Grammars

(DCGs) [17] formalism (cf. Section 2.1.3) works on almost exactly the same principle, but building upon logic programming. Accordingly, it suffers from similar limitations.

5.2.3 Monadic Parsers

Monadic parsing [35] is a well-known way to build functional-style parser-combinator libraries, made popular by Haskell libraries such as Parsec [61]. In this paradigm, the type of a parser is a function parameterized by a result type, *i.e.*, with signature $string \to (string, result)$, where the parameter string is the input text and the returned string is the input remaining after parsing. The parser type is also a monad instance, meaning there is a `bind` function whose signature, in Haskell notation, is:

```
Parser r1 -> (r1 -> Parser r2) -> Parser r2
```

where `r1` and `r2` are result types. This function takes a parser as first parameter, and a function which transforms the result of the parse into another parser as second parameter. When invoked, the parser returned by `bind` will invoke the first parser, pass its result (of type `r1`) to the function, then invoke the parser this function returns, yielding a result of type `r2`.

The important point about monadic parsers is that they can handle context sensitivity. Indeed, the second parameter to `bind` (the function) returns a parser from a result. This means that the behaviour of the parser returned by `bind` depends on data acquired during the parse: this is a form of *recall*.

An in-depth analysis of this aspect was done by Atkey [4]. In particular, he formalizes monadic parsers by introducing *active right-hand sides*, which are the right-hand sides of rules that can contain monadic combinators. These combinators generate grammar fragments at parse-time (much like a monadic parser generates a new parser), hence the term *active*. While monadic parsing seems at first sight very similar to the data-dependent grammars from Section 5.2.2, Atkey [4] carefully contrasts the two approaches:

> *We characterise their [Jim* et al. *] approach as refining context-free grammars: each Yakker grammar has an underlying context-free grammar with regular right-hand sides, and the constraints allow for sophisticated data-dependent filtering of parses. In contrast, we consider active right-hand sides that generate the grammar as the*

input is read.

Nevertheless, monadic parsers suffer from the same pitfalls as data-dependent grammars: the state is threaded through the grammar (or code), leading to poor maintainability and composability.

5.2.4 Attribute Grammars

Attribute grammars [51] associate attributes to syntax tree nodes. The attributes can be synthesized: their value derived from the attributes of sub-nodes, or inherited: their value computed by a parent node. The formalism supports context-sensitive parsing through production guards predicated over attributes.

However, attribute grammars are not context-transparent. To enable recall, they need to propagate the recalled value from the definition site to the use site, through a chain of of synthesized and inherited attributes. Even reference attributed grammars [33], which allow attributes to contain references to nodes, do not fully solve this distribution problem.

5.2.5 Unprincipled Stateful Parsing

Manipulating parse-wide state can be an effective solution to the problem of data dependence: the data depended upon can be written in the state when encountered and read or even altered later on.

Broadly speaking, we can distinguish two big classes of stateful parsing tools. First, there are parser combinator libraries that allow users to write their own sub-parsers. Notable examples include Parboiled [23], Lua Peg [36] and Scala's parser combinators [67]. Since these custom parsers are implemented in a general-purpose programming language, they can manipulate state, even though the libraries make no provision for this. Second, there are parsing tools that provide very general semantic actions and semantic predicates. Notable examples include Bison [91] and ANTLR [72]. These work much like their counterpart in Yakker (cf. Section 5.2.2), except that instead of returning a value, semantic actions may modify a global state object.

Unfortunately, most parsing tools in both categories do not make the necessary provisions for dealing with backtracking and memoization: if the parser backtracks over a construct that made state changes (semantic action or custom parser), these changes need to be undone; if the parser can memoize the result of a construct, state changes need to be memoized

as well. In the absence of such guarantees, a construct can only access state which it is sure has not been corrupted by changes that should have been discarded. It must also be sure that some state-altering construct was not skipped due to memoization. These are tricky propositions to verify even for medium-sized grammars, and every change to the grammar threatens to falsify them.

One may think that solving the backtracking problem is simply a matter of inserting a construct that reverses state changes whenever a rule fails. However, a rule can be backtracked over even if it succeeded. It suffices that one of the rules through which our rule was reached fails. Hence this scheme would entail, for each state-altering construct, the modification of every rule through which it can be reached.

5.2.6 Rats!

Rats! [31] is a fully-memoizing (*packrat*) PEG parser. Rats! is, to the best of our knowledge, the only stateful parsing tool that provides some guarantees for state usage, by ensuring that state changes are discarded if certain conditions are met.

For this purpose, Rats! introduces *transactions* that wrap rules under which state changes might occur. A transaction can either succeed, in which case its state changes are retained, or fail, in which case the changes are discarded. Rats! also requires that a nonterminal invoked at a given position within a transaction must always modify the state in the same way, no matter how that nonterminal was reached. This requirement ensures that Rats! will never have to discard the memoization of a rule, hence upholding the linear-time guarantee of packrat parsers.

In spite of its advantages, this scheme has two important pitfalls. First, it requires nonterminal invocations at a given position to always return the same result. This precludes parsing expressions that modify the behaviour of the parsing expression they invoke. However, this capability is valuable in practice. For example, we use it to enable transparent left-recursion handling in Autumn in the presence of context-sensitivity, as well as to enable longest-match parsing in the same conditions.[5]

Second, state changes are not memoized. If a rule succeeds after applying a state change, but the enclosing transaction fails, the changes are lost.

[5]See Section 4.5 for details on transparent left-recursion handling, and Section 5.6.4 for implementation details on the `Longest` parser.

If we wanted to call the rule at the same position again, the memoized result would be used and it does not include the state changes. This means that a state change cannot safely be referenced by two different transactions, and that transactions cannot be re-tried after a state change higher up in the grammar hierarchy.

5.2.7 Marpa and Ruby Slippers

The Marpa [45] parsing framework enables some context-sensitive language features using a technique called *ruby slippers*. Marpa is based on the Earley algorithm, a chart-based algorithm that performs a breadth-first exploration of the possible parses (cf. Section 2.1.5). The Earley algorithm consequently never backtracks and Jeffrey Kegler, the author, observes that it is possible for the parser to become *left-eidetic*: at each position, aware of every possible interpretation of the input already processed.[6]

Marpa also enables for grammar rules to generate events, which are processed by a user-supplied event handler. Essentially, semantic actions. Because an Earley parser never backtracks, the event handler can manipulate the state in a way that is *relatively* safe: it is not necessary to undo the state due to backtracking, but changes to the state have to consider that there may be multiple concurrent interpretations of the parse (due to Earley's breadth-first nature). The left-eidetic property does mitigate the issue by enabling inspection of these interpretations, but one still has to be careful of the impact of future grammar changes.

Finally, the event handler is limited in its capability to affect the parse, being only able to manipulate the lexical (token) stream, and potentially to restart a parse to undo previous changes to the stream. Nevertheless, the ability to generate virtual tokens — the *ruby slippers* technique — does enable to solve many context-sensitive issues such as semicolon insertion or layout-sensitive parsing. [46]

5.3 Context Transparency

As the previous section has shown, enabling the definition of context-sensitive languages without jeopardizing maintainability, composability or even

[6] Strictly speaking, each such "interpretation" is a minimal subset from the set of states held in the Earley chart (cf. Section 2.1.5) that could lead to the current Earley state. In practice, it helps to think of these interpretations as partial ASTs for the left part of the input.

safety is no easy feat. We put forward the notion of *context transparency* as the gold standard that a context sensitive parsing mechanism needs to meet in order to be considered sufficiently practical.

> A grammatical construct is **context-transparent** if it is unaware of the context shared between its ancestors and its descendants — meaning this shared context is not encoded in the declaration of the construct.

Data-dependent grammars, monadic parsers, DCGs and attribute grammars are not context-transparent because of the need to explicitly pass values around. For instance, consider two data-dependent grammars[7]: a grammar for a Python-like language with significant indentation, in which the rules for block-level constructs (statements, definitions) are parameterized by the indentation level; and a grammar for a simple query language (something like SQL), in which newlines can be used as whitespace separator. We want queries to be able to appear inside our code, potentially over multiple lines, while preserving the indentation requirement of the Python-like language.

A simple solution is to rewrite the whitespace-matching rule of the query language to check that any newline is followed by enough whitespace to match the current indentation level.

The difficulty is in how to communicate the current whitespace level to this rule. In all the previously cited grammar paradigms, every rule that may directly or indirectly match whitespace (which is to say all of them) must be rewritten to pass the indentation level around explicitly, so that the whitespace matching rule may access it. In a context-transparent grammar paradigm, you could simply access the indentation level from the whitespace-matching rule without rewriting any intermediary rules.

Stateful parsers also are not context-transparent, as they must ensure that no unforeseen backtracking or memoization takes place. For instance, if a parser a manipulates the state and its callers do not expect it to backtrack, it cannot be swapped for a parser $c(a)$ (where c is some parser combinator) without first ensuring that $c(a)$ never backtracks over a.

[7] The same reasoning applies to monadic parsers, DCGs and attribute grammars.

Marpa is not context-transparent either, as introducing new ambiguity (or even nondeterminism!) in the grammar can cause unexpected results because of lexical-level transformations made by previously-defined event handlers. In this case, the issue is not so much that new rules have to be made context-aware, but rather that they could be affected by lexical-level changes — so that the old rules have to be made aware of the new rules they could interfere with.

Lack of context transparency makes grammars hard to reason about, hence hard to write and to maintain: refactoring, extending or composing grammars becomes particularly challenging, because each change to a rule might entail the need to modify all rules through which it is (transitively) reachable. In stateful parsers, such changes are liable to introduce undesired backtracking or memoization.

We suggest a simple solution: use stateful parsing (which does not thread context through the grammar), but undo state changes upon backtracking and allow the memoization of state changes. And to achieve this, we introduce a new context sensitivity handling discipline: *principled* stateful parsing.

5.4 Intuition

In Section 5.1, we established the relevance of context-sensitive parsing and introduced the notion of *recall* as a way to express context-sensitive features in terms of backreferences to previously matched input. We enable recall by storing the matched input (or data derived thereof) in a mutable data store: the *parse state*.

We now dive into how *principled stateful parsing* is able to work with parse state while avoiding the usual pitfalls of stateful parsing (cf. Sections 5.2.5 and 5.3). Before diving into a formal explanation, we present the remarkably simple intuition behind the approach.

The point of using state is to pass context around implicitly, without the need to hardwire context in the grammar, hence achieving context transparency (cf. Section 5.3).

If the execution of a parser were linear, simply reading/writing to this state would suffice. Unfortunately, parsers must sometimes perform speculative executions that may fail further down the line, a phenomenon called backtracking. When backtracking occurs, all state changes in the

speculative execution being backtracked over must be reversed. Hence, we need an operation that can take a **snapshot** of the state at a given point, and an operation that can **restore** the state described by such a snapshot.

Given these requirements, it helps to think of the parse state as a log of the operations applied to the state, which can be snapshot and rolled back as required. Appropriately, this is also how we formalize the parse state.[8]

As an aside, let's note that it is not a requirement for these snapshots to be fully self-contained or persistent. The actual requirement is for them to be valid at long as we could have to **restore** them.

Additionally, it is sometimes desirable to save the result of a speculative execution (whether it failed or not), *i.e.*, the state changes it induced: a *delta* acquired by performing a **diff** between the states before and after the execution. This can be represented as a slice of the log of state modifications. It is also necessary to be able to **merge** these changes back into the state. The most straightforward application of the *diff* and *merge* capabilities is the memoization of parse results. However, other valuable use cases exist in the presence of context-sensitivity, such as longest-match (Section 5.6.4) parsing and transparent left-recursive parsing (Section 4.5).

This motivates the need for four primitive state-manipulation operations: **snapshot**, **restore**, **diff** and **merge**. These operations are described in Section 5.5.3.

> **Principled stateful parsing** is an approach where parsers behave transactionally: each parser invocation either succeeds or leaves the state untouched. Additionally, it is possible to generate and merge deltas corresponding to state changes made by parser invocations. All this is made possible through the use of formally specified state manipulation operations.

[8] The implementation also closely follows this principle, each state change being represented as a pair of *apply* and *undo* actions.

Principled Stateful Parsing and Transactional Databases

Our state handling system bears some similarity to transactional databases — we even used the word "transactional" to describe it. There are indeed similarities: parsers may induce state changes. If a parser fails, all the changes induced by itself and all of its successful sub-parsers must be undone, just like a database transaction.

However, the metaphor only goes so far: transactional databases typically deal with multiple transactions, which may be concurrent. A parse is more similar to a hierarchy of nested transactions (one per parser), with the grammar's root corresponding to the outermost transaction.

Our parsing system is also not concerned with persistence: all parse state is transient, so should a system failure occur during a parse, the state is scraped and the parse has to be started anew.

Nevertheless, the comparison is a good way to understand principled stateful parsing as a first approximation. And it is possible that pondering the similarities of both systems might yield interesting insights for future developments.

5.5 Formalization

We formalize our approach using the Z notation [86], though eschewing its schema calculus in favor of a purely functional presentation.[9] The Z notation is a formal specification language that builds on top of Zermelo-Frankel set theory, first-order logic and simply typed lambda calculus. As such, Z can be seen as a language where functions can be defined in lambda calculus extended with predicates from first-order logic and set theory. Formal assertions over the functions can be made using the same notation. We also note that in Z, all types used in the lambda calculus are sets. We will introduce elements of notation as we go, but if any doubt persists, we recommend checking *The Z notation* by Spivey and Abrial [86].

[9] To improve the presentation, we took some liberty with the Z layout (but not with the notation). A machine-understandable version of the specification is available online [59].

In our formalization, parsers are simply functions manipulating parse state (Section 5.5.1) whose set-theoretic signature is given in Section 5.5.2. Section 5.5.3 formally specifies the primitive state-manipulation operations that were briefly introduced in Section 5.4. Finally, Section 5.5.4 gives the semantics of parser invocation by specifying the *call* operation, which maps a parser (as defined in Section 5.5.2) to a single state transformation.

5.5.1 Parse State

At the core of our approach lies the notion of parse state. The parse state abstracts over a general mutable data store. We do not place any constraint on the data within the store. This is formalized as follows.

$$[\mathit{CHANGE}]$$
$$\mathit{STATE} == \mathrm{seq}\ \mathit{CHANGE}$$

The square brackets introduce the abstract set *CHANGE* of all state changes. What exactly constitutes a state change (most likely the mutation of a memory location) is an implementation concern that is not relevant to the formalization. If it helps, you may think of the state as a key-value store, and changes as writes or deletions from this store — though this is entirely immaterial to the rest of the formalization.

STATE is the set of possible parse states: *i.e.*, of possible configurations of our mutable store. We represent a parse state as a sequence of state changes. This means that a state can be seen as a log of the operations over the mutable store it represents, assuming some well-defined initial state.

In Z, the set of sequences of items from the set S is written $seq\ \ S$ and corresponds to the power set of pairs $(i, s) \in \mathbb{N} \times S$, or equivalently to the power set of partial functions $\mathbb{N} \rightarrowtail\!\!\!\!\!\!\rightarrow S$. In each sequence, the indices are unique and consecutive.

In practice, an implementation of the approach will want to use parse state to reify important parsing notions, such as input position. We consciously avoided making our formalism needlessly specific, hence the absence of some usual parsing notions such as input position. This enables using our approach to parse non-linear inputs (*e.g.*, object graphs), or perform computations that only bear nominal resemblance to traditional parsing, even though this direction is outside the scope of this thesis.

5.5.2 Parsers

A parser represents a computation over the parse state that either succeeds or fails, and has side effects on the parse state, in the form of *state changes*, as introduced in the previous section.

$$\begin{aligned}
&TRANSFORM = STATE \to STATE \\
&PARSER = STATE \to \text{seq } TRANSFORM \\
&RESULT ::= success \mid failure \\
&result : STATE \to PARSER \to RESULT
\end{aligned}$$

Formally, a parser is a function from a state — the current state at the time of invocation — to a sequence of transformations, which move from one state to another. This amounts to defining a parser in terms of its execution trace.

Two things seem to be missing from this definition. First, it does not say if the parse succeeds when run over a specific state. This property is exposed separately through the *result* predicate rather than as part of the *PARSER* signature. This approach is not significant: it simply makes the math look nicer. Second, the input being parsed does not explicitly appear in the signature. Instead, the input is assumed to be held within the parse state.[10]

A parser is a recognizer of states. It accepts states for which $result\ state = success$ holds. If within the input state one dissociates the *parse input* from the rest of the state (the *context*), one can see that the parser recognizes — hence also defines — different languages depending on the context.

But a parser is also a transformer of states as well: when invoked it performs a $STATE \to STATE$ transformation. In Section 5.5.4 we explain how to derive this transformation from a parser (recall that parsers have type *PARSER* defined as $STATE \to seq\ TRANSFORM$), as a means of defining the semantics of a parser given its execution trace. We could alternatively have defined *PARSER* as $STATE \to TRANSFORM$ (with the result being the composition of the transformations in the sequence), or directly as $STATE \to STATE$. We chose to emphasize the execution trace — a sequence of transformations — instead, because the primitive state operations described in the next section are suppliers

[10] Nothing precludes the input from being mutable, even though we have not investigated the usefulness of the idea.

of such transformations, to be composed to yield the transformation performed by the parser.

This representation also emphasizes that the parse state is both an input of the parser and an input of the returned transformations. This reflects the fact that a parser is context-sensitive: it chooses which operation to perform depending on the state. This is closely related to the notions of active right-hand sides [4] and monadic parsing [35]. In fact, each operation in the sequence is chosen depending on the state obtained by running the initial state through the composition of all preceding transformations. Abstracting over this makes the specification much simpler, without altering its meaning.

5.5.3 Primitive Operations

We now present six primitive operations (amongst which the four announced in Section 5.4) that parsers can perform.

$$
\begin{aligned}
&\mathit{SNAPSHOT} = \mathrm{seq}\ \mathit{CHANGE}\\
&\mathit{DELTA} = \mathrm{seq}\ \mathit{CHANGE}\\
&\mathit{call} : \mathit{PARSER} \to \mathit{TRANSFORM}\\
&\mathit{snapshot} : \mathit{STATE} \to \mathit{STATE}\\
&\mathit{diff} : \mathit{SNAPSHOT} \to \mathit{STATE} \to \mathit{DELTA}\\
&\mathit{applyChange} : \mathit{CHANGE} \to \mathit{TRANSFORM}\\
&\mathit{restore} : \mathit{SNAPSHOT} \to \mathit{TRANSFORM}\\
&\mathit{merge} : \mathit{DELTA} \to \mathit{TRANSFORM}
\end{aligned}
$$

Call Of these six, *call* has a special status: it represents the invocation of a parser. We will define this operation in Section 5.5.4, hence specifying the semantics of parsers given their execution trace. Note that the signature definition of *call* expands to $\mathit{PARSER} \to \mathit{STATE} \to \mathit{STATE}$: a parser must be called with a state as parameter.

Snapshot A snapshot, as the name implies, is a capture of the state at a specific point during the execution. Naturally, this makes $\mathit{SNAPSHOT}$, the set of all snapshots, equivalent to STATE. Formally, the *snapshot* operation, which creates such a capture, is simply the identity function.

$$\mathit{snapshot} = \lambda\, x : \mathit{STATE} \bullet x$$

Diff The *diff* operation returns a DELTA object representing the difference between a snapshot and the current state, as a set of state changes.

As a precondition, this operation requires the snapshot it receives to be a prefix of the current state. This is expressed with the Z built-in *prefix* infix operator. By keeping the deltas append-only, we ensure that a delta can be later *merged* to any state, not just the one corresponding to the snapshot.

$$\forall\, sn : \mathrm{dom}\, diff \bullet \forall\, st : \mathrm{dom}\, (diff\ \ sn) \bullet \\ \quad sn\ prefix\ st$$

Since deltas are state suffixes, *DELTA*, the state of all deltas, is equivalent to *STATE*.

Assuming the precondition is respected, *diff* can be defined as the remainder of the current state after chopping off the prefix corresponding to the snapshot. In Z, the *squash* function packs the indices (left-hand side) of a set of pairs in $\mathbb{N} \times S$, where S is some set, in order to turn this set into a proper sequence. For instance, it turns $\{(2, x), (5, y)\}$ to $\{(1, x), (2, y)\}$.

$$diff = \lambda\, sn : SNAPSHOT \bullet \lambda\, st : STATE \bullet \\ \quad squash\ (st \setminus sn)$$

Transformations All operations except *diff* and *snapshot* return a transformation. Recall that we defined *PARSER* as $STATE \rightarrow seq\ TRANSFORM$. The transformations returned by the operations are precisely those which will be part of a parser's execution trace. *diff* and *snapshot* are different because they do not modify the parse state. Instead, *diff* and *snapshot* create new objects, which can be freely passed through the parse state.

ApplyChange The *applyChange* operation is very simple: given a change, it simply returns a transformation that applies this change, by appending it to the change log. It can be defined as follows, using the concatenation operator ($\frown$) to append the change to the old log.

$$applyChange = \lambda\, c : CHANGE \bullet \lambda\, st : STATE \bullet \\ \quad st \frown \langle c \rangle$$

This "operation" models the fact that parsers can perform arbitrary state changes.

Restore The *restore* operation takes a snapshot as input and returns a transformation that brings the state to that described by the snapshot.

$$restore = \lambda\, sn : SNAPSHOT \bullet \lambda\, st : STATE \bullet sn$$

Merge The *merge* operation takes a delta as input and returns a transformation that appends this delta to the input state.

$$merge = \lambda\, d : DELTA \bullet \lambda\, st : STATE \bullet st \frown d$$

5.5.4 Parser Invocation Semantics

We now look at how the transformation returned by the *call* operation can be derived from the execution trace returned by a parser. Recall that the *call* operation's signature is $PARSER \to TRANSFORM$.

We start by defining two helper functions. *composeTwo* maps sequences of transformations of length $n \geq 2$ to a sequence of length $n - 1$ similar to the input sequence, but where the first two items have been replaced by their composition ($s\ 1$ and $s\ 2$ (function calls!) access the first two items of s while $\fatsemi$ is the relational composition operator). *reduceN* takes a natural n and a sequence of transformations and returns the composition of its n first items, or the identity transformation if $n = 0$. This is achieved by iteratively running the sequence through *composeTwo*, using the Z built-in *iter* operator.

$$\begin{aligned}
&composeTwo = \lambda\, s : \mathrm{seq}\ TRANSFORM \bullet \\
&\quad \langle s\ 1 \fatsemi s\ 2 \rangle \frown tail\ (tail\ s) \\
&reduceN = \lambda\, n : \mathbb{N} \bullet \lambda\, s : \mathrm{seq}\ TRANSFORM \bullet \\
&\quad \textbf{if}\ (n = 0)\ \textbf{then}\ \mathrm{id}\, STATE \\
&\quad \textbf{else}\ iter\ (n - 1)\ composeTwo\ s\ 1
\end{aligned}$$

With this in place, we define the result of *call* as the composition of all transformations within the call's execution trace, assuming the parser invocation is successful. Otherwise, the identity transformation is returned. The hash sign (#) is an operator returning the cardinality of a set.

$$\begin{aligned}
&call = \lambda\, p : PARSER \bullet \lambda\, st : STATE \bullet \\
&\quad \textbf{if}\ (result\ st\ p = success) \\
&\qquad \textbf{then}\ reduceN\ (\#p\ st)\ (p\ st)\ st \\
&\quad \textbf{else}\ st
\end{aligned}$$

5.6 Implementation

The preceding sections should have given you a good understanding of the theory behind our principled state handling strategy. This section explains how we actually deploy this theory in practice.

We start by a brief explanation of the transition from theory to practice (Section 5.6.1). We then show how our primitive state manipulation operations can be translated to implementation concepts (Section 5.6.2). We also address some operationalization and usability concerns (Section 5.6.3). We give concrete examples of both context-sensitive parsing and the use of state manipulation operations to make existing parser compatible with context-sensitivity (Section 5.6.4). Finally, we discuss possible alternative implementation (Section 5.6.5).

5.6.1 From Theory to Practice

In the formalization (Section 5.5), the state is represented as a sequence of changes. You will also recall that a parser receives a state, then based on it computes a sequence of transformations (state transitions), which corresponds to the primitive operations (Section 5.5.3) it calls during its invocation. The result of calling a parser is the composition of all these transformations, *i.e.*, collapsing all the consecutive state transitions into one.

What we do not formalize, however, is how a parser determines which operations it executes. This is by design: our system's primary selling point is that it can be extended with new parsers, as long as this implementation stays within the boundaries of the system's rules.

Something that can be said about the invocation of a parser, however, is that a parser will need to apply the transformation returned by each of its operations on the state, yielding a new state. It needs to do this so that it may determine what the next operations are — if the parser's implementation is context-sensitive; or in order to pass the state to a sub-parser invocation (which in turn, needs it for the same reasons).

How does all this translate in Autumn? Clearly it is not practical to represent the state as a sequence of primitive changes — assuming we want to consult the state to make parsing decisions. In Autumn, the state is simply encoded as regular program state (*i.e.*, part of the program's object graph — the value of certain fields). In this explanation we will stick with "state" to refer to the *parse state* — the state that can be

modified and read by parse, and that we may also call *context*.[11]

One issue here is that the program state is unique (there is only one copy of it), yet parsers must sometime execute speculatively — leading to backtracking. This is not an issue in the formalization, because the program state is reified as a single immutable value (a list of changes). Changes to the state are represented by transformations (state transitions) returned by primitive operations. We cannot adopt that mode of operation in Autumn, as it would run counter to context transparency (Section 5.3).

A second difficulty is that having a single copy of the state does not make it possible to implement snapshots, which are further required for the *diff*, *restore* and *merge* primitive operations. In fact, *snapshot* and *merge* can be seen as a way for parsers to perform backtracking manually, even in cases where backtracking would not naturally occur — for instance, they can be used to undo the state changes incurred by a successful parser invocation. This is for instance the case in longest-match parser presented in Section 5.6.4.

To enable backtracking, snapshotting and diffing, we thus need to represent the context as a series of changes from which the program state representation can be re-created. We already hinted at the solution earlier: each change will be associated with an equivalent *reverse* change: an *undo action*. These pairs are kept in a stack-like data structure named `Log`, which is held within the `Parse` instance. The log is kept in sync with the program state: every change pushed to the log is immediately applied onto the program state.

We already said that parsers must actually apply transformations to the state in order to execute, and that they cannot return transformations as that would break context transparency. Instead, parsers push changes to the log directly. In the formalization, we define a parser as a function from its input state to a sequence of transformations, which we said correspond to its execution trace. In practice, we do not want the execution of a parser to return a description of what it would do, we want the parser to actually do it.

[11] Of course, not all program state is *parse state*. For instance, the Autumn framework has its own state, and is likely to be used in a program that has other functionalities than parsing.

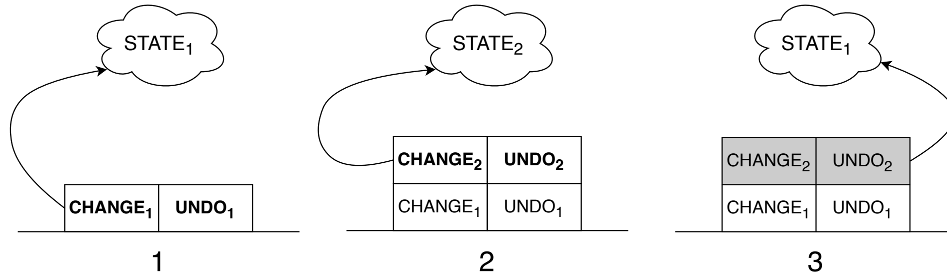


Figure 5.1: Illustration of how two *applyChange* and one *restore* operation affect the log data structure and the program state.

In fact, returning the execution trace was a formalization trick that allowed us to avoid encoding backtracking explicitly into the formalism: transformations that occur during the execution of failed parsers ultimately disappear — as there is no state mutation, nothing needs to be undone. In the implementation however, parsers do mutate the state, hence to enable backtracking it must be possible to undo state changes. This is why every change is assorted with an undo action.

Figure 5.1 provides a visual illustration of how three operations affect the log data structure and the program state. Each row above one of the numbers represents a *side effect*[12] made of both a change and the corresponding undo action. During the first operation (1), a side effect (bolded) is pushed onto the log along with its corresponding undo action. The change is immediately applied to the program state to yield $STATE_1$. The second operation (2) depicts the same process for a second side effect, taking the program state to $STATE_2$. Finally, the third operation (3) is a *restore* operation, causing us to backtrack over the second side effect. Therefore, we apply its undo action in order to move the program state back to $STATE_1$. The side effect is grayed because it will be popped from the stack after the undo action is applied.

The first listing in Figure 5.2 shows how a parser might enact a change to the context. We access the `Log` instance via `parse.log` and call the

[12]The term *side effect* roughly corresponds to a "change" in the formalization, but in the context of the implementation, we use to refer specifically to the pair made by both a change function and its corresponding undo function. Using the term *side effect* emphasizes the stateful nature of the changes and their peculiar status with regard to usual parser semantics (which can be formalized in a functional way — though the implementation is not functional), requiring them to be managed specially.

```
1  public class MyParser extends Parser {
2      // ...
3      @Override protected doparse (Parse parse) {
4          // ...
5          parse.log.apply(() -> {
6              counter_state.data(parse).x += 1;
7              return () -> counter_state.data(parse).x -= 1;
8          });
9      }
10 }
```

```
1  parse.log.apply(() -> {
2      boolean removed = set_state.data(parse).remove(some_item);
3      return () -> if (removed) set_state.data(parse).add(some_item);
4  });
```

Figure 5.2: Example of how a parser can push a reversible state change to the log. The second listing shows a variant where the undo action captures data from the change.

`apply` method on it. This method takes a parameter of type `SideEffect`. A `SideEffect` is a function that returns another function — namely the undo function — and should apply changes to the program state.

The `apply` method immediately runs the `SideEffect` function, collecting the undo function, and saving both the original function and the undo function as a `SideEffect.Applied` in the log.[13]

The `counter_state.data(parse)` code fragment is a pattern that lets us access data stored in the `Parse` object — which avoids storing context in parsers themselves, which in turns lets use reuse parsers in multiple concurrent parses. We will cover its implementation in more detail in Section 5.6.3.

This example is then very simple: the change accesses a plain Java object that has an `x` integer field, and increments it by 1, while the reverse change naturally decrements it by 1.

The setup may seem peculiar: why not provide the regular change and

[13]Saving the original `SideEffect` function is necessary in order to perform *diff* and *merge* actions.

<table>
<tr><th>Operation</th><th>Encoding</th></tr>
<tr><td>applyChange</td><td><pre>parse.log.apply(() -> {
   a_function(a_state.data(parse));
   return () ->
      an_undo_function(a_state.data(parse));
});</pre></td></tr>
<tr><td>call</td><td><pre>some_parser.parse(parse)</pre></td></tr>
<tr><td>snapshot</td><td><pre>int pos0 = parse.pos;
int log0 = parse.log.size();</pre></td></tr>
<tr><td>restore</td><td><pre>parse.pos = pos0;
parse.log.rollback(log0);</pre></td></tr>
<tr><td>diff</td><td><pre>List<SideEffect> delta =
   parse.log.delta(log0);</pre></td></tr>
<tr><td>merge</td><td><pre>parse.log.apply(delta);</pre></td></tr>
</table>

Table 5.1: List of our primitive state manipulation operations, along with an example of their usual code encoding in Autumn.

the reverse change separately? It is because the nature of the change may depend on the state at the time when the change is applied. Therefore the reverse change may need to capture part of the state. This is illustrated in the second listing of Figure 5.2: the change attempts to remove a given item from a set. In this case, the undo action captures whether the removal was successful (the `removed` boolean), as well as the item to add if that is the case (`some_item`).

As you may have guessed, the `Log#apply` method corresponds to our *applyChange* primitive operation. The next section will deal with how we can express the other primitive operations in terms of the log.

5.6.2 Primitive Operations and the Log

The goal of this section is to further explain how the primitive operations from Section 5.5.3 map to implementation concepts. A summary of how these operations are encoded into Autumn code is also shown in Table 5.1.

applyChange and *call*

As we saw in the last section, the *applyChange* primitive operation corresponds to the `Log#apply` method, which pushes an applied side

effect (a pair of a state-mutating function with its corresponding undo function) onto our `Log` data structure.

The *call* primitive operation is also simple: it simply maps to parser invocation in the implementation. The invoked parser will then interact with the `Log` instance on its own. The implementation does however have to take care of backtracking — as a failed parser should leave the log unchanged (cf. Section 5.5.4). This is actually implemented using the *snapshot* and *restore* operations.

snapshot* and *restore

The *snapshot* and *restore* operations are more involved. It should first be made clear that, in practice, we're not interested in keeping snapshots around forever.[14] In fact, we're only interested to use them in a way that enables backtracking, whether the automatic kind, or the kind that is forced by a parser. As such, the fundamental restriction is that *operations that make use of a snapshot (*restore *or* diff*) should occur during the execution of the same parser that ran the* snapshot *operation which created the snapshot.* It is the responsibility of the user to enforce this.

Given this restriction, it then suffices to take the size of the log to create a snapshot. To restore this "snapshot", it suffices to pop items off the log and apply their undo action until the size is restored to the saved one. The *snapshot* and *restore* capabilities are implemented as the `Log#size()` and `Log#rollback(int target_size)` methods.

We note that *restore* is the only operation that is able to shrink the size of log,[15] and that we are not at risk to shrink the log further than its snapshot size — even if other *restore* operations are nested in between *snapshot* and *restore*, they will not be able to, given the restriction outlined above.

As for how backtracking is implemented, it is very simple. Recall from Section 3.2 that we invoke a parser by calling its `parse` method, which in turns calls the `doparse` method implemented by the user. We said earlier that `parse` took care of some bookkeeping. One of the things it does is create a snapshot, and restore it after calling `doparse` in case it

[14] A fact admittedly not encoded in the formalization, but see shortly for how to reclaim this capability.

[15] Even parser invocations cannot, given the restriction outlined above on snapshot usage.

fails (returns `false`).

If we really needed to lift the restrictions on snapshots lifetime, we could define a snapshot as the whole content of the log. Then one could perform a *restore* by first undoing the entirety of the current log and then reapplying all the changes (side effects) in the saved log. This could further be optimized by avoiding to undo the changes that appear in a shared prefix between the two logs. In fact, this is perfectly doable in our implementation (even for the parser author), it just is not necessary for any use case that we encountered so far — and it is vastly less efficient than the restricted log size approach.

diff and *merge*

diff takes a snapshot and a state and returns a delta between the two. The formalization defines all three of these things as sequences of changes. In the implementation, the state will always be the current state represented by the current state of the log, while the snapshot is an integer inferior or equal to the current size of the log (recall the above restriction: the *snapshot* operation must have occured during the same parser invocation as the *diff*). Given that, a delta is simply the slice of `SideEffect` instances (we do not need the undo actions) that sits on top of the log above the snapshot size. Such a delta can be obtained via the `Log#delta(int snapshot_size)` method.

Unlike a snapshot, a delta may freely escape the purview of the parser that created it. This is notably the case for memoization (cf. Section 6.1.4), where the delta is part of the information memoized about a parser invocation, and is consequently entered into a memo table. Memoization is a recurrent use of *diff* and *merge*, including more local forms of memoizations — such as the one used in our `left_recursive` combinator (cf. Section 4.5.2), the optimization built into `right_fold` (cf. Section 4.8.3) or the `longest` combinator (cf. Section 5.6.4).

Regarding *merge*, things are very simple: it simply consists of applying the listed `SideEffect` onto the log. This can be achieve via the method `Log#apply(List<SideEffect> delta)`. We note that it is not necessary to merge a delta onto the same log that was used to create it (if that were the case, it would yield a poor form of memoization). A delta may even be merged multiple times (though we do not have a use-case where that is actually needed).

5.6.3 Operationalization and Usability Concerns

This section gives some more details on a couple of small operationalization and usability concerns relevant to the implementation, namely (1) maintaining per-parse state, (2) reversible data structures, (3) the possibility of nested side effects, (4) the special status of the input position as parse state, and (5) the possibility of memoization in the presence of context.

Per-Parse State

You may recall from Figure 5.2 that we used the `some_state.data(parse)` construction to access a Java object that was part of our context. We now explain what this means.

The problem this solves is fairly simple: we want the context (the *parse state*) to be linked to a particular parse (*i.e.*, a `Parse` instance) — this enables parsers to be freely shared between multiple concurrent parses. Yet, the context has to be accessed from the parsers, which needs to have a way to reference the correct state. We also want to avoid conflicts between independently developed parsers and associated data objects.

We resolve this conundrum by letting parsers manipulate a `ParseState<Data>` object, where `Data` is the type of the context object (it could be anything). To construct a `ParseState` one needs to supply two things: a key that will uniquely identify the parse state, and a function that creates the initial instance of `Data`.

So by filling in the code from Figure 5.2, we get the code in Figure 5.3. The key is `MyParser.class` — a good choice because it will never clash with another key used by another parser, and it signals clearly that the same `IntHolder` instance is going to be used by all the instances of `MyParser`. Another common choice is to use the instance itself (`this`) as key — in the case we need one instance of the data object per parser instance.

Internally, the `Parse` instance is equipped with a map from key to objects. Calling `some_state.data(parse)` looks up the key in that map and returns the associated item. If the key is absent, the data object is created using the supplied constructor and entered into the map.

This is further optimized by caching one instance of the data object (*i.e.*, *the* instance for a single parse) in the `ParseState` instance — ensuring

```
public class MyParser extends Parser
{
    static class IntHolder {
        public int x;
    }

    private ParseState<IntHolder> some_state
        = new ParseState(MyParser.class, IntHolder::new);

    // ...

    @Override protected doparse (Parse parse) {
        // ...
        parse.log.apply(() -> {
            some_state.data(parse).x += 1;
            return () -> some_state.data(parse).x -= 1;
        });
    }
}
```

Figure 5.3: An expansion of Figure 5.2 showing the setup used to get per-parse context.

that if only a single parse is ongoing at any one time, we will not incur the overhead of a hash table lookup at all.

Reversible Data Structures

The side effect mechanism is relatively lightweight. By only requiring that undo actions be provided for changes that are actually made, we limit the overhead placed on the user.

Nevertheless, it would be even easier to get access to data structures which automatically push their changes onto the log along with the appropriate undo action — hence hiding the side effect mechanism from the user.

We have actually already seen uses of one such structure: the value stack used to build abstract syntax trees (Section 3.3). Indeed, way back in Section 3.3.2 we introduced the idea that the value stack is a form of parse state. It is maybe a stretch to call it *context*, because parsing decisions are normally not made based on AST nodes (though nothing precludes this in Autumn). However, changes to the value stack are a

side effect, and therefore need to be undone upon backtracking.

The value stack actually has type `SideEffectingArrayStack`, which is also made available to users. This is precisely a reversible data structure that automatically updates the log whenever it is updated.

It would be fairly easy to write more of these data structures. However, it is not quite clear how beneficial this endeavour would be when a side effect does not consist of a single change to a single of these data structures. For instance, undoing multiple such changes at once might be much more effective using an undo action.

Nested Side Effects

One may wonder what occurs when the log is modified within a side effect. The good news is that this actually works by construction. The side effect will be ran before it is added to the stack — this is necessary because it will return the undo action needed to construct the `SideEffect.Applied` instance pushed onto the log. Any modification to the log within the side effect, will thus be applied onto the log first, and consequently be undone after the outer side effect.

This enables relatively easy side effect composition, should it be required. A good example is using the reversible data structures (introduced in the last section) inside a side effect.

On the other hand, modifying the log within an undo action is unsound and should not be done.

The Input Position as Parse State

As far as our formalization goes, the input position should just be a regular instance of parse state. In practice, our implementation always manipulates the input position separately when it comes to making snapshots, diffs, restoring and merging.

The reason is quite mundane: the input position is manipulated **a lot**: the majority of parsers update it. If each of these manipulations yielded the creation of multiple objects and the modification of the log, it would slow down the parse a great deal.

In addition, it makes sense to give the input position a special status. For simple grammars and custom parsers, the user does not have to concern

himself with the context (beyond the AST-building logic — which is cleanly encapsulated via a reversible data structure). On the other hand, the input position is a key notion.

Memoization in Context

While a full explanation of Autumn's memoization facilities must wait for Section 6.1.4, one may wonder how memoization may work in the presence of context-sensitivity. Indeed, memoization relies on the *single parse rule* to ensure that the invocation of a parser at a given position always yields the same result. Context-sensitivity voids this guarantee by making it possible to have multiple invocations with different context at the same position.

Autumn's solution to this issue is to outfit each memoizing parser that needs to worry about context with a *context extractor*: a function that returns an object encapsulating the relevant context at the time of invocation. This *context object* is then memoized along with the result of the invocation. When we try to use a memoized result, we re-extract a new context object from the current context, and compare it to the memoized one.

Using the context object has one unfortunate consequence however: it breaks our *context transparency* property (Section 5.3). If the grammar is refactored, such that a parser reachable through a memoized parser depends on context that is not captured, one risks recalling a wrong memoized result. As such, memoization should only be applied as a last optimization step when writing a grammar, and should be applied on low-level parsers (*i.e.*, parsers through which few other parsers are reachable) whenever possible. As we will see in Section 6.1, pervasive memoization is not necessary (or even detrimental) for good performance and frequently-invoked parsers logically tend to be low-level parsers.

As we will see, Autumn allows specifying different interchangeable memoization strategies — one can change the memoization strategy without changing the rest of the grammar. Given the existence of a context object, multiple strategies are conceivable. We can memoize all *(input position, context object)* pairs we encounter for a given parser, or just memoize the last context object for a given input position, while making certain we do validate the memoized context object. The two memoization strategies built into Autumn do produce distinct entries when encountering different contexts at the same input position for a given parser.

5.6.4 Examples

We now give a couple of examples of how the context-sensitive facilities Autumn provides through its side effect log are used in practice. We first show how to use them to actually define some context-sensitive features, and we then turn to usage within more general parser combinators.

A Simple XML Language

We will first look at a full full-blown example of context-sensitive parsing implemented in Autumn. The listing below shows a grammar and all relevant associated code necessary to define a simple XML language.

Our language is made up of paired opening and closing tags, *e.g.*, `<mytag>` and `</mytag>`. A tag name can be any string of alphanumeric characters starting with a letter. Inside a tag there can be other tags, as well as arbitrary text that does not contain the `<` character.

Our language is of course simplistic. Compared to the real XML language it does not contain attributes, any kind of comment, or character escapes — to cite only a few things. It is nevertheless sufficient to demonstrate a context-sensitive parsing capability, namely that it ensures that every opening tag is paired with the corresponding closing tag and sets an appropriate error message if that is not the case.

Let's have a look at the listing, after which we will give more details.

```
1  // ... (imports from norswap.autumn and java.util)
2
3  public final class SimpleXML extends DSL
4  {
5      public final class Tag {
6          public final List<?> contents;
7          public Tag (List<?> contents) {
8              this.contents = contents;
9          }
10     }
11
12     private final ParseState<ArrayDeque<String>> tag_stack
13         = new ParseState<>(SimpleXML.class, ArrayDeque::new);
14
15     public rule identifier =
16         seq(alpha, alphanum.at_least(0));
17
```

```
18      public rule open_identifier =
19          identifier.collect()
20          .action_with_string((p, xs, str) -> p.log.apply(() -> {
21              tag_stack.data(p).push(str);
22              return () -> tag_stack.data(p).pop();
23          }));
24
25      public rule close_identifier =
26          rule(new CloseTag(identifier.get()));
27
28      public final class CloseTag extends AbstractWrapper
29      {
30          public CloseTag (Parser identifier) {
31              super("close_tag", identifier);
32          }
33
34          @Override protected boolean doparse (Parse parse)
35          {
36              int pos0 = parse.pos;
37              if (!child.parse(parse))
38                  return false;
39
40              String close_tag =
41                  parse.string.substring(pos0, parse.pos);
42              ArrayDeque<String> tstack = tag_stack.data(parse);
43              String open_tag = tstack.peek();
44
45              if (open_tag == null) {
46                  parse.set_error_message(
47                      "Closing␣tag␣without␣matching␣opening␣tag:␣</"
48                          + close_tag + ">");
49                  return false;
50              }
51
52              if (!close_tag.equals(open_tag)) {
53                  parse.set_error_message(
54                      "Mismatched␣opening␣and␣closing␣tag:␣<"
55                          + open_tag + ">␣and␣</" + close_tag + ">");
56                  return false;
57              }
58
59              tstack.pop();
60              return true;
61          }
62      }
63
64      public rule open_tag = seq("<", open_identifier, ">");
65      public rule close_tag = seq("</", close_identifier, ">");
```

```
66
67      public rule text =
68          cpred(c -> c != '<').at_least(1)
69          .collect().action_with_string((p, xs, str) -> {
70              p.stack.push(
71                  Arrays.stream(str.split("\n"))
72                      .map(String::trim)
73                      .collect(Collectors.joining("\n")));
74          });
75
76      public rule contents =
77          choice(lazy(() -> this.tag), text).at_least(0);
78
79      public rule tag =
80          seq(open_tag, contents, close_tag)
81          .push(xs -> new Tag(list(xs)));
82  }
```

Our AST for this language will consist of two types of objects: instances of `Tag` and plain `String` objects. Tags hold a list of their children, which can be other tags or strings (in the order in which they appear in the XML source).

The whole context of the grammar is held in a single `ParseState` instance called `tag_stack`. Since it is the only parse state being used, we use `SimpleXML.class` as a key.

You should be able to understand many of the grammar rules from the explanation we gave in Chapter 3, but there are a few novelties that we will detail. All the context-sensitive logic is concentrated in the `open_identifier` rule as well as in the `CloseTag` class. We will go over these in details shortly, but we first explain non-context-sensitive peculiarities.

In rule `text`, `cpred` is a character predicate that consumes a character that satisfies the given predicate (here, the character should be different from `<`. We could also have expressed the same thing by using the parser `seq(character('c').not(), any)`. We then use the `action_with_string` combinator, which gives us access to the matched text as the parameter `str`. The action creates a copy of the string without the leading and trailing whitespace of every line, and pushes it onto the value stack.

`alpha` and `alphanum` are built-in rules, respectively matching alphabetic and alphanumeric characters. You will notice we also forego using the special built-in rule `ws` (Section 3.1.2) in this grammar: all allowed

whitespace will be parsed as part of the `text` rule.

Regarding context-sensitivity, the first thing you will notice is the asymmetry between our two context-aware rules. `open_identifier` is not context-sensitive per se, it just writes the context — therefore we access the context (the tag stack) within the action passed to the `action_with_string` combinator.[16] `close_identifier`, on the other hand, needs to actually read the context to make a parsing decisions — that needs to happen within a parser's `doparse` implementation.

Regarding the logic, `open_identifier` simply retrieves the matched tag name (via the `str` parameter) and pushes it onto the tag stack. The undo action is simply to pop the name off the stack.[17] The `CloseTag` class extends `AbstractWrapper` which is an abstract class offered by Autumn to ease the definition[18] of parsers that can match the same thing as their single child, but are able to add additional restrictions — in this case, context-sensitive ones. The class compares the closing tag name to the opening tag name recorded on top of the stack, and sets an appropriate error message in case it encounters an orphaned or mismatched closing tag (the error message mechanism is explained in Section 6.4.2). Otherwise, it succeeds and pops the opening tag off the stack.

Finally, we note that Autumn also supplies a `ContextPredicate` parser class that takes a function from `Parse` to boolean as argument. These parsers can be built with the `context(...)` method and used to implement simple context checks. We could slightly shorten our example by using instead of defining the `CloseTag` class: we just need to push the tag onto the value stack, then access it from the context predicate and verify it is equal to the top of the tag stack — like we do in `doparse`.

```
1  @Override public boolean doparse (Parse parse)
2  {
3      int pos0 = parse.pos;
4      int log0 = parse.log.size();
5
6      int max_pos = pos0;
7      List<SideEffect> delta = null;
8
9      for (Parser child: children)
10     {
11         boolean success = child.parse(parse);
12         if (success) {
13             if (parse.pos > max_pos) {
14                 max_pos = parse.pos;
15                 delta = parse.log.delta(log0);
16             }
17
18             parse.pos = pos0;
19             parse.log.rollback(log0);
20         }
21     }
22
23     if (delta == null)
24         return false;
25
26     parse.pos = max_pos;
27     parse.log.apply(delta);
28     return true;
29 }
```

Figure 5.4: The implementation of the `doparse` method for Autumn's built-in `Longest` parser.

`Longest` Parser

Let's move onto an example of using the *snapshot*, *restore*, *diff* and *merge* operations in a parser. Our `Longest` parser is a modified choice parser that tries all of its children on the input and matches the same thing as the one that matches the most input. Accordingly, only the state modifications made by this longest-matching parser invocation should be borne out.

Figure 5.4 shows how the `Longest` parser's `doparse` method is implemented in Autumn.[19] As explained earlier, our snapshot is simply the size of the log assigned to `log0`. We try parsing each parser in turn, then reset the parse to its previous state by resetting the input position to its initial value[20] and performing a *restore* operation via `parse.log.rollback(log0)`. If the parser is the longest-matching so far, we also save the extent of the match as well as its side effect through a *diff* operation implemented by `parse.log.delta(log0)`. Finally, we *merge* the side effects of the longest-matching parser invocation through `parse.log.apply(delta)`.

Similar mechanisms are deployed in other parsers built into Autumn. This is notably the case of the `Memo` parser implementing memoization, which we describe in Section 6.1.4.

5.6.5 Alternatives

The implementation we present above is not the only possible way to operationalize the principles of principled stateful parsing.

[16] This combinator is somewhat analogous to `push` (cf. Section 3.3.1): it collects the objects pushed on the value stack by the wrapping parser and passes them to a user-supplied function. Compared to `push`, the function does not have a return value that will be automatically pushed onto the stack, and it takes an additional `str` parameter that holds the string matched by the wrapped parser.

[17] We could have used our `SideEffectingArrayStack` from last section here, but we chose to be a bit more explicit in this introductory example.

[18] By providing implementations of some of the `Parser` abstract methods and adequate default behaviours for the built-in parser visitors of Section 6.5.5. Abstract parsers are covered in Section 6.5.7.

[19] The full code can be found online in the Autumn source code repository [56], at path `src/norswap/autumn/parsers/Longest.java`.

[20] We covered the special status of the input position at the end of Section 5.6.3.

Indeed, in our 2016 paper [58], we outlined a completely different method of implementation. The key difference in that version is that snapshots make actual copies of the context. Of course, that is relatively costly in general, so we made sure to encourage the use of efficient *immutable* data structures.

Immutable data structures can be copied by reference. On the other hand, the costs shift to "mutations" of the data structure: instead of mutating the structure in place, a fresh modified copy needs to be created, which is expensive. To counteract these costs, efficient immutable data structures employ *structural sharing* to avoid copying data as much as possible — the canonical example being the venerable singly linked list. The old version of Autumn supplied with the paper also provided a Hash Array Map Trie (HAMT) [7] based on the recent high-performance improvements proposed by Steindorfer and Vinju [88].

In that old version, we required context to be held in classes implementing an interface specifying how the primitive operations of Section 5.5.3 should work on the given class. The users were thus responsible for creating their own delta objects and snapshots, as well as handling the *merge* and *restore* logic. Quite obviously, this had a whole lot more overhead than the current approach, both conceptually and in terms of implementation effort. We did supply built-in data structures to alleviate some of that effort (*e.g.*, the aforementioned HAMT implementation). Interestingly, note that this interface can be implemented in terms of a log like that present in the newer Autumn versions.

Doubtless, there are even more ways to implement a system that implements our primitive operations and satisfy our requirements.

A particularly interesting avenue of inquiry is that of reversible computation: instead of specifying undo actions manually, the system would be able to automatically determine the necessary step to undo a change. This method is notably used by the Colm [92] language — which we discussed in Section 5.2.1. Colm is its own language and runs on top of a custom virtual machine, which is able to record state changes as they occur and build up a procedure to reverse them.

This approach would be difficult to emulate in Autumn, as it is implemented as an embedded Java domain specific language (DSL). We could get something similar by restricting state changes to reversible data structures — though there is no way of checking that the user actually

respects this restriction,[21] in the same way that we cannot verify if the user-supplied undo actions actually reverse the changes that were made.

5.7 Conclusion

In this chapter, we proposed an approach to tackle the problem of context-sensitive parsing. Our solution, unlike existing ones, possesses the property of *context transparency*: grammatical constructs are unaware of the context shared between their ancestors and their descendants, making it easier to write, evolve and compose context-sensitive grammars.

We proceeded in two parts. First, we allowed parsers to manipulate a mutable data store, so as to enable context-sensitivity through *recall*. Second, we required parsers to behave transactionally: a parser must either succeed, or fail and leave the state unaltered. This transactional discipline, which we call *principled stateful parsing*, prevents parsing mechanisms such as backtracking and memoization to break the guarantee of context transparency.

To enforce the principled stateful parsing discipline, we supplied formally specified state manipulation operations: *applyChange* applies a change to the state; *call* corresponds to a parser invocation, which will induce further operations; *snapshot* creates a “snapshot” of the state, which can be restored using the *restore* operation; *diff* enables creating a “delta” (a patch) between a snapshot and the current state, which can be (re-)applied later using the *merge* operation.

We implemented the approach in our Autumn parsing tool by keeping a log of reversible changes to the parsing state, and showed how it can be used in practice to define the syntax of context-sensitive language features, as well as how the approach can be made compatible with existing facilities such as memoization and longest-match parsing. We underline the flexibility enabled by the approach, while maintaining the amount of boilerplate and conceptual overhead low.

[21] Besides full-blown static source code analysis, which would defeat some of the advantages of using an embedded DSL.

Chapter 6

Engineering Aspects

In Chapter 1 we emphasized our wish to develop a pragmatic parsing approach, and to demonstrate this approach in a practical tool — our Autumn parsing framework. We positioned our approach as a trade-off between the simplicity and declarativeness of grammarware on the one hand, and the flexibility of ad-hoc parsing on the other hand.

So far, we have mostly focused on issues of expressiveness, and the solutions afforded by the ability to write custom parsers. Chapter 3 introduced these capabilities along with other basic concepts of Autumn. Chapter 4 used them to implement different left-recursion and left-associativity handling strategies, proving the power of the mechanism and that the approach does not need to be beholden to the limitations of the PEG formalism. Chapter 5 extends our approach with a new capability: free-floating context and means to manipulate it safely and in a well-circumscribed way (*i.e.*, *context-transparently*). It also showed how to use this capability in practice.

While writing custom parsers is the key element of our approach and the bedrock of our claim to flexibility, it is not by itself sufficient to satisfy the ambitious goals we have set in Chapter 1. Nor do they all relate to expressiveness.

Therefore, this chapter is concerned with aspects of parsing that go beyond expressiveness, and in particular focuses on aspects of a less *algorithmic* nature than those discussed before. We will still discuss a number of custom parsers, but with the goal of using them towards more

pragmatic endeavours, such as improving performance and bettering the user experience.

We have gathered together a number of concerns under the umbrella term of *"engineering aspects"*, to emphasize that they relate to implementation and code architecture — though that is by no means the extent to which these concerns matter. These aspects are diverse:

- Section 6.1 discusses the ever-important performance aspect. We recapitulate previous comments on performance, present Autumn's support for memoization, and make a few observations on the nature of top-down combinator performance.
- Section 6.2 showcases performance measurements made with Autumn and shows how it compares to some state-of-the-art parsing tools.
- Section 6.3 shows how a scannerless combinator framework like Autumn can be used to simulate lexical analysis — a practice that simplifies grammars, helps a lot with performance and can improve error reporting.
- Section 6.4 discusses error reporting and recovery, outlining Autumn's capabilities and investigating possible advanced error-reporting and error-recovery strategies that can be implemented using Autumn's existing features.
- Section 6.5 discusses Autumn's support for grammar reification. In particular, we present our architecture for grammar traversal as well as creating operations that can be specialized for every kind of parser, tackling a modified form of the *expression problem* along the way. These capabilities are used to perform key static grammar analyses in Autumn, as well as support grammar composition.
- Section 6.6 presents Autumn's support for debugging and tracing a parse's execution, two capabilities which are supremely useful in the process of writing correct and efficient grammars/parsers.
- Section 6.7 discusses the possibility of grammar composition in Autumn as well as some related challenges.

These aspects are not to be taken lightly if one wishes to produce a useful

parsing tool — and it is consequently important that the underlying parsing approach should be compatible with them.

Unfortunately, some of these aspects, though they have obviously been "researched" (after all, many parsing tools have some of these capabilities) are under-discussed in the academic literature (in fact, in the literature in general). We can ourselves attest that finding quality discussions of the implementation side of these aspects is arduous when it is possible at all — all too often, the only point of reference is source code. We therefore consider our investigations of these aspects in the context of Autumn's implementation, and the present discussion of them, as important contributions of this thesis.

6.1 Performance Considerations

In general, *simple PEG parsing* (cf. Section 2.4.4) has worst-case exponential time complexity, while *packrat parsing* has linear time complexity, but superior (linear) memory requirements (cf. Section 2.4.5). Just like before, we use "PEG" here as an umbrella term for both classical PEG parsing and top-down combinator parsing with user-defined custom parsers.[1]

We've argued before that these complexities may be less relevant than they seem, as most programming language grammars are nearly deterministic (cf. shaded box on page page 32). A relatively small lookahead is generally enough to determine the right rule to apply, and the amount of backtracking should be small even with simple PEG parsing. Section 6.1.1 tries to give some intuition as to why this is the case. But because a grammar is not exponential does not mean it cannot be inefficient. This section explores the nature of these inefficiencies, and potential remedies.

Section 6.1.3 looks at a common grammatical idiom that causes pathological performance for non-memoizing parsers. Section 6.1.3 explains why we can't just use packrat parsing to memoize the whole grammar and forget about performance. Nevertheless, targeted memoization can still be useful, and Section 6.1.4 presents Autumn's memoization support. All topics from this section will be further illustrated in the performance

[1] In principle, one cannot put any theoretical bound on combinator parsing, as user-defined parsers can be arbitrarily inefficient. We obviously assume common sense in our discussion (as there are no good reasons to write exponential (or worse) parsers), and will be careful to point out potential performance pitfalls.

comparison of Section 6.2.

6.1.1 The Conspicuous Absence of Exponentiality

We'd like to emphasize just how hard it is for PEG to actually devolve into exponential behaviour in practice.

It's hard to even come up with an exponential PEG grammar. An example would be $S \to aSb \mid aSa \mid \epsilon$. This grammar matches strings whose first half is made up entirely of a's, while its second half can be a mix a's and b's.

To get exponential behaviour in PEG, the pattern is that there should be a rule that recursively calls itself multiple time at a further input position. If the amount of input the rule can match is unbounded, worst-case exponential behaviour ensues. This is the case for our rule S above: when called at input position x, it might end up being called at position $x + 1$ twice — as indeed it will for every position, when the input is a string consisting of only a's.

More pragmatically — as the above rule is unlikely to be written in practice — we personally never encountered a single instance of exponential behaviour, neither in our grammars or that of others. Becket and Somogyi also note "exponential behaviour just doesn't seem to happen in practice anyway" [8].

6.1.2 Inefficient Idioms in Simple PEG Parsing

While exponential run-times are inexistent in practice, language determinism does not necessarily preclude "bad" (but non-exponential) backtracking idioms in the grammar.

In particular, we managed to find one idiom that showcases such inefficiencies. Worryingly, this idiom is in widespread use for CFG grammars, and as such is often used in PEG grammars too. The idiom is a way to encode precedence in binary expressions using different rule names, shown in Figure 6.1.

Grammars of the kind shown in Figure 6.1 are parsed inefficiently by non-memoizing parsers. Consider a grammar for infix binary expressions with L levels of precedence and P operators per level of precedence. In our example, $L = P = 2$. This grammar will parse a simple number (*e.g.*, `'42'`) in $O((P+1)^L)$ expression invocations. In our example, S will cause

$$
\begin{aligned}
S &\rightarrow P \text{ `+' } S \mid P \text{ `−' } S \mid P \\
P &\rightarrow N \text{ `*' } P \mid N \text{ `/' } P \mid N \\
N &\rightarrow [0-9]+
\end{aligned}
$$

Figure 6.1: A PEG grammar for simple arithmetic expressions that exhibits an undesirable backtracking pattern.

P to be called thrice at the same position, and for each of its invocations P, will cause N to be called thrice as well, for a total of 9 invocations of N. For real languages like C or Java, this adds up to thousands of invocations to parse a simple number. The complexity is somewhat amortized for longer expressions, but the cost remains prohibitively high.

We can also see why the behaviour isn't exponential in the input size: it is fixed by constants (P and L) that depend on the grammar, not on the input size.

This should feel familiar. Indeed, the grammar from Figure 6.1 is the same as the *right-associative layered encoding* in Figure 4.2a of Section 4.2.

But the issue is also reminiscent of the performance issues with the transparent left-recursion combinator (Section 4.5.3) and with the right-folding combinator (Section 4.8.3). In fact, all three issues share the same failure mode: they invoke the parser for the higher-precedence level multiple times at the same position (once for each operator at the current precedence level until a match is found).

The good news is that we have already established that the solution to these performance problems is to use the expression family combinator of Section 4.9, which does not exhibit this bad pattern.

We should also note that this is the only such pathological pattern we ever encountered while writing PEG-style grammars, nor have we ever seen such patterns reported elsewhere.

6.1.3 Packrat Parsing

Packrat parsing (cf. Section 2.4.5) is a technique that builds upon the simple top-down recursive-descent PEG parsing strategy, by adding memoization of parse results on *(input position, parsing expression)* pairs.

Since packrat parsing does solve the issues outlined in the previous section, and has a much lower (linear) time complexity bound, it is only natural to wonder why we don't use this technique and call it a day.

Unfortunately, it turns out that packrat parsing is, in practice, often *less* efficient than simple PEG parsing. We quickly review two studies to that effect.

Becket & Somogyi 2008

In a study done by Ralph Becket and Zoltan Somogyi [8], they found out that — amongst different memoization strategies — memoizing all grammar rules was always the worst possible strategy in terms of parse time when using their grammar for the Java programming language. When compared to the best memoization strategy found, memoizing every rule was 300-700% slower, while memoizing nothing was only 0-30% slower.

These results must be tempered somewhat. First, they were obtained using a single grammar: that of Java, and a single tool: the Mecury programming language. Mecury is a programming language based on the *logic programming* paradigm (like Prolog). As such, it includes the possibility of defining Definite Clause Grammars (DCGs), as introduced in Section 2.1.3. The semantics of DCGs are that of CFGs, absent left-recursion. These are basically the same semantics as PEG with the addition of lateral backtracking (cf. Section 2.4.4). However, the authors port a CFG for Java to the DCG formalism by inserting cut operators after each choice alternative[2] (and modifying choices wherever necessary). This has the effect of making all choices ordered and preventing lateral backtracking, making the difference in semantics moot.

DCG rules can also have parameters, and those have to be memoized alongside the rule and input position. The authors note that this is costly, but it's unclear how much this affects the overall memoization costs. We do note that the vast majority of rules aren't parameterized, but by itself this gives no information on much they are called.

In summary, the experiment is valid despite the apparent difference in formalisms — but has still been conducted using only a single tool

[2] In reality, the authors use the `if ... then ... else ...` operator, but this operator can (and usually is) defined in terms of the cut operator. The cut operator prevents backtracking to other alternatives of a Prolog rule when encountered.

and grammar. We have reasons to suspect the memoization overhead might not be as bad for other tools: the fastest tool in the performance comparison of Section 6.2 is a packrat parser generator! Nevertheless, the benchmark will also show that for Autumn's Java grammar, memoization beyond the lexical layer doesn't help performance.

Redziejowski 2008

The next study [76] does not in fact show that packrat parsing is slower than simple PEG parsing, but hints in that direction, and helps explain why it might be the case.

In the study, Redziejowski demonstrates a PEG parser generator — called Mouse — that does not perform memoization. The author does not record actual (temporal) performance, but instead the number of function calls made during a parse (the number of such calls being proportional to the number of parsing expression invocations).

Redziejowski then goes on to show that on his corpus (about 10.000 source files comprising the J2SE 5.0 JDK source code), only 16.1% of all function calls are *repeated*, *i.e.*, called at the same input position than a previous call of the same function. This number further drops to 10% after he modifies the generated parser to perform a hash-table lookup on matched identifiers to see if they are not equivalent to reserved keywords, hence avoiding a large number of (repeated) function calls. This change also reduces the total number of function calls by about 20%.

In a further experiment, he shows that memoizing the last invocation of each function reduces the number of repeated calls to only 3.3%, and memoizing the two last invocations to 1.1%.

From the author's numbers we note that, interestingly, the total number of calls drops more or less proportionally to the reduction in repeated calls. So when the percentage of repeated calls drops from 16.1% to 3.3% (a 12.8% reduction), the total number of calls drops by 14.1%. This shows that the memoized calls don't tend to make too many function calls of their own. This is perhaps not very surprising, since the author reports that the average backtracking length is only 3.8 characters.

It is a shame that the author did not report run times, as that would have given a sense of how the memoization overhead measured up against the reduction in function calls. Nevertheless the result do align pretty neatly with those of Becket & Somogyi [8] and our own (Section 6.2).

Mouse [77] is one of the tools that we benchmarked in Section 6.2. As we shall see, the (non-optimized) grammar distributed by Redziejowski runs suspiciously slowly and we will speculate to the cause of this slowness in relationship with the experiment presented in this section.

6.1.4 Memoization in Autumn

Autumn does offer the possibility of memoizing parser results, but — given the results from the previous section — does not do so by default. This section explains Autumn's memoization facilities, which are built entirely on user-available features.

In Autumn, memoization is handled by a custom parser named `Memo`. This parser wraps a sub-parser whose result it will memoize. It also needs two other things: an instance of `Memoizer`[3] which will dictate the memoization strategy to use, as well as to provide the storage for the memo table; and — if the sub-parser is context-sensitive — an *extractor* function to isolate the relevant part of the context, to be memoized along with the results. We talked about the role of the extractor before in Section 5.6.3.

The basics of memoization have been covered during our discussion of packrat parsing in Section 2.4.5, but to summarize it briefly, we simply memoize parse results (success, amount of input matched, generated syntax trees) on a *(input position, parser)* key. Per the *single-parse rule* (Section 2.4.3), these results are guaranteed to be unique. The introduction of parse state and context-sensitivity changes things somewhat: syntax trees are now a form of parse state, and all changes to the parse state can be memoized via our *diff* operation (Section 5.6) Memoization must now also discriminate on the context, using the aforementioned context extractor (Section 5.6.3) — with the annoying drawback of breaking our context-transparency property (Section 5.3).

Memoization strategies are built by implementing the `Memoizer` interface. In particular, each memoizer must define how to memoize an entry (made out of the parse results listed previously, including a *delta* list of side effects) on the basis of an *(input position, parser, context object)* key. It must also define how these entries can be retrieved.

Autumn comes bundled with two memoizer implementations. The first,

[3]Which must be wrapped in `ParseState` to enable safe parser reuse, as explained in Section 5.6.3.

MemoTable, simply memoizes every entry it is passed. This strategy is equivalent to that employed by classical packrat parsers. The second memoizer, MemoCache, only memoizes a bounded amount of entries, evicting older entries as new entries are added. The number of entries to be kept can be customized. In practice, it seems MemoCache is generally superior: anecdotal evidence and common sense suggest that unless your grammar exhibits wild backtracking, a MemoTable will mostly drive memory consumption up without much performance benefits. Finally, users can implement their own memoizers.

In Section 5.6.3 we hinted at one possible strategy that is not implemented in Autumn: we could remove the context object from the memoization key, hence keeping only a single (or alternatively, a bounded amount of) memo entries per *(input position, parser)* pair.

There is also another axis on which to vary your memoization strategy: you can opt to outfit each different Memo parser with its own memoizer, or to share the same memoizer between all parsers (or some in-between compromise). For instance, memoizing every rule using a MemoCache of size 1 yields Redziejowski's memoization strategy, which we presented in Section 6.1.3.

Regarding the DSL, memoizing a parser is as simple as affixing the .memo() combinator — which memoizes the parser in its own MemoTable. Multiple overloads of the combinator are also available: it's possible to specify an integer — in which case a MemoCache with the given size is used instead — a context extractor, a specific memoizer to use, or any combination of the aforementioned. Here are two simple examples:

- `seq(a, b).memo(8)`
- `seq("</", identifier, ">")`
  `.memo(p -> XMLContext.data(p).peek())`

Autumn also offers another form of memoization in the form of lexical analysis simulation, which is covered in Section 6.3.

6.1.5 Megamorphic Call Sites

Autumn is a parser interpreter. It doesn't have a separate compilation or generation step. This is convenient, but we pay a performance penalty for it, most notably in the form of dispatch overheads at megamorphic call sites and missed optimization opportunities.

A megamorphic call site is a location in the code where a virtual method is invoked that has three or more actual invocation targets (*i.e.*, the actual method implementation that will be called). Megamorphic call sites are distinguished from monomorphic call sites (a single actual invocation target) and bimorphic call sites (two actual invocation targets).

We emphasize that *actual* invocation targets means targets called in practice. It doesn't matter how many implementations a virtual method has, if a given call site always goes to the same invocation target, then the call site is monomorphic. The nature of a call site is determined by the JVM's (Java Virtual Machine) tracing JIT (just in time) compiler, at run time. The JIT monitors the invocation targets (that's why it is a *tracing* JIT), then when enough data has been accumulated (and if the code is "hot" — meaning invoked frequently enough), it dynamically (re)compiles the code, taking into account the call site's perceived nature.

A call site's nature may change during execution. If a call site that was exclusively used with a single invocation target suddenly starts seeing another invocation target, its nature will change from monomorphic to bimorphic. For this reason, all call sites that cannot be proven to be non-virtual[4] must be guarded by a *type guard*, a check verifying that the method's receiver does indeed have the type we assumed.[5]

When the JIT decides to compile some code, different call sites will be handled differently with regards to optimization opportunities. Non-virtual method calls can be inlined by the JIT without risk. In the monomorphic and bimorphic cases, the JIT can choose to either only inline the method lookup in the class' virtual table (often referred to as *vtable*), or to inline the implementations resulting from that lookup. It must also include the aforementioned type guard. Inlining is never possible for megamorphic call sites.

Unfortunately, in Autumn, megamorphic call sites are all over the place: each time a parser invokes one of its sub-parsers by calling its `parse` method, that is most likely a megamorphic call site: most common combinators will be used with more than two different kinds of parsers

[4] Non-virtual call sites include calls to `static` methods and methods marked `final`, as well as calls to methods considered effectively final because they are never overriden in any loaded class.

[5] And if not, causes the compiled code to be discarded in favor of slower interpretation in the JVM, at least until the call site's nature can be ascertained again.

as children.

What are the costs? The article *"Too Fast, Too Megamorphic: what influences method call performance in Java?"* [104] gives us some insight. Without inlining, non-virtual and monomorphic calls have roughly the same performance. Inlining makes non-virtual calls about 3.3 times faster and monomorphic calls 1.87 times faster. The difference is due to the presence of the type guard in the monomorphic case. When inlined, bimorphic calls are as fast as monomorphic calls.[6] Megamorphic calls are about 2.75 times slower than (inlined) monomorphic calls.

Of course, benchmarking is an art. There are countless details to be taken into account. The article *"The Black Magic of (Java) Method Dispatch"* [84] written by JVM performance expert Alexei Shipilëv goes into much more details on the performance of method dispatch in Java. Shipilëv reviewed the previously cited article, which lends credence to the fact that the figures are not wildly inaccurate.

Nevertheless, it must be said that in reality, method implementations are not empty. When inlining is performed, it enables additional optimization opportunities and it is the loss of these optimizations that are responsible for the real cost of megamorphic call sites, far more than the already costly tripling in method invocation overhead.

The effect of megamorphic call sites on parser combinator performance has been observed in practice. Using Scala macros, Béguet and Jonnalagedda [9] were able to construct a combinator framework that achieved an order of magnitude improvement in performance over the standard Scala parser combinator library. The method is disarmingly simple: global inlining of every combinator implementation (excepted when recursion prevents it). This is an excellent illustration of the performance gains inlining can buy.

This also matches our own observation: a previous version of Autumn was written in the Kotlin language. In this version, parsers and combinators were defined not as objects but as functions, which used the `inline` keyword to enable pervasive inlining. We saw a 2.6x performance improvement from the previous version, it is also 1.6x faster than the

[6] This is likely due to a flaw in the benchmarking methodology: it warms up the VM using the two invocation targets, but only measures performance using a single target. In reality, we expect to see a slight performance hit due to the occasional branch mispredict that happens when we need to hit the second target.

current version using an equivalent benchmark.[7] Such a comparison should be taken with a grain of salt, since other aspects of the design underwent changes; but to us it is quite clear that inlining was the main driving force behind the performance difference.

When profiling Autumn, it is apparent that, beyond lexical analysis, there is no single piece of code or algorithm that dominates the run time: overhead is spread relatively evenly over the parsing part of the code base. Increased performance must therefore come from either squeezing performance on the shared code paths (*e.g.*, `Parser#parse`) or via a more structural elimination of the megamorphic overheads.

A potential solution: using Autumn's parser graph reification and traversal support, we could turn it into a parser generator, which would generate code free of megamorphic overheads. We discuss the idea further in Section 6.5.8.

6.2 Performance Comparison

In this section, we measure Autumn's performance by having it parse a corpus of Java source files and produce matching parse trees. We also subject other PEG parsing tools, as well as ANTLR to the same benchmark and compare the results.

Our benchmark corpus is the code of the Spring framework (version 5.1.8), comprising 6933 Java files, for a total 38MB size. The project is written in Java 8, and comprises 1 111 059 lines of Java code, among which 622 720 non-whitespace non-comment lines.

All measurements were taken on an otherwise idle 2013 MacBook Pro with a 2.3GHz Intel Core i7 processor, 4GB of RAM allocated to the Java heap (using the Oracle HotSpot JVM bundled in Oracle's JDK 12 distribution, unless specified otherwise), and an SSD drive.

The other evaluated parsing tools are *Rats!* [31] (part of *xtc*[8] version 2.4.0), a state of the art packrat PEG parser generator with many optimizations;

[7] We abandonned the `inline` design for the current version, because it didn't support parser reification, which we consider crucial (cf. Section 1.3).

[8] The eXTensible Compiler project.

Parboiled (version 1.3.1)[9], a popular Java/Scala PEG parser combinator library; *Mouse* [77] (version 1.9.2), a minimalistic PEG parser generator that does not allow memoization; and *ANTLR 4* [72] a popular and efficient state of the art CFG parser generator (cf. Section 2.1.3, version 4.7.2). Both Parboiled and Rats do not support syntax introduced in Java 8, and so are unable to parse 177 files from the corpus.

Why those? It is true that there are plenty of parsing tools to choose from, especially for PEG parsers. All the selected parsing tools are written in Java (and, where relevant, generate Java parsers), which enables a fair performance comparison. They also all offer a Java grammar written by the tool's author, which should guarantee that the tool is exploited to the maximum of its ability. Many parsing tools supply simpler example grammars (JSON is a popular example) but we believe that a good use case should at least include an extensive infix expression syntax and some lexical intricacies, since those seem to be some of the major performance difficulties in practice.

These tools also have some characteristics that make them relevant, and together they cover a lot of ground: ANTLR is the most popular Java parsing tool and is considered to be one of the fastest (almost-)general CFG parser available [72]. Rats is the fastest PEG parser available and has been the object of optimization research [31]. Parboiled is a relatively popular PEG parser, that, on the other hand, doesn't use memoization. Mouse doesn't memoize either, but was the object of a packrat parsing experiment we described in Section 6.1.3.

To evaluate Autumn, we used the Java grammar reproduced in Appendix A. In addition, we also benchmark an alternative version of the grammar, which uses a separate lexer (adapted from the OpenJDK lexer). In that case, the input is a list of token objects.

6.2.1 Run Times and Hot Spots

The first line of each row in Table 6.1 shows the time the parsing tools took to parse the whole corpus and generate matching trees.[10] The *Time (Single)* column reports the median of 10 task runs in separate VMs. The *Time (Iterated)* column reports the median of 10 task runs inside a

[9]We use Parboiled version 1 — version 2 is a Scala-only redesign that uses macros to avoid the pitfalls we describe in Section 6.1.5, and doesn't offer a Java grammar.

[10]Excepted for Mouse, whose grammar does not include tree-generating semantic actions.

Parser	Time (Single)	Time (Iterated)	Time (Graal)
	Spread	Mem (Median)	Mem (Peak)
Autumn	21.912s	21.837s	15.266s
	3.954s (2.118s)	6M	10M
Autumn + Lexer	6.844s	5.295s	4.119s
	0.206s (0.118s)	6M	9M
Mouse	147.074s	152.875s	152.922s
	13.664s (7.392s)	5M	5M
Parboiled 1	18.987s	17.233s	9.952s
	0.609s (0.340s)	6M	21M
Rats!	6.851s	2.705s	2.355s
	1.445s (1.425s)	5M	7M
ANTLR 4	10.541s	7.388s	5.298s
	0.162s (0.052s)	80M	82M

Table 6.1: Comparing the run times and memory footprints of Autumn to other PEG parsing tools, as well as ANTLR 4. Measurements done over 38MB of Java code.

single VM. The *Time (Graal)* column reports the median of 10 task runs inside a single VM, using the new Graal VM [107] (version 19) which supports advanced optimizations. The reported times do not include the VM boot time, nor the time required to assemble the parser combinators (when applicable), if those can be reused for multiple parses. It does however include the time needed to read the files from disk, as many of the tested parsers read from the file all along the parse using a buffered reader. Autumn, for its part, reads each file in memory before starting the parse,[11] as does Parboiled.

The *Spread* column indicates the difference between the slowest and the fasted iterated run, after excluding the first run (usually the slowest because the VM is just warming up and the parser isn't entirely JIT-optimized yet). The number between parentheses indicates the difference between the slowest and the median run.

The run time results tell a lot of interesting stories. First off, Mouse is very slow even though it has the advantages of not suffering from megamorphism issues (cf. Section 6.1.5), handling infix expressions *very* naively — parsing them as a big list of sub-expressions and operators, without regard for precedence and associativity,[12] and not even generating a parse tree. In our review of Mouse's packrat parsing experiment (Section 6.1.3), we noted that the author's experiment with memoization and optimized identifier/keyword discrimination allowed him to shave off about 20% of all function calls made by the parser.[13] Our theory is that these 20% of function calls account for much more than for 20% of the total run time, as they have to perform comparison to the actual input. As a Mouse parser is a graph of function calls without polymorphic indirection, function call overhead can be optimized away fairly aggressively.

This theory aligns nicely with our experience with Autumn. Using Autumn's profiling capabilities (cf. Section 6.6.4)), we noticed that

[11] This shouldn't be a problem, even when reading multi-gigabyte files, as long as the size of the Java heap has been set accordingly. It does however force us to wait for the whole file to be read before starting the parse, preventing us from aborting the parse early in the case of an error near the start of the input.

[12] Our own grammar uses the expression family parsers introduced in Section 4.9 in order to generate ASTs with the correct associativity.

[13] The author did not report the run time difference, and sadly did not distribute the optimized version of the grammar.

Autumn spends most of its time at the lexical level — even though our lexical analysis support (cf. Section 6.3) makes sure that we never repeat any work there! Simply finding the correct token for a given position is what ends up taking the most time. This is perhaps not surprising as our approach is still slightly naive: we run parsers for all tokens and then select the longest one, whereas in reality it is fairly easy to short-circuit this process (*e.g.*, a valid number literal is never a valid identifier or keyword). This longest-match approach also requires us to perform side effect application, diffing and re-application. Introducing a more performant lexical analysis support system is high-priority future work for Autumn.

The cost of lexical analysis is readily apparent when one compares the run times of our original Autumn Java grammar with that of our lexer-supported grammar (whose measurement does include the time needed to do the lexing). Run times are divided by a factor between 3 and 4. In that version, Autumn's performance is on par with that of Rats and ANTLR.

We further used tracing to identify other parsers to memoize, but it appears that, once the lexical layer was taken care of, memoizing the parsers that had the highest invocation count or ran the longest didn't affect performance (or affected them slightly negatively).

Interestingly, the ANTLR grammar repository [97] lists two Java grammars: one that sticks close to the grammar from the Java Language Specification, and a faster one. The faithful grammar was much too slow to include in this benchmark (it took almost 35 minutes for a single run over the corpus). The issue seems to be that the syntax of infix expressions is defined using the layered left-associative encoding (as shown in Figure 4.2c). This further goes to show that parsing infix expressions syntax is a challenge even for well-established and well-researched tools. The ANTLR grammar we benchmarked in this section replaces the layered encoding by a big choice expression with associativity annotations (Figure 4.2e) — a bit like our expression clusters (cf. Section 4.7), though the parsing algorithm is entirely different.

Parboiled manages admirable performance for a non-memoizing parser, though that is not entirely true — a couple rules are annotated with `@MemoMismatches` which memoizes failed invocations. This is notably the case for the rule matching identifiers and the rule matching keywords (used to prevent matching an identifier that uses a keyword's name). As noted

by Mouse's author and reported in Section 6.1.3, identifier/keyword discrimination is a performance hot spot. The annotation is used in a couple of other strategic locations, such as the rule that matches primitive type names (`int`, `float`, ...) and the rule that matches Java annotations. Parboiled matches infix expressions using the idiomatic encoding (Figure 4.2b).

The spread numbers are fairly unsurprising, except in the case of Rats!, where the numbers indicates that most of the spread sits above the median — *i.e.*, that the median run is much closer to the fastest run than the slowest run. The reason is that Rats! actually takes two runs to warm up entirely and get to run times that are very close to the median. One low-confidence hypothesis is that since Rats! memoizes most of its rules, the code paths for some parsers are much less exercised than in other tools (at most once per file), and so take longer to get optimized. We also note that the spread for all parsing tools is not significantly different when using the Graal VM.

6.2.2 Virtual Machine Effects

By comparing the different columns, we can tell that most parsing tools benefit from VM warmup. The effect is particularly striking for Rats!. Mouse actually gets slightly worse performance from warmup. This is probably due to its long runtime causing CPU overheating (as we were alerted by swooshing sounds of fans), which in turns causes CPU downclocking on MacBooks.[14]

All tools excepted Mouse also perform better under the Graal VM. Autumn gets between 22 and 30% run time reduction while Parboiled gets a whopping 42% reduction. We did expect a speedup, but not quite that large. What surprised us however was that even generated parser benefited, with ANTLR getting a 28% reduction and Rats! getting a more modest 12% reduction. We don't know which optimizations are responsible for the speedups. Making this determination would probably require a lengthy investigation, but it suffices to say there is not a single obvious candidate. From the Graal website[15]:

> The compiler of GraalVM provides performance advantages for

[14] Or more accurately, disables CPU overclocking once a set thermal limit is reached. We only realized this after making the measurements, but fortunately our measurements were not temporally contiguous, so they could not have polluted each other.

[15] https://www.graalvm.org/docs/why-graal/

> highly abstracted programs due to its ability to remove costly object allocations in many scenarios. [...] Better inlining and more aggressive speculative optimizations can lead to additional benefits for complex long-running applications.

The fact that Mouse doesn't benefit is perhaps not surprising: the generated code doesn't perform a lot of local object allocation and is free of polymorphic overheads that would be helped by more aggressive inlining.

6.2.3 Memory Footprint

The *Mem* columns in Table 6.1 show the memory usage of the parsing tools for our parsing task. The *Median* column shows the median heap size measured after a garbage collector activation[16] over the course of ten runs in the same VM. The *Peak* column shows the highest heap size measured in the same conditions. We don't hold to any objects in between runs. The reason we measured ten runs instead of just one is that garbage collector activations are non-deterministic, and so more run time gives us more data points to sample from.

The memory measurements were made using the OpenJDK 12 Hotspot VM. We also checked memory use under GraalVM[17], but found that the heap size kept unexplicably increasing across iterations. A potential explanation is that the GC algorithm never comes around to doing a full round of garbage collection. Whatever the reason, these numbers do not capture the information we seek, and so we do not report them.

Memory consumption is reasonable across parsing tools. ANTLR does consume an order of magnitude more memory than other tools, but 80M is not an unreasonable amount of memory in practice.

6.2.4 Discussion

Results show that Autumn's performance is on the same order as the fastest parsing tools benchmarked (which are among the fastest general parsing tools full stop) when lexical analysis is properly optimized, and well within an order of magnitude of them otherwise.

We note that our benchmark — using Java 8's grammar — seemingly

[16] Obtained via the `-Xlog:gc` flag.

[17] Using the equivalent `-XX:+PrintGC` flag.

doesn't involve context-sensitivity. While the grammar does not contain context-sensitive rules, it still uses a pervasive form of parse state in the form of the value stack used to build ASTs. Any modification to this data structure results in changes being pushed onto our `Log` data structure (cf. Section 5.6.3). In practice, we expect the `Log` activity due to AST construction to dwarf any other kind of context changes. So our benchmark does indeed prove the scalability of the underlying context-sensitive architecture. Context-sensitivity also impacts (but does not preclude) memoization — but given the experiments from Section 6.1.3 and our measurements, it seems clear memoization is not necessarily needed beyond the lexical layer in most languages.

There might still be optimization opportunities in Autumn, though we believe we have plucked all the low-hanging fruits. A more promising direction, if performance should be a concern, would be to retool Autumn to act as a parser generator instead of a parser interpreter as is currently the case. This would enable getting rid of megamorphic overheads and consequently unlock new optimization opportunities for the JIT compiler, as well as empower us to code specific optimization to get rid of provably redundant checks or optimize common grammatical patterns. For this, the work of Grimm on Rats [31] is a valuable source of information — this benchmark has clearly shown that the approach bears fruit in practice. We discuss this possibility in Section 6.5.8.

6.3 Lexical Analysis

PEG parsers are often marketed as *scannerless* [27] — this means they do not need a separate lexing (tokenization) step and can handle a stream of characters directly (instead of a stream of tokens).

In theory, CFG parsers can be scannerless too, but many programming languages do define their lexical layer separately from their grammar — and do so using constructions that cannot be reproduced in the CFG paradigm. Typically, the next token to generate for a stream of characters is taken to be the longest-matching token, something that cannot be emulated in CFG. Another common restriction is that identifiers may not use the name of *reserved keywords*.[18]

[18] It is also possible to extend the CFG formalism with new constructs in order to avoid separating the lexical and grammatical steps. This is for instance done by the Scannerless Generalized-LR Parsing (SGLR) technique [100] used in the SDF syntax definition formalism.

These are not necessarily major impediments, but they could lead to ambiguous parses (if your lexical layer uses longest-matches), which then need to be disambiguated by some other means; and to invalid syntax trees (if keywords are used as identifiers), which then need to be filtered a posteriori.

In contrast, PEG does have lookahead operators, which can be used to exclude some matches (such as identifiers using keyword names). We also saw that it is easy to add a longest-match parser combinator (Section 5.6.4) in the extensible procedural approach, and prioritized choice can generally be used to guarantee the same outcome. However, there are other reasons why lexical analysis is potentially advantageous, even in PEG.

6.3.1 Motivation

The first motivation for lexical analysis in PEG is performance, as we saw in Section 6.2. The PEG rules corresponding to tokens are "leaves" in the parsing graph, and as such can be invoked many times at the same input position. Second, usual programming language lexical rules enforce that only a single token may match at any given position. So not only are multiple invocations of the same "token rule" at the same position redundant, but the invocation of *any* token rule at the same position is redundant, once the correct token as been determined.

We noted a related observation in Section 6.1.3: Redziejowski found that performing a dictionary lookup on identifiers to see if they weren't using reserved keyword names reduced the amount of function calls by 20% (by getting rid of the negative lookahead for each keyword that would be otherwise needed).

A second motivation for lexical analysis is improved error reporting. In particular, knowledge about the lexical layer enables expressing errors in terms of tokens instead of just error positions. We discuss this further in Section 6.4.6, but for now we will just note that for this to be possible at all, lexical constructions need to be explicitly marked as such in the grammar.

Finally, supporting longest-matching lexical analysis emulation is convenient. If that is how the language has defined its lexical layer, then we can just mark every token rule and call it a day, instead of trying to reproduce the longest-matching semantics using the usual combinators.

6.3.2 Autumn's Lexical Analysis Emulation

All this suggests a new custom parser. A parser that, aware of all *tokens* defined in the grammar, will when first invoked at a input position, determine the token to match at that position (if any) and memoize the result in a memoization table. Then it will compare the result with the requested token and succeed or fail accordingly. Subsequent invocation of such "token parsers" at the same position will retrieve the memoized entry directly.

Autumn does implement this parser, and includes support for facilitating its use in its DSL. Code fragments demonstrating Autumn's facilities for lexical analysis are shown in Figure 6.2 — these were adapted from our full Java grammar.[19] The full Java grammar is reproduced in Appendix A, including the whole lexical layer specification.

If you don't recall the meaning of the `as_val`, `word` or `push` combinators, refer back to Chapter 3 (notably Section 3.1.2 and Section 3.3.1). `push_string_match` is a combinator that pushes the string matched by its sub-parser onto the value stack.

In Figure 6.2, we mark our *tokens* by using the `token()` combinator. This has two effects: first it registers the underlying parser as an "underlying token parser" and second it wraps it in a *token parser* that acts like described above: upon invocation it will determine the correct token (from the memoization table or by running all underlying token parsers and selecting the longest match) and succeed only if the selected token is the one recognized by the parser the `token()` combinator was applied to. These parsers can then be referred to directly, as shown in the `while_stmt` rule which refers to `_while`.

Under the wraps, each instance of `DSL` possesses an instance of the `Tokens` class, which collects all the underlying token parsers and manages the memoization of results.[20] Each token parser keeps a reference to this instance, which it uses to find the correct token for a given position.

[19] The identifier syntax was simplified to only allow alphanumeric characters.

[20] The `Memoizer` used to manage the memoization can be customized, as per Section 6.1.4.

```
// ...

public rule _while = word("while")                     .token();
public rule _false = word("false") .as_val(false)      .token();
public rule _true  = word("true")  .as_val(true)       .token();
public rule _null  = word("null")  .as_val(Null.NULL) .token();

public rule iden = seq(alpha, alphanum.repeat(0))
    .push_string_match()
    .word()
    .token();

// ...

public rule while_stmt =
    seq(_while, par_expr, _stmt)
    .push(xs -> WhileStatement.mk($(xs,0), $(xs,1)));

// ...

public rule literal = token_choice(
        integer_literal,
        string_literal,
        _null,
        float_literal,
        _true,
        _false,
        char_literal)
    .word()
    .push(xs -> Literal.mk(xs[0]));

// ...
```

Figure 6.2: Code fragments showcasing Autumn's facilities for lexical analysis.

Note that the order in which the tokens are defined is relevant: whenever two token parsers match the same amount of input, `Tokens` will select the token parser that appeared first. In Figure 6.2 this allows us to make sure that we will never match an identifier named `while`, `true`, `false`, ... which is a requirement in most languages.

What if our language *does* allow identifiers that clash with keyword names? Then it suffices to remove identifiers from the set of tokens. If performance is a concern, the `iden` parser can still be memoized on its own, as explained in Section 6.1.4.

Autumn includes a further performance-enhancing trick regarding tokens. It is not uncommon to want to match one token amongst a predefined set. This is exemplified in the `literal` rule of Figure 6.2. The naive way of doing this entails building a disjunction of token parsers, which means we will perform (at worst) one memoization table lookup per token in the set. Instead, Autumn offers the `token_choice` combinator, which will perform only a single lookup (or initial matching) and then compare all the token parsers in the set to the result.

Section 6.2 demonstrated that our lexical emulation brought about considerable performance improvements when compared to support compared to a naive (non-memoized) approach. But it also hinted at the fact that carefully ordered lexical-level rules along with judicious use of memoization could be even faster — by virtue of achieving the longest-matching semantics without actually having to try every lexical parser. Our preliminary experiments seem to confirm this. More research is required to determine if we can preserve the convenience of obtaining longest-match semantics by simply marking token rules, while closing the performance gap with the hand-optimized case.

One of the best things about Autumn's support for lexical analysis is that it is completely optional — it is an efficient emulation performed by custom parsers. One can still match against the character stream directly.

In fact, it is possible to take things even further than that, and mix multiple modes of lexical analysis! This is made possible by instantiating new `Tokens` instances. New token parsers can then be created by using `Tokens#token_parser(Parser)` and new token choice parsers by using `Tokens#token_choice(Parser...)`. This is handy when one wants to compose languages that use different sets of tokens or impose different

lexical rules.

We stress that the whole support for lexical analysis is build using user-available features. We conveniently package it for use, but users could have built it from scratch (including the DSL representation) without running into limitations imposed by Autumn.

Finally, Autumn also supports conventional external lexical analysis: Autumn's input can be either a textual `String` or a list of objects — which could be token emitted by a separate lexer.

In summary, Autumn enables emulating a separate lexical analysis step via a specialized form of memoization, made available as a built-in parser. There are multiple benefits to using this feature: first there are large performance gains to be had; second, the parser can be used as a marker of the boundary between the grammatical and lexical layer, which can be used to improve error reporting and handling; third, it enables composing multiple lexical analysis modes.

6.4 Error Reporting & Recovery

We discussed some background on error reporting in Section 2.6. We said that within most parsing tools, error reporting is usually based on *the furthest error heuristic*: whenever a parse fail, report the furthest input location reached by the parser.

In terms of error identification, it is difficult to *automatically* go beyond the furthest-error heuristic. Any improvement must necessarily come from domain knowledge and so be encoded into the grammar somehow. In fact, all the non-recovering error-reporting techniques surveyed in Section 2.6 rely on the user annotating the grammar in some way.

Error recovery, on the other hand, is slightly more amenable to automatization. This can in turn serve error reporting: indeed, an error that is successfully recovered from (leading to a successful parse, or even just to a big increase in the furthest error position) is more likely to constitute the real syntax error made by the programmer.

In this section, we discuss Autumn's modest built-in capabilities for error reporting, then we discuss multiple possible strategies for advanced error reporting and recovery — with or without domain knowledge — that can be implemented using Autumn's existing capabilities. We lead with

some performance discussion that is relevant to what follows.

6.4.1 Performance & Error Reporting

It is important to consider the cost of error-reporting techniques. Since many potential errors are encountered when running a backtracking parser, any computation made on these occasions is going to significantly contribute to the performance overhead of the parsing framework. Here too, the furthest error heuristic is advantageous, as it just requires us to keep track of a single number.

In general, parsing performance is much more crucial on valid input than on invalid input. When compiling a whole software project from scratch, we care about speed very much. However, in case of error, it's fine to take a bit more time, as long as the information comes fast enough. In other words, the issue changes from one of throughput to one of latency. To give some perspective, users don't seem perceive the latency of a touch action (like pressing keys on a keyboard) when it is under 96ms (when the feedback is projected in front of them) [20]. So for instance, it should take less than 96ms for a text editor to display a letter once it has been typed on a keyword, otherwise the user notice the input lag. By comparison, Autumn parses more than 6000 Java files in 23 seconds in the worst case (cf. Section 6.2), or less than 4ms per file on average. Clearly, even a two order of magnitude slowdown on the parsing of errors is acceptable. Even if many files contain errors, the user cannot take in the error messages as fast as they are produced. Therefore, a common error-handling strategy is to do a first, fast, parse; and in case of failure to perform a second, slower, parse that is able to more accurately pinpoint errors.

To support slower user-defined error-handling strategies, Autumn lets the user define custom options that are accessible from the `Parse` object. Advanced error tracking options can be set on slow parses but not on fast parses.

6.4.2 Autumn's Error-Reporting Capabilities

By default, Autumn only records the furthest error position. It also lets users specify error messages in failed parsers by using `Parse#set_error_message(String)` — after which the parser should return `false`, lest the call be meaningless and ignored. If the error turns out to be the furthest error encountered so far, the message is recorded — it will be discarded if an even furthest error is encountered.

Every parser also has an `ignore_errors` field, which if set to true, will cause any parsing error that occurs during the parser's invocation (including the invocation of its sub-parsers) to be ignored for the purpose of determining the furthest error. Within Autumun's lexical analysis emulation support (cf. Section 6.3), every token parser has this flag set: in this way, errors are never reported inside a "token" but only at token boundaries.

Though it is meant more as a help for grammar authors, Autumn is also capable of recording and displaying the stack of parser invocations at the time of the furthest parse error. This will be explained in Section 6.6.1.

Finally, Autumn can automatically translate input positions (which are simply indices) into a *(line, column)* format via a `LineMap` object. This object allows specifying a few parameters such as the starting column (as this can be 0 or 1 depending on the text editor), and the size of tab stops.

Autumn does not include further built-in error-reporting strategies, but has all the infrastructure necessary to build a vast array of error-reporting strategies building upon the user's domain knowledge. The rest of this section will explore a few strategies that could be employed to exploit these possibilities, even though we didn't thoroughly investigate the effectiveness of these options in practice.

There are many interesting directions, and not everything can be pursued. The ideal approach would have been to try to produce "production-level" error messages (*i.e.*, at least comparable in quality to those generated by the compilers of mainstream programming languages) for a real-world-sized grammar, such as our Java grammar (Appendix A). Evaluation can be performed on a predefined corpus of erroneous source files, potentially assorted with a corresponding corrected source file, or with some data indicating the nature and location of the actual error. This effort could have been the basis for the selection of a number of techniques and strategies that seem broadly applicable and high-value-added in practice, which could then be included in Autumn as built-in components.[21]

[21] In fact, this kind of process is what underpins the selection of most of the "built-in" facilities of Autumn — components (parsers, visitors, ...) that could have been written by the user but are valuable enough to be provided by default.

6.4.3 Custom Error States & Introspection

One first obvious thing that can be done is to collect additional information to serve in error reporting. Such information can be stored in a `ParseState` instance (cf. Section 5.6), but most likely won't need to be handled via the `Log`: we don't want error information to be discarded upon backtracking!

Custom parsers are also able to inspect the state of the `Parse` object to get the current input position and furthest recorded error (and possibly, the associated error message). Whenever the option is enabled, the parser call stack (cf. Section 6.6.1) can also be inspected.

6.4.4 Longest-Match Analysis

In our 2018 vision paper *"Red Shift: Procedural Shift-Reduce Parsing"* [55], we proposed a parsing paradigm based on unambiguous (prioritized) shift-reduce parsing. The peculiarity of the approach was that instead of attempting to only match rules that the grammar deemed acceptable, we attempted to match every grammar rule each time a parsing decision had to be made, and picked the first rule that succeeded. In fact, Red Shift doesn't feature grammar rules, but rather *reduction rules* which are custom functions that can manipulate the LR-like stack and the token stream. Reductions rules are an ad-hoc LR pendant to custom parser combinator.

We never came around to implementing a full-fledged version of this idea. First by lack of time, but second and foremost because we realized that while custom functions offer a lot of flexibility, there are a number of challenges around the definition of infix expressions (*of course*) that are very hard to solve without departing from Red Shift's principles. We still think our idea has merit, but a re-examination of the underlying principles would be required to overcome the challenge. A particular intuition we have here is that — just like transparent left-recursion handling in PEG required the introduction of a limited form of bottom-up parsing in the parsing — maybe the challenges around expression parsing will require the introduction of a limited form of top-down parsing.

Nevertheless, the whole experiment did inspire an interesting error-reporting strategy applicable to Autumn. The idea is as follows: whenever we fail to parse an input, we start a second "parsing stage" where we actually attempt to parse every single grammar rule at the start of the input. Once the longest-matching rule has been determined, we restart

this process, but this time at the input position where the previous match ended (or at the next position, if no rule matched). We continue this process until we have consumed the whole input, obtaining a sequence of matches in the process. This sequence of matches is equivalent to what we call *the most reduced stack* in the Red Shift paper. The sequence can be used as a basis for error recovery and error reporting.

In the Red Shift paper [55], we propose that the user define a series of *error-reporters* that try to determine why a reduction didn't occur. The same principle could be applied, but more broadly, in our proposed Autumn strategy. For instance, we could instantiate an error reporter for each sequence parser, which would try to find instances of the sequence with a single mismatched or missing item, within the sequence of matches. Let us take a concrete example: the error-reporter for the parser `seq("a", "b", "c")` would detect that in the input *axc*, replacing *x* by *b* could make the parser succeed. In this case, the longest-match sequence would be something like *[match(word("a"), 0), no_match(1), match(word("c"), 2)]* — indicating that *a* was matched at position 0 and *c* at position 2, while *x* didn't match any parser in the grammar.

Many other error reporters can be imagined. First for other kind of parsers (repetitions, longest-match, ...), but also implementing more general strategies. We could thus recapture and expand some strategies from Section 2.6: imagine reporters that speculatively insert, remove or replace a match from the sequence. We note this is a much more general than speculatively inserting a character or token: it allows inserting whole higher-level constructs, but also enables making predicate parsers succeed (which wouldn't necessarily be possible otherwise for context-sensitive predicates).

Such an approach is necessarily costly: parsing itself will be slower than a regular parser by a factor that depends on the size of the grammar and the input size. This aspect can be mitigated by memoizing all parsers, and sharing the same memoizer between all parsers and parses (something that can already be done in Autumn). The error reporters themselves could also prove costly (each of them may have to iterate the whole longest-match sequence), and there may potentially be many of them. Measurements are required to tell a more complete story.

Another challenge is to discriminate between multiple possible error reports. This probably requires the introduction of a second tier of heuristics to favour errors that can be corrected with finer changes.

If the approach were to work, it would be very attractive, as it allows us to deploy a mixed set of general heuristics (the error reporters) while leveraging the already-recognized structure of the input (in the form of the longest-match sequence). It remains to be seen if the approach is able to produce quality error messages in practice, and to do so in an acceptable amount of time.

6.4.5 Error-Recoverable Parsers

A way to enact error recovery in Autumn is to make parsers lie: make them pretend they succeed when in reality they didn't. This can be seen as a form of graceful degradation: first the parser tries to match according to its "real" semantics, then if it fails, it performs an error-recovery action. For instance, it could search for a synchronization point in the input (cf. Section 2.6) and then succeed as though it had properly consumed the input up to the synchronization point.

Such a "synchronization point" parser can be very effective if applied at strategic points in the grammar. In a language with curly brackets and semicolons, it could be applied around the rule for statements, and instructed that if it fails to match a statement, it should skip to the next semicolon or past the next balanced pair of curly brackets. Best of all, such a parser doesn't really require special support: just make a prioritized choice whose first sub-parser is the real deal, while the second sub-parser consumes input up to the synchronization point.

Another form of parser that may perform error recovery is what we call a *bounded parser*. A bounded parser has two sub-parsers: the first, *coarse*, sub-parser delimitates a part of the input (for instance, a portion of input contained between a matched set of curly brackets, without inspecting the intervening characters except to ensure proper bracket pairing), while the second, *fine*, sub-parser is used on the delimitated input.

Note that these semantics cannot be obtained by using the lookahead combinator: nothing prevents the fine parser to parse further than the coarse parser. In fact, implementing the bounded parser required us to add a small feature to Autumn: an `end_of_input` field was added to `Parse`, holding the position of the end of the input. The bounded parser temporarily changes this field to make sure the fine parser can't escape the boundary delimited by the coarse parser.

Beyond its increased expressiveness, the bounded parser can be used for error recovery: if the fine parser fails to match, the bounded parser can

still succeed having matched the same thing as the coarse parser. The canonical use case is matched pairs of delimiters such as brackets and quotes (`{}`, `<>`, `''`, etc).

In Autumn, the bounded parser is implemented in the `Bounded` class, which additionally to its two sub-parsers, takes a function that will be run on the `Parse` object if the fine parser fails, and returns a boolean which determines if the bounded parser succeeds or fails.

We could also work in the opposite direction: instead of degrading our parse in case of error, we could progressively refine the parse: we can imagine a first parse that only matches code as a set of nested blocks, a second parse may break these blocks up in statement while a third parse tackles the nitty gritty of expressions. The advantage of this approach is that it doesn't require the use of custom parsers.

We still need to discuss an important piece of the puzzle: when a parse is degraded in favour of a recovery action (whether reaching a synchronization point, or eschewing a failed fine parser), how should this information be communicated? We can conceive of two solutions. The first is to emit AST nodes corresponding to the errors. For instance, instead of the node that the fine parser would have generated, we generate a node that indicate an invalid (but bounded) stretch of input. In fact, this is not only a way to report an issue, but also a necessity in many cases: the AST construction logic will typically expect a node from the bounded parser, and so a node must be provided! The second, complementary solution is to report the event somewhere, such as a list of "errors" stored in a `ParseState`.

These error-recovery mechanisms don't really help precisely pinpoint errors or generate good error messages. What they do however enable is to continue parsing past an error site, which is a necessity if we want to report multiple errors within a single input. Often, even partial parse data can be exploited. This is most notably the case in IDEs: even if a statement contains a syntax error, we would still like to provide services to other statements, such as syntax highlighting, code analysis, auto-completion, etc.

6.4.6 Lexical-Level Error Reporting

As we have seen in Section 6.3, Autumn includes support for lexical analysis emulation. Lexical analysis (the process of breaking up the input in a chain of tokens) can be used to improve error messages — by

referring to tokens instead of just input positions. In fact, the following kind of error message has become rather standard in all sorts of compilers: *"Unexpected token X at line 2 column 3, expected token Y, Z or W."*

Can we achieve something like this in Autumn: enrich the furthest error message with token data? We can, provided that we supply the `Tokens` class (which is responsible for parsing tokens) with a custom memoizer (cf. Section 6.1.4). This memoizer should have two requirements: first, it should hold on to the tokens it matched, and second, it should record which tokens were attempted at which position.[22] The second requirement does add a fair bit of overhead, and full memoization doesn't seem to be the fastest option in practice (Autumn uses a cache with 8 entries by default).

Another reason why lexical-level error reporting is not built into Autumn is that while these messages look good, they typically aren't very helpful: looking at the indicated input position immediately reveals which token was matched there, and there are typically *a lot* of tokens that could have been accepted instead. For instance, in Java, if a semicolon is misplaced after an identifier in a parameter list, the parse would have progressed if it had been replaced with almost any infix or postfix operator, a comma, an opening or closing parenthesis, ... There just are not a lot of cases where we only expect a couple of different tokens, though these surely do exist: for instance (still in Java), expecting an identifier after a type name in a variable or method declaration. But in these cases, the user's surprise generally is not about what was expected there, but about the fact that what he put there was rejected. This would for instance be the case if trying to add a type-parameter list after the return type in Java: `void <T> foo();` is illegal but `<T> void foo();` is legal.[23]

In summary, Autumn's default error-reporting capabilities are simple: it can report the furthest error encountered, and user-specified messages can be attached to such errors. Prioritized choice, as well as the `Bounded` parser can be used to implement permissive parsing — falling back to a permissive input specification whenever the precise input specification fails

[22] We could discard memoized information for input position lower than the current furthest error, but some parsers (those that have their `exclude_errors` field set) can reset the position of the furthest error — so doing so is inherently unsound.

[23] This example is interesting, because the error message we would like to generate — saying that `<T>` is misplaced — is almost impossible to generate automatically: we have to encode this particular case explicitly. An error reporter from Section 6.4.4 might be able to do so, however.

to match. Autumn also includes many features that makes it possible to build advanced language-specific error reporting and recovery capabilities, such as the possibility for users to manage their own state, and the ability to inspect the parser invocation stack. We also explored an example of what a powerful error reporting and recovery system could look like in practice (longest-match analysis).

6.5 Grammar Traversal & Parser Visitors

In Section 1.3, we emphasized the need for the grammar *reification*. Autumn doesn't really feature a first-class notion of grammar, though the term is used to mean the class in which rule definitions appear. Instead, the grammar is de facto reified in the form of the parser graph, made out of all `Parser` instances (cf. Section 3.2), with the method `Parser#children()` defining the parent-child relationship — directed edges. This section explains how this reification can be exploited in practice.

6.5.1 Grammar Traversal

First we note that, in the presence of recursion, a parser graph will be cyclic. This makes its traversal challenging. Indeed, it precludes us from traversing the graph as if it were an acyclic graph, using simple recursion to traverse the children. Additionally, it is generally desirable to traverse a given parser only once, even when it is reachable through multiple paths in the parser graph.

To remedy these issues, we provide the `ParserWalker` abstract class. This class defines the `walk(Parser)` method, used to traverse the parser graph starting from the given parser (only parsers reachable through this parser will be traversed), and the abstract `work(Parser, State)` method to define the work to be done for a given parser. Here `State` designates one of multiple possible states the walker can be in, in relation with the parser.

The possible values of `State` are `BEFORE`, `AFTER`, `RECURSE` and `VISITED`. For each reachable parser, the `work` method is guaranteed to be called exactly once with state `BEFORE` (before traversing the parser's children) and `AFTER` (after traversing the parser's children). Additionally, it may be called any number of times with state `RECURSE` — when encountering a recursion on the parser (*i.e.*, `work` has been called for that parser with state `BEFORE`, but not yet with state `AFTER`). This will cause the recursion to be cut off: the parser's children won't be traversed again. For a given

parser, `work` may also be called any number of times with state `VISITED`, indicating that the parser has already been traversed (*i.e.*, `work` was invoked with both `BEFORE` and `AFTER` states) but was encountered again through a different path in the graph. This also cuts off recursion to ensure that no parser is traversed twice.

This interface enables non-recursive depth-first traversal of the graph, and should be reasonably familiar to most programmers. We do note the existence of more expressive traversal models [6, 11], which enable flexible traversal customization and, accordingly, more expressive matching of the state of the traversal (via *join points* and *pattern matching*). These approaches are often also concerned with tree rewriting during the traversal. We only support rewriting in the sense of producing a modified copy, which is achieving by extending `CopyVisitor`, a `ParserWalker` implementation (and also a visitor) which performs a copy of the parser graph. This will be explained in Section 6.5.6.

6.5.2 The Expression Problem

In addition to traversing the grammar, it is often opportune to define operations whose implementation varies depending on the type of parser you are dealing with. In an idealized world, one would add an abstract method to `Parser` and implement it for all parsers. Alas, `Parser` is defined as part of Autumn, and users cannot modify it (nor can they modify its built-in subclasses). An alternative is to use the well-known *visitor pattern* [29]. Here is a brief summary of how the pattern would apply to Autumn:

- Create a `ParserVisitor` interface with one overload per `Parser` implementation (*e.g.*, `visit(Sequence)`, `visit(Choice)`).

- Add a single abstract `accept(Visitor)` method to `Parser`.

- Implement this method for all parsers, having it call the overload method with `this` as parameter (*i.e.*, `visit(this)`). Because the call is made in the method implementation, the static type of `this` will coincide with its runtime type and the correct overload will be called.

- Define new operations specialized per-parser by implementing the `ParserVisitor` interface, and passing an instance of the implementation to a parser's `accept` method.

This solution has a big downside: all parser implementations need to be listed in the visitor interface. Imagine we distribute such an interface with Autumn, including an overload for each built-in parser. In theory, users won't be able to add specialized operations for the custom parsers they implemented themselves! Since extensibility is one of the main selling point of Autumn, this is unacceptable.

The issue — the inability to add both new operations *and* new forms of data (in this case, parsers) — is a well-known one: the *expression problem*, a term coined by Philip Wadler in a discussion on the Java Generics mailing list [102]:

> The expression problem is a new name for an old problem. The goal is to define a datatype by cases, where one can add new cases to the datatype and new functions over the datatype, without recompiling existing code, and while retaining static type safety (*e.g.*, no casts).

In an object-oriented system, datatype cases are subclasses (or interface implementations), while functions are methods (more precisely an abstract method and its implementations) or visitor implementations. One can see the dilemma: if methods are fixed (in the superclass), one can add new subclasses easily. If the subclasses are fixed, one can add new visitors easily. It is when one wants to be able to add both that things become difficult.

The literature on the expression problem in object-oriented languages (and in particular Java) is surprisingly rich [95, 69, 103]. Most of these solutions do however have issues that make them undesirable for our purpose.

First, they all struggle with the issue of independent extensibility [109], which in our case means that if two different users implement (for instance) new visitors and new parsers which they redistribute, a third user should be able — after supplying the required specializations — to use the visitors of the first user with parsers of the second, and vice-versa.[24] Zenger and Odersky, who first publicized the independent extensibility requirement [109] propose a solution in Scala, which uses a combination of *abstract type members* and *mixin composition*, two advanced features

[24] The issue generally stems from Java and similar language's proscription of multiple inheritance. Extensions can be made linearly by inheritance, but divergent extensions cannot be composed back together. As we will see, the introduction of default method implementations in Java 8 brings about change.

unavailable in Java.[25] Similarly, object algebra are amenable to independent extensibility [70] in Scala, but this involves a tremendous amount of boilerplate and is not type-safe when done in Java. Other methods do not support independent extensibility.

Another limitation, less often noted, is that since these solutions do use complex type encodings (for instance, using advanced parametric polymorphism, or even representing an expression as a call graph instead of a data structure [69]), it is not always possible to aggregate them in even a simple collection.[26]

6.5.3 A Clean But Verbose Partial Solution

Quite clearly, existing solutions to the expression problem in Java won't do — parsers are frequently collected in arrays inside parsers, we value independent extensibility, and foisting a lot of byzantine generics upon the user seems like one of the worst possible thing to do. So we set out in search of a solution that would work better for us.

The first scheme we arrived at is based on the classical visitor pattern as presented in Section 6.5.2. We forego type-safety in the sense that we cannot longer guarantee that a visitor passed to a parser's `accept` method will be able to handle that particular parser. It is now the user's responsibility to ensure that the supplied visitor handles all parsers it might visit.

We enable independent extensibility by using Java 8's `default` method definition in interfaces. Our visitor implementations now live in interfaces instead of classes. The final user is now responsible to create the class that will "tie together" the visitor's implementation. The advantage of this is that a single class can inherit default implementations from multiple interfaces, making independent extensibility possible.

[25] The solution has common another issue, namely requiring to change all data classes when new operations are introduced, which either causes data classes in different part of the code to be incompatible, or requires the parameterization of all instantiation sites.[27]

[26] When it is possible, it is either that type-safety is not enforced — which we'll see is an interesting concession to make — or that (for the concerned solutions) all type parameters have been made uniform. However, this requires parameterizing all the instantiation sites.[27]

[27] Such parameterization of instantiation sites adds a lot of noise to the code, even when using a specialized tool such as a dependency injection framework.

To make things more concrete, we will use the code skeleton shown in Figure 6.3 to exemplify the principle of the solution.

The `Sequence` parser class is used as an example of how the `accept` method is implemented in built-in parsers. We also have one visitor implementation (`MyVisitor`) with one overload for each built-in parser. This is simply the classical visitor pattern, with the important difference that `MyVisitor` is an interface!

We now imagine that two users each independently create a new parser: `Foo` and `Bar`. The scheme calls for them to extend the `ParserVisitor` interface with an overload for their new parse: `VisitorFoo` and `VisitorBar`. To make `MyVisitor` compatible with the new parsers, our users need to extend it and implement the specialized visitor class. This is done in `MyVisitorFoo` and `MyVisitorBar` (again, these are interfaces).

Finally, imagine a third user that uses both the `Foo` and `Bar` parsers in his grammar, and also wants to run `MyVisitor` on the parser graph. This user must compose `MyVisitorFoo` and `MyVisitorBar` together as `MyVisitorFooBar`, which he can then use to actually invoke the visitor (exemplified in the `main` method).

Dear reader, this is not the solution we ended up using.

This was, we still think, a good solution. It trades a bit of compile-time safety in order to satisfy the other requirements, without resorting to complex encodings that require extensive parameterization. Additionally, it supports independent extensibility and the free manipulation of parsers (which can be collected in collections).

But good doesn't mean best. When taken from the user perspective, a user that writes some new parsers now needs to define one additional interface, and one more for *each* existing visitor he'll extend (and ideally, we would want him to extend all of them!). If he wants to use any of these visitors, he'll also have to create a class for each of those. There is not much complexity going on here, but this is a whole lot of boilerplate that might understandably put off the user. From the perspective of a user that reuses third-party parsers and visitors, there is composition to be performed (*i.e.*, writing an empty class that `implements` the proper interfaces) for *every* visitor he wishes to use. That just is not very good user experience.

```
1  public final class Sequence extends Parser {
2      // ...
3      @Override public void accept (ParserVisitor visitor) {
4          visitor.visit(this);
5      }
6  }
7
8  public interface MyVisitor extends Visitor {
9      // ...
10     @Override default void visit (Sequence parser) { /* ... */ }
11 }
12
13 public final class Foo extends Parser {
14     // ...
15     @Override public void accept (ParserVisitor visitor) {
16         ((FooVisitor)visitor).visit(this);
17     }
18 }
19
20 public interface VisitorFoo extends Visitor {
21     void visit (Foo parser);
22 }
23
24 public interface MyVisitorFoo extends MyVisitor, VisitorFoo {
25     @override default void visit (Foo parser) { /* ... */ }
26 }
27
28 public final class Bar extends Parser {
29     // ...
30     @Override public void accept (ParserVisitor visitor) {
31         ((BarVisitor)visitor).visit(this);
32     }
33 }
34
35 public interface VisitorBar extends Visitor {
36     void visit (Bar parser);
37 }
38
39 public interface MyVisitorBar extends MyVisitor, VisitorBar {
40     @override default void visit (Bar parser) { /* ... */ }
41 }
42
43 public class MyVisitorFooBar implement MyVisitorFoo, MyVisitorBar {}
44
45 public static void main (String args[]) {
46     some_parser.accept(new MyVisitorFooBar());
47 }
```

Figure 6.3: Code skeleton showing the outline of our first solution to the expression problem, using a scenario where independently-developed extensions (new parsers extending an existing visitor) are composed.

6.5.4 A User-Friendly Partial Solution

The amount of boilerplate in the previous solution bothered us very much. We consider the ability to not only develop custom parsers but also *reuse* those created by others to be key in Autumn. At the same time, the ability to reify and work with the grammar is paramount for a flexible and re-purposable parsing tool. Making visitors too verbose makes these two aspects at odds with one another: the authors of custom parsers are less likely to extend visitors they don't themselves use, and the users of independently developed custom parsers will be bothered by the need for manual assembly.

One possibility we had in mind was to simply have a visitor be a map from (parser) classes to some kind of function corresponding to a former `visit` overload. This is not compile-time safe, but neither was the previous solution. A glut of hash table lookups can slow things down, but visitor invocations don't tend to be on the critical performance path. They are used as analysis and generation tools, and are run against the grammar, not the input. Our built-in visitors (cf. Section 6.5.5) also tend to be data-structure heavy already, so adding these lookups won't fundamentally alter their performance characteristics.

But we can do better: we can keep the classical visitor pattern for built-in visitors, but add an overload that will catch all unhandled parsers (with signature `visit(Parser)`) and perform a map lookup within that overload. In that way, no overhead is introduced for built-in parsers, which should form the bulk of the parsers used in a grammar.

To underline the difference, let's look at Figure 6.4, which develops the same scenario as Figure 6.3 but using this new hybrid solution. We didn't reproduce the `Sequence` class, which is unchanged from before. `MyVisitor` is now a class, and only requires specifying a default action for unhandled custom parsers: when possible, the most conservative assumptions possible should be made, otherwise an exception should be thrown — these are the cases that compile-time safety was supposed to catch. Note it is now no longer necessary to override `accept` in the custom parsers. The visitor logic for the custom parsers now lives in a

```
public class MyVisitor implements Visitor
{
    // ...

    @Override public void default_action (Parser parser) {
         // ... (some conservative behaviour, or an exception)
    }

    @Override default void visit (Sequence parser) {
        // ...
    }
}

public final class Foo extends Parser {
    // ...
    static {
        ParserVisitor.extend(MyVisitor.class, Foo.class, (parser, visitor) -> {
            // ...
        });
    }
}

public final class Bar extends Parser {
    // ...
    static {
        ParserVisitor.extend(MyVisitor.class, Bar.class, (parser, visitor) -> {
            // ...
        });
    }
}

public static void main (String args[]) {
    some_parser.accept(new MyVisitor());
}
```

Figure 6.4: Code skeleton showing the outline of our second solution to the expression problem, using a scenario where independently-developed extensions (new parsers extending an existing visitor) are composed.

static initializer within the parser.[28]

There is actually a lot more than meets the eyes in this version. Where are these maps we talked about earlier? They are actually managed by `ParserVisitor` by default. This is itself quite tricky: how can an interface manage a map for each of its implementations? By going meta, and keeping these maps in a higher-level map whose keys are visitor `Class` objects. To avoid the overhead of doing not one but two hash table lookups each time a custom parser is visited, a caching mechanism is implemented. The theory is simple: we expect to see a visitor used repeatedly, rather than see multiple interleaved invocations of different visitors. If that should nevertheless be the case, we added the option for the visitor to manage his own map, alleviating the need for the first lookup. For bonus points, both the caching mechanism and the `ParserVisitor.extend` method are thread-safe.

Another notable thing is that `ParserVisitor` manages the *dispatch* of the visitor to the correct implementation (held in the maps) itself. It does so by providing a default implementation for the `ParserVisitor#visit( Parser)` "catch-all" method, which is called by the `Parser#accept( ParserVisitor)` method for all custom parsers. If no such implementation is found, then the `default_action` method is called instead.

Our first iteration on the idea wasn't so elegant: it required each visitor to manage its own map, extension mechanism (*i.e.*, the equivalent of the `extend` method) and dispatch mechanism. Composition became easier, but on the other hand it still foisted some boilerplate on the authors of visitors.

In summary, we went from an efficient but verbose and inconvenient solution to one that traded-off some (but in most cases, very little) of that efficiency for an almost-optimal level of brevity. We appreciate that this solution is really simple and hard to misuse: the only thing to do is call `ParserVisitor.extend` before running a visitor — preferably only once, but the method is idempotent so even that aspect of it is forgiving. By contrast, the previous solution (Section 6.5.3) had quite a lot of rules in how the boilerplate had to be constructed — so many opportunities to

[28]This is a good place for them, because it guarantees that the visitor logic will be loaded whenever the class is loaded. When adding visitor logic to a parser you didn't write, you can alternatively put it in a `static` initializer of your grammar class. The key is that the `extend` call should be made before the visitor may visit the parser — and preferably only once!

get things wrong. The independent extensibility scenario involves three users, and it is possible that only the user performing the composition may suffer from a poorly executed boilerplate (such as a visitor directly implemented as a class instead of an interface). Not an enviable situation.

The visitor architecture may at first seem a rather inconsequential matter, but it has a big user-experience impact on one key aspect of the tool, and it was one of the big implementation challenges we faced. We felt it deserved a good discussion, which shows how these matters can be handled in practice, and in extensible frameworks in particular.

6.5.5 Well-Formedness Checking Using Built-In Visitors

Autumn bundles four built-in parser visitors and three parser walkers (two classes actually belong to both categories). All but one of these classes work in concert to check if Autumn grammars are *well-formed* [27]. Well-formed grammars contain no left-recursion — in our case, *unhandled* left-recursion: left-recursion broken by the `left_recursive` (cf. Section 4.5) combinator is fine — and no repetitions over parsers that are nullable (*i.e.*, parsers that can succeed without matching any input). These visitors and walkers are:

- `VisitorNullable` is both a visitor and a walker, used to check the nullability of parsers.

- `VisitorFirstParsers` is a visitor that, for a given parser, determines its *FIRST* set (cf. Section 4.6): the set of direct sub-parsers (children) that this parser may invoke *at the same input position*. This visitor uses a `VisitorNullable`. Indeed: `sequence(a, b)` includes both `a` and `b` in its *FIRST* set if `a` is nullable.

- `VisitorNullableRepetition` is a visitor that, for a given parser, determines if that parser can repeat over a nullable parser. It obviously uses a `VisitorNullable` as well.

- `WellFormedChecker` is a walker that uses a `VisitorFirstParsers` to find unhandled left-recursive cycles in the grammar; and runs `VisitorNullableRepetition` on every parser in the grammar.

By default, Autumn runs the well-formed checker on a grammar before each parse (this can be disabled via an option), and emits error messages if it finds unhandled left-recursive cycles or nullable repetitions. This prevents parses that unexpectedly loop forever.

A point of detail: `VisitorNullable` is both a visitor and a walker because, unlike the two other visitors, it is highly recursive in nature. Its use by `VisitorFirstParsers` caused a massive amount of repeated visitor invocations, therefore we had to memoize `VisitorNullable`'s results. Given that, implementing it as a `ParserWalker` is mostly a matter of convenience. When the nullability of a parser is queried, a walk is triggered. A parser's nullability is determined on the `AFTER` state. At this point, by definition, the nullability of its sub-parsers has already been established and we can read these results from the cache.

Moreover, a walker retains its state after a walk, and we can consequently traverse the parser graph from multiple different entry points while preserving the guarantees outlined in Section 6.5.1: the `work` method is only invoked once with the `BEFORE` and `AFTER` states, but is invoked with `VISITED` each time it is encountered after the `AFTER` state. Therefore, as `VisitorFirstParsers` and `VisitorNullableRepetition` query `NullableVisitor`, the cache of results builds up progressively, without any repeated work.

6.5.6 Grammar Transformation Using `CopyVisitor`

The last built-in visitor is a also a walker: a `CopyVisitor` performs a deep copy of a grammar's parser graph. By extending this class and overriding its `visit` methods, one can opt to replace a straight parser copy with a modified version instead.

In Section 4.6, we said that while we could statically detect left-recursion (as we explained in the previous section), we do not want to rewrite the grammar to insert left-recursive combinators. One key reason is complexified debugging (traces don't correspond to the grammar). Another problem concerns the semantics of parser walkers: should they be run before or after a grammar transformation, or maybe both?

The idea of performing a transformative *copy* helps clarify the semantics of walkers and visitors. We are not *rewriting* the grammar in-place: we are creating a new modified grammar, which can be used independently of the original, and on which walkers and visitors may also be run independently.

There are a couple of things to look out for. A copy of a grammar will refer to the original grammar's `Tokens` object and `ws` rule. This is not necessarily an issue: the `ws` rule and `TokenParser` instances can be modified during the copy, and one can get hold of the `Tokens` instance

via the embedded grammar's instance, to extend it with new tokens, or reuse it as-is.

A slightly more intricate pitfall is that a number of parsers take functional interfaces as parameters, which are typically supplied as lambda functions. These functions often capture values from the original grammar. If the author of the grammar followed the framework's guidelines and used `ParseState` instances appropriately, this should be safe.

However, there is a case we cannot handle: if lambda functions capture parsers from the original grammar, we have no way to replace them by their copies. Therefore, the result probably won't be what is expected. We note that we have never needed to capture parsers in lambda functions, and that the issue can always be avoided (by adding a layer of indirection) if one has the foresight to consider that the grammar might be copied.

We would, in general, advise against performing grammar transformation if someting else (say a judiciously inserted custom parser) might work instead. Nevertheless, making rewriting grammar copies opens up interesting applications, and none more interesting than the ability to perform grammar composition — which will be the object of Section 6.7.

6.5.7 Abstract Parsers

Section 6.5.4 showed how we made it very easy to make new parsers compatible with visitors. But can we not do even better? We have stripped away all the boilerplate, but one still has to provide an implementation for each *(parser kind, visitor)* pair.

Well, maybe not. We observed that not all parsers are meant to be reused. Very often, we will develop parsers that will be specific to a particular grammar. For instance, such is the case of the `CloseTag` parser in the XML grammar of Section 5.6.4. In fact, this is typical of context-sensitive parsers.

To help us, we introduced a series of *abstract* parsers: abstract classes that can be extended to create new custom parsers. Each kind of parser captures some kind of recurring combinator pattern, and helps in two ways. First by providing default implementations of some abstract `Parser` method (such as `toStringFull` for string representations). Second, they enable specifying default visitor actions that have the potential to cover a wide class of parsers. These actions can still be overriden when necessary.

The bundled abstract parsers are:

- `AbstractPrimitive`: for parsers that have no sub-parsers and match directly against the input. Its constructor expects a string to be used as string representation of the parser, and a boolean that indicates the nullability of the parser.

- `AbstractWrapper`: for parsers that have a single sub-parser, and can match the same input as their sub-parsers (or a subset thereof). Typically used for parsers that wrap another but add some kind of context-sensitive guard.

- `AbstractForwarding`: for parsers that delegate their parsing to a single sub-parser. This differs from `AbstractWrapper` in that the `doparse` method is already implemented: this parser does *exactly the same thing* as its sub-parser. The point of a forwarding parser is two-fold. First it may help with debugging as this is a parser whose name will appear in traces. Second and most importantly, the parser serves as a "marker" of sort that may be used by some visitors to enact specialized behaviour.

- `AbstractChoice`: for parsers that behave like a choice and can match the same thing as any of their sub-parsers. Typically used when context is used to enact the choice between these parsers.

6.5.8 Potential Further Applications

We have multiple ideas of how visitors and walkers could further be put to work to implement interesting features.

Automatic Synthesis of Context-Object Extractors

Recall from Section 5.6.3 that mixing context-sensitive parsing and memoization requires extracting a *context object* from the global context, which is then used to discriminate between different memoized entries for the same parser at the same input position. The user must define an extractor function to perform this task. As we said, this breaks the property of *context transparency* (Section 5.3): if a parser reachable by the memoized parser becomes context-sensitive, the extractor needs to change accordingly.

We could alleviate this issue by automatically synthesizing the extractors, so as to guarantee that they always stay synchronized with the grammar.

For this to work, we would need each parser to declare the context it depends on.

A simple way to do this would be to have parsers specify local extractors, which can be used to acquire local context objects. These local context objects encapsulate the context that parser locally depends on (so the context depended upon by their children need not be taken into account). We can then write a parser walker to aggregate local extractors. From a given memoized parser, the reachable parsers are traversed and their local extractors are collected. They are then used to synthesize the memoized parser's extractor.

A potential pitfall is that multiple parsers can depend on the same context, which could lead to duplication inside the context object. A possible mitigation is to predefine and reuse frequently-used local extractors.

Parser Generation

In Section 6.1.5, we discussed performance issues resulting from the abundance of megamorphic call sites in Autumn (and parser combinator frameworks in general). The issue is that the combinator approach — while it enables reification — incurs performance penalties in the form of dispatch overheads at megamorphic call sites. These call sites, because of their inherent indeterminacy, also preclude inlining — and subsequently a lot of other optimization opportunities — from being applied by the JVM.

However, using walkers and visitors makes it relatively easy to turn Autumn into a parser generator. We need a visitor that generates relevant parsing code for each parser, potentially reusing the code generated for its sub-parsers. The walker is simply responsibly to ensure that the visitor is run on all parsers in the proper order (the sub-parsers before the parent) and handles recursion.

The simplest generation scheme is simply to generate for each parser one method that contains the same code as the combination of its `parse` and `doparse` methods, replacing sub-parsers invocations by call to the methods generated for these sub-parsers. This removes the megamorphic call sites and so unlocks optimization opportunities. From the theory and results in the literature, we expect performance gains from this alone.

But it is possible to take things even further, by performing domain-specific optimizations on the generated code. For instance, a lot of the

logic defined in `parse` to handle backtracking is quite often redundant. When generating, we can statically identify these cases and optimize them (either on the fly, or as a post-processing step, using other walkers and visitors). Other optimizations of this kind — such as automatic left-factoring of shared prefixes in choice alternative — have been researched by Robert Grimm and deployed in the parsing tool Rats! [31], with good performance results.

Annotation-Driven Analysis & Generation

In general, interesting analyses and generations may require further information on parsers that has to be provided somehow. Sometimes this information only depends on the kind of parser (the subclass of `Parser`) and then it can be provided by a visitor. But sometimes, the information pertains to how parsers are used, and the user has to annotate parser instances himself.

A good example is derivation of a syntax highlighter from a grammar: the user has to specify what parts of the syntax has to be highlighted, and how. Providing these annotations can be done in a variety of ways, such as a map from `Parser` or even rule name to data; or as Java language annotations, which can be retrieved by performing reflection on the grammar class. Creating a new kind of parser that extends `AbstractForwarding` (as discussed in Section 6.5.7) is also a good option.

6.6 Debugging & Tracing

Autumn comes outfitted with a couple features that make it easier for grammar authors to ascertain that the resulting parser behaves as expected, troubleshoot an incorrect grammar, and identify the cause of lackluster performance.

We already saw one such tool in the form of the *well-formedness check* of Section 6.5.5. And of course, error-reporting facilities double as debugging facilities (Section 6.4). This section lays out the rest of what Autumn has to offer to promote faster and safer grammar development.

6.6.1 Parser Invocation Stack Traces

One of Autumn's essential debugging features is the ability to record and inspect the stack of parser invocations. The Java runtime can also supply stack traces, but these only indicate a *(class, method)* pair — so the exact parser can't be pinpointed. If the stack shows `Sequence#parse`

```
Parse succeeded, consuming up to 31:5.
Furthest parse error at 31:13.
        at 1:1 in root
        at 1:1 in sequence(ws, maybe(package_decl),
                        import_decls, type_decls)
        at 27:1 in type_decls
        at 27:1 in repeat(choice(type_decl, SEMI), 0)
        at 31:5 in choice(type_decl, SEMI)
        at 31:5 in type_decl
        at 31:13 in type_decl_suffix
```

Figure 6.5: Pretty-printed parser stack trace for a syntax error in a Java file, involving an early closing curly bracket.

as one of its entries, that doesn't tell you which sequence parser in your grammar it refers to, nor at which input position it was invoked. Such a stack trace also contains intermediate function calls which are irrelevant to the purpose of debugging. In contrast, the parser invocation stack in Autumn directly records parser objects, as well as the input positions at which they were invoked.

The stack of parser invocations (henceforth: the *parser call stack*) is analogous to the method call stack one can obtain from Java: it holds every parser whose invocation is ongoing. Parser call stack recording is an option that can be enabled for any parse. It is disabled by default, as it is meant for users who want to troubleshoot an issue in their grammar, and slows things down a fair bit.

When the option is activated, Autumn keeps track of the parser call stack at any given time. It also keeps around a copy of the stack made when the furthest error was encountered. This last stack can be accessed as part of the parse results returned after a parse. Both the current parser call stack and the furthest error parser call stack can also be accessed by custom parsers, for further diagnosis during the parse. Autumn of course supports pretty-printing parser stacks.

Figure 6.5 shows an example of pretty-printed parser call stack, using our Java grammar (which is reproduced in Appendix A). In this case the error is that a class body was closed too early. The class starts at line 27, and ends before line 31. Consequently, the parser skips the whitespace following the closing curly bracket, and tries to match another type declaration (rule `type_decl`), but fails to do so. Note that it says

"parse succeeded" because the parser was able to match a prefix of the input (the package and import declarations).

Finally, we note that it is possible to only show parsers which correspond to named rules in the grammar to further cut through the noise. In Figure 6.5 this would be equivalent to removing the `sequence`, `repeat` and `choice` parsers from the listing.

6.6.2 Custom Parsers for State Inspection

We said that a custom parser could access the parser call stack during the parse. This is actually a common pattern: inject a custom parser into the grammar to inspect the state of the parse as some parse is executed. The easiest way to achieve this is to add a `ContextPredicate` parser in a sequence. This parser takes a `Parse` → `boolean` function and is normally intended to serve as a predicate over the context, but we can use it to print out debugging information (or store it in some global data structure), and then return `true` to proceed with the parse. Another possibility is to set a debugging breakpoint in the function passed to the combinator, which enables inspecting the parse state using the debugger whenever the parser is invoked.

6.6.3 Testing Support

Building upon the foundation of parser stack traces (Section 6.6.1), Autumn provides testing support which makes it easy to test a grammar piece-wise, by defining inputs on which a given parser should succeed or fail. It's also possible to verify that the parse produces the correct AST, or that it fails at the expect input position.

Testing support is implemented in the `TestFixture` class, which test classes are supposed to extend. The implementation is compatible with the *JUnit TestNG* testing frameworks. [29] The regular testing workflow is to define the parser under consideration, then to invoke one of the testing methods defined by `TestFixture`[30]:

- `success(String)` checks that the parser being tested succeeds matching the entirety of the given input.

[29] And probably with others too: we simply throw a standard `AssertionError` whenever an assertion fails.

[30] In reality the methods can accept not just string inputs but also inputs that are lists of objects.

```
1  @Test public void literals()
2  {
3      rule = grammar.literal;
4
5      success_expect("4_2L",          Literal.mk(4_2L));
6      success_expect(".42e42",        Literal.mk(.42e42));
7      success_expect("0x8",           Literal.mk(0x8));
8      success_expect("0x8p8",         Literal.mk(0x8p8));
9      success_expect("0111",          Literal.mk(0111));
10     success_expect("true",          Literal.mk(true));
11     success_expect("false",         Literal.mk(false));
12     success_expect("null",          Literal.mk(Null.NULL));
13     // ...
14
15     failure("#");
16     failure("identifier");
17     failure("_42");
18     failure("42_");
19
20     success_expect(".42e-48f",
21         Literal.mk(new LexProblem("Float literal is too small.")));
22     // ...
23 }
```

Figure 6.6: A test method for literals in the Java grammar, using Autumn's testing facilities.

- `success_expect(String, Object)` also checks for success but additionally checks that the value at the top of value stack (Section 3.3.1) is equal to the second parameter.
- `prefix` and `prefix_expect` are analogous to `success` and `success_expect`, but only require the parser to match a prefix of the input.
- `failure(String)` checks that the parser fails to match the entirety of the given input.
- `failure_at(String, int)` checks that parser fails to match the input, and also that the furthest error occurs at the given position in the input.

Figure 6.6 shows a test method from the test suite for our Java grammar (Appendix A). This method tests the syntax of literals. We start by

setting the field `rule` to the `literal` rule of our Java grammar. The assertion methods will automatically extract the parser to test from rule. We follow this with a couple of inputs that should succeed, also specifying the `Literal` AST node that should be generated. A couple of failing inputs follow, as well as examples of syntactically valid literals which are nevertheless forbidden by the Java language specification — in this case we setup our grammar so that the a `Literal` node will still be generated, but will wrap an object describing the problem instead of an actual value.

A specificity of the assertion methods is that, by default, they actually parse the input twice and compare the two outcomes. The goal is to detect issues that could arise from improper state manipulation — when some program state is changed in a way that influences the parse, but the proper change is not registered on the `Log` object of the parse. If such a discrepancy occurs, the assertion will fail and report on it, listing out the two different outcomes. It will also suggests that the underlying issue might be parse state mishandling.

When using a testing framework such as TestNG or JUnit, a failed assertion generates an exception that is caught by the framework in order to display the location where the exception was thrown. However, in our case this includes a few methods from the `TestFixture` class. What the user is really interested in is the line in his test class where the failing assertion method (`success`, `failure`, etc) appears. To get rid of the noise, we actually rewrite the Java stack trace to remove these methods, so that the first line of the stack trace refers to the failed assertion. As the user may himself want to extend our test framework with helper methods, we let him specify how many additional methods should be peeled off the top of the stack trace.

6.6.4 Performance Tracing

After talking at length on how to make grammars correct, let's talk briefly about how to make them fast.

In Section 6.1.5, we talked about the overheads of the parsing framework itself. But there is no denying that the shape of a grammar greatly influences its performance.[31] Autumn is a backtracking parsing framework, and when backtracking occurs, it means that work was done speculatively — in other words, in vain. Anything that prevents backtracking is liable

[31] This is demonstrated for PEG in Section 6.1.2, and for ANTLR (and its two Java grammars) in Section 6.2.1.

to improve performance.

We saw that while Autumn (and PEG parsing in general) has exponential worst-case complexity, exponential behaviour just doesn't seem to happen in practice (Section 6.1.1). We also highlighted the only highly inefficient pattern of backtracking we ever found while writing grammars (Section 6.1.2). Finally, we saw that employing some custom memoization on the bottom (lexical) layer of grammars can speed up parsing dramatically (Section 6.2, Section 6.3).

What remains then is a more ordinary kind of inefficiency: maybe a grammar rule is called particularly often, or maybe many rules share a common prefix that is consequently re-tried many times at the same input position. Whenever necessary, these issues can be fixed by applying a memoization combinator (Section 6.1.4) or refactoring the grammar.

This leaves us with two major questions:

- How do I know I did not fall into one of the major performance pitfalls or miss one of the major optimization opportunities?

- Where are the remaining "ordinary" optimization opportunities?

These questions essentially boil down to "Where are the slow parsers?" Unfortunately, we can't resort to using a traditional tracing tool for this purpose. A tracing tool will, predictably, tell you that the code spends most of its time in `Parser#parse`, in a couple of `doparse` methods for common parsers (`Sequence`, `Choice`, ...), and in some primitive string comparison methods. But this doesn't tell us which parsers are the ones affected![32]

To nevertheless answer the question, Autumn offers a tracing mode, in which performance metrics are recorded for each parser in the grammar. In particular, we record the following three metrics:

- Cumulative total time: the total time spent in the parser (in its

[32]The problem is similar to the one we encountered with Java stack traces — methods don't uniquely identify parsers.

`parse` and `doparse` methods).[33]

- Cumulative self time: the total time spent in the parser minus the time spent in its sub-parsers.

- Number of invocations: the number of time the parser was invoked (this includes recursive invocations, as it well should).

Of the three, the number of invocations is the most useful. It tells you which parsers are *hot* and called often. If the hot parsers are not those you expect, investigate. These parsers are prime targets for memoization.

The cumulative self time is useful to ferret out the parsers whose logic itself is slow. You can additionally compute the average time spent per invocation by dividing this number by the number of invocations.

Finally, the cumulative total time is the least useful measure: the root parser of the grammar will always take 100% of the parse time. Nevertheless, this measure can be useful to find low-level parsers that take a lot of time by virtue of invoking a lot of different sub-parsers. It's also great for visualizing where the parse time is spent, for instance using flame graphs [30].

Tracing does have a cost: enabling our tracing code on our Java benchmark (using the Java grammar with lexical analysis emulation — see Section 6.2) slows it by a factor of 9. We should also note that, while we take care not to exclude time spent in the metrics-recording code, just the fact of actually running the metrics-recording code impacts the run time significantly, by means of cache pollution and missed JIT optimization opportunities. In fact, in our Java benchmark, the reported cumulative time of all parses is 5 times superior to the time that a non-tracing run would take.

6.7 Grammar Composition

Ideally, we would like to be able to reuse existing grammars, *i.e.*, to compose them. The term *"compose"* is very vague, and so we'll consider

[33] We take care to not double-count the time spent in recursive parsers. This could happen if we increment the total time spent by the time spent in a recursive call, then increment it again by the time spent in the non-recursive call, which overlaps with the recursive call.

multiple concrete scenarios in turn.

The first and simplest scenario involves *language embedding*. We simply want to define a new grammar that uses another grammar as a component. This is trivial in Autumn, it suffices to refer to parsers of the old grammar in the new grammar — the result being an extended parse graph. The old grammar is still usable on its own too.

There are actually two slightly different ways to perform this form of composition: by inheritance or by object composition. If the new grammar class extends the old grammar class, rules from the old and new grammar will share their `Tokens` object (cf. Section 6.3), and will share a simple `ws` whitespace rule (cf. Section 3.1.2). If the two languages have different lexical or whitespace requirements, then you should perform object composition instead: your new grammar includes a field that stores an instance of the old grammar, which is used to refer to the rules of the old grammar.

In a second, more complex scenario, we want to customize language embedding so that the embedded language may refer back to the host language. This is more difficult: as we have said previously, the parser graph is immutable.

We already hinted at the solution in Section 6.5.6: we can deep-copy the old grammar's parser graph using a `CopyVisitor`, and apply modifications during the copy. In our scenario, we would perform a copy of the embedded grammar, modifying it to refer back to some rules of the host grammar, and finally refer to the copy from the host grammar.

A grammar copy is in many ways similar to the object composition method in the first scenario: the copy will refer to the original embedded grammar's `Tokens` object and `ws` rule. As we explained in Section 6.5.6, this is not necessarily an issue, but remains something to be aware of.

You may wonder how we can make the embedded and host grammar refer to one another in this scenario. We use the same trick we used to handle recursion in monolithic grammars: use a `lazy` parser along with an indirect reference (cf. Section 3.1.3).

Finally, in a third scenario we have two *existing* grammars that we want to compose together so that they refer to each other. This scenario is very similar to the second one, except we are not the author of the second

grammar. Here, we want to create a third grammar that will serve as host for the composition. Then, we simply create modified copies of both grammars with a `CopyVisitor`. The new host grammar can also include *glue rules* to further extend the composition. We can even further compose such composed grammars together!

In closing, let's briefly discuss the theoretical properties of such compositions. Both PEGs and CFGs are closed under composition, and so are Autumn grammars. However, composing PEGs is liable to introduce *prefix capture*, while composing CFGs is liable to introduce *ambiguity*. See Section 2.1.2 for a discussion of ambiguity and Section 2.4.3 for a discussion of prefix capture, and how the two concepts relate to each other. We know that predicting prefix capture and ambiguity is undecidable in general [81]. Autumn, just like PEG, is liable to prefix capture. However, composition only tends to be problematic when pervasive mixing is attempted. For instance, trying to merge the expression syntax of two languages is going to be fraught with peril. In other cases, the result of a composition tends to be fairly predictable.

6.8 Conclusion

This chapter covered a lot of ground, from an extensive discussion of performance considerations in combinator parsing, to support for lexical analysis, debugging tracing, error-recovery and error-reporting strategies; as well as an examination of the possibilities opened up by parser reification and the built-in traversal and visitor facilities.

What we hope to have shown is that Autumn does not merely possess nice expressiveness properties (left-recursion, infix expression parsing, context-sensitivity) but is also a fully-featured practical parsing tool — one that is fast but also user-friendly, debuggable, and of course extensible.

Something we emphasized all along this chapter is that these features and improvements were not built *despite* Autumn's extensibility, but *thanks to it*. As the underlying extensible architecture has served us so well in tackling these challenges, we are confident it will scale well to new challenges and use cases we did not foresee.

Finally, this chapter goes into a lot of the nitty gritty details of parser engineering. It reports on a lot of knowledge that is often left unspoken or merely hinted at in conference hallways or internet comment sections, and very rarely finds its way to published papers. We are proud to have

given it the consideration it deserves, as we think there is a lot to be gained by disseminating it more broadly. For instance, we don't think that even the majority of parsing tool authors have a good handle on what makes a parser fast (or more pointedly, slow). This isn't due to incompetence, but merely ignorance. And admittedly, not everyone has the luxury of doing a PhD on parsing to acquire the relevant knowledge. But once acquired, this knowledge can be disseminated, and this chapter is our contribution to that endeavour.

Chapter 7

Conclusions and Future Directions

Simplicity and Flexibility

We started this thesis by setting an ambitious objective: finding the sweet spot between the simplicity and declarativeness of grammarware and the flexibility of ad-hoc parsing; and developing a well-engineered and easy to use solution in that space.

To achieve this goal, we outlined a series of capabilities which we believe a good solution should possess — ways to manifest both the flexibility and simplicity properties.

In this concluding chapter, we would simply like to go over these criteria and show how they were addressed in the thesis.

The flexibility capabilities:

1. The ability to extend the parsing system with new combinators.

2. The ability to create new primitive parsers.

   These two capabilities are easy to integrate in a top-down recursive-descent combinator parser approach. We explained how parsers are defined in Autumn in Section 3.2.

3. Full control over the generation of the syntax trees.

   Control over ASTs is realized through the manipulation of the value stack via a number of built-in AST-building parsers (cf. Section 3.3). The value stack itself is a form of parse state that is underpinned by the log mechanism presented in Section 5.6.

4. Allowance for a wide variety of context-sensitive features in the language's syntax and generated syntax trees.

   Context-sensitivity is the object of the whole of Chapter 5.

5. The ability to customize the error-reporting strategy and the reported error messages.

   We show how Autumn supports implementing error-reporting and error-recovery strategies in Section 6.4, where we also present a few such strategies.

6. The ability to compose independently-developed grammars.

   Grammar composition is the object of Section 6.7.

7. The reification (availability for programmatic inspection) of the grammar.

   The support for grammar reification is demonstrated in Section 6.5, in the form of parser walkers in visitors. We also show practical applications of these capabilities (the well-formedness check and the copy visitor).

8. Sufficient performance for parsers to be practically usable.

   As demonstrated by the performance comparison of Section 6.2.

The simplicity capabilities:

1. The grammar should be readable, in order to avoid maintaining a separate language description.

   The reader can judge of this by herself, by inspecting the introductory grammars of Section 3.1 and Section 3.3 as well as the

complete Java grammar listed in Appendix A.

2. The system should, as much as possible, be devoid of counter-intuitive pitfalls. In general, the *simple* way should be the *right* way.

   This is important concern and was an ongoing theme, but is perhaps most readily apparent in our treatment of the different infix expression parsing solutions of Chapter 4, and in our Section 6.5 efforts to find a friendly to let users extend existing parser visitors.

3. The system should cover, using a limited number of built-in primitives, most common parsing use-cases.

   The built-in parsers available in Autumn cover a strict superset of the PEG semantics. Noteworthy additions include the longest-match parser, lexical analysis emulation (Section 6.3), the bounded parser (Section 6.4.5), and the left-recursive and infix expression parsers of Chapter 4.

4. The system's principles should be simple and easy to understand so that it is simple to extend and customize.

   The basic concepts of Autumn were presented in Chapter 3, and most features introduced afterwards were simply built on top of this simple foundation. The major exception being the context-sensitive system, embodied by a log of undoable actions, which is introduced in Section 5.6.

5. The reification of the grammar should be easy to manipulate.

   Achieving this through the use of parser visitors and walkers is the object of Section 6.5.

We believe it fair to say that we have addressed every capability in the list in a way that is more than satisfactory. The result is a not just a prototype with a couple of nice theoretical properties, but a cohesive parsing tool that can be used in practical real-world scenarios and can compete with both state-of-the-art and battle-tested tools in both feature set and performance.

More importantly, we were able to do so without sacrificing too much

simplicity nor too much flexibility. The trade-off proved to be a bargain.

As a matter of fact, in the part of the design space that Autumn occupies, simplicity and flexibility are not so much at odds as they are in symbiosis. In many ways, Autumn is flexible *because* it is simple: advanced features can be built over a simple set of primitive operations and principles. Autumn's advanced capabilities can co-exist and even interact because their semantics is transparently defined in terms of this simple base layer. But Autumn is also simple *because* it is flexible: by leaving advanced capabilities out of the core layer, they can be tackled orthogonally without needing to revise the foundations on which the tool is built.

Future Direction

We certainly met the goals we set for our tool, but is everything perfect nonetheless? What can still be improved?

There are a couple of Autumn features that could not have been added by a user, most notably the support for recording the parser call stack (Section 6.6.1) and for tracing (Section 6.6.4). The implementations of both of these require code to be run at the start and at the end of the `Parser#parse` method. We could consider introducing some sort of *parser plugin* feature as a means to let user run code in these locations.

While Section 6.4 presents multiple possible error-reporting and error-recovery strategies, we have not actually tried them out in practice. The section does explain how we would go about validating these strategies (using a corpus of errors). Identifying effective error-handling strategies could lead us to abstracting and packaging them in Autumn, which would further simplify the task of users.

Another area of potential improvement is Autumn's support for lexical analysis (Section 6.3). While it simplifies things a good deal by alleviating the need to carefully order rules to reproduce longest-matching semantics and manually inserting memoization combinators, the performance comparison of Section 6.2 shows it still incurs a big performance penalty compared to the hand-optimized case. To quote ourselves:

> More research is required to determine if we can preserve the convenience of obtaining longest-match semantics by simply marking token rules, while closing the performance gap with the hand-optimized case.

We also think we can improve grammar composition (Section 6.7). While the underlying support for composition is solid, we only examined the idea fairly recently, and its practical deployment could probably benefit from some more research in order to reduce friction and minimize boilerplate.

Finally, Section 6.5.8 lists further possible applications of reification. We are particularly interested in pursuing automatic synthesis of context-object extractors, as well as making Autumn capable of generating parsers. Not only would the latter allow increased performance via reduced overheads and the possibility of domain-specific optimizations, it could also enable generating parsers in different languages, making Autumn usable beyond the JVM environment.

Final Word

We hope that this thesis has shown the potential of principled procedural parsing, and of striving to improve the status quo with better trade-offs — no matter the area.

We are proud of what was achieved. We find Autumn useful yet elegant, and many of the insights presented in this thesis valuable in practice. We hope that readers will come to share some of these feelings.

The work of these past couple of years was not *always* a joy, but it was most of time. Completing it, and having produced something interesting to show for it is all I could have asked for.

Bibliography

[1] Adams, M. D., and Might, M. Restricting Grammars with Tree Automata. *Proceedings of the ACM on Programming Languages (PACMPL) 1*, OOPSLA (2017), 82:1–82:25.

[2] Afroozeh, A., and Izmaylova, A. One Parser to Rule Them All. In *ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward!)* (2015), ACM, pp. 151–170.

[3] Aho, A. V., Lam, M. S., Sethi, R., and Ullman, J. D. *Compilers Principles, Techniques, and Tools (2nd Edition).* Addison-Wesley, 2006.

[4] Atkey, R. The Semantics of Parsing with Semantic Actions. In *IEEE/ACM Symposium on Logic in Computer Science (LICS)* (2012), IEEE, pp. 75–84.

[5] Aycock, J., and Horspool, R. N. Schrödinger's Token. *Software: Practice and Experience 31*, 8 (2001), 803–814.

[6] Bagge, A. H., and Lämmel, R. Walk Your Tree Any Way You Want. In *International Conference on Theory and Practice of Model Transformations, ICMT 2013* (2013), pp. 33–49.

[7] Bagwell, P. Ideal Hash Trees. Tech. Rep. LAMP-REPORT-2001-001, Ecole polytechnique fédérale de Lausanne, 2001.

[8] Becket, R., and Somogyi, Z. DCGs + Memoing = Packrat Parsing: But is it worth it? In *Practical Aspects of Declarative*

Languages (2008), Springer-Verlag, pp. 182–196.

[9] Béguet, E., and Jonnalagedda, M. Accelerating Parser Combinators With Macros. In *Scala Workshop (SCALA@ECOOP)* (2014), pp. 7–17.

[10] Birman, A., and Ullman, J. D. Parsing Algorithms With Backtrack. *Information and Control 23*, 1 (1973), 1–34.

[11] Bravenboer, M., Kalleberg, K. T., Vermaas, R., and Visser, E. Stratego/XT 0.17. A language and toolset for program transformation. *Science of Computer Programming 72*, 1-2 (2008), 52–70.

[12] Burge, W. H. *Recursive Programming Techniques*. The Systems Programming Series. Addison-Wesley, 1975.

[13] Burke, M. G., and Fisher, G. A. A Practical Method for LR and LL Syntactic Error Diagnosis and Recovery. *ACM Transactions on Programming Languages and Systems (TOPLAS) 9*, 2 (1987), 164–197.

[14] Chomsky, N. Three Models for the Description of Language. *IRE Transactions on Information Theory 2*, 3 (1956), 113–124.

[15] Chomsky, N. Formal Properties of Grammars. In *Handbook of Mathematical Psychology*. Wiley, 1963, ch. 12, pp. 360–363 and 367.

[16] Cocke, J., and Schwartz, J. T. Programming Languages and Their Compilers: Preliminary Notes. Tech. rep., Courant Institute of Mathemathical Sciences, 1970.

[17] Colmerauer, A. Metamorphosis Grammars. In *Natural Language Communication with Computers*. Springer-Verlag, 1978, pp. 133–188.

[18] Cox, R. Generating Good Syntax Errors, 2010. https://research.swtch.com/yyerror.

[19] de Jonge, M., Kats, L. C. L., Visser, E., and Söderberg, E. Natural and Flexible Error Recovery for Generated Modular

Language Environments. *ACM Transactions on Programming Languages and Systems (TOPLAS) 34*, 4 (2012), 15:1–15:50.

[20] Deber, J., Jota, R., Forlines, C., and Wigdor, D. How Much Faster is Fast Enough?: User Perception of Latency & Latency Improvements in Direct and Indirect Touch. In *33rd Annual ACM Conference on Human Factors in Computing Systems (CHI)* (2015), ACM, pp. 1827–1836.

[21] Degano, P., and Priami, C. Comparison of Syntactic Error Handling in LR Parsers. *Software: Practice and Experience 25*, 6 (1995), 657–679.

[22] Dijkstra, E. W. ALGOL 60 Translation : An ALGOL 60 Translator for the X1 and Making a Translator for ALGOL 60. Tech. Rep. MR 34/61, Stichting Mathematisch Centrum, 1961.

[23] Doenitz, M. The Parboiled Homepage, 2015. https://github.com/sirthias/parboiled.

[24] Earley, J. An Efficient Context-free Parsing Algorithm. *Communications of the ACM 13*, 2 (Feb 1970), 94–102.

[25] Ford, B. Packrat Parsing: a Practical Linear-Time Algorithm with Backtracking. Master's thesis, MIT, 2002.

[26] Ford, B. Packrat Parsing: Simple, Powerful, Lazy, Linear Time, Functional Pearl. In *ACM SIGPLAN International Conference on Functional Programming (ICFP)* (2002), ACM, pp. 36–47.

[27] Ford, B. Parsing Expression Grammars: A Recognition-Based Syntactic Foundation. *SIGPLAN Notices 39*, 1 (Jan 2004), 111–122.

[28] Frost, R., and Launchbury, J. Constructing Natural Language Interpreters in a Lazy Functional Language. *The Computer Journal 32*, 2 (1989), 108–121.

[29] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[30] Gregg, B. The Flame Graph. *Communications of the ACM 59*, 6 (2016), 48–57.

[31] Grimm, R. Better Extensibility Through Modular Syntax. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)* (2006), ACM, pp. 38–51.

[32] Grune, D., and Jacobs, C. J. *Parsing Techniques: A Practical Guide, p. 21–23*, 2nd ed. Springer-Verlag, 2008.

[33] Hedin, G. Reference Attributed Grammars. *Informatica (Slovenia) 24*, 3 (2000).

[34] Hutton, G. Higher-Order Functions for Parsing. *Journal of Functional Programming 2*, 3 (1992), 323–343.

[35] Hutton, G., and Meijer, E. Monadic Parsing in Haskell. *Journal of Functional Programming 8*, 4 (1998), 437–444.

[36] Ierusalimschy, R. A Text Pattern-Matching Tool Based on Parsing Expression Grammars. *Software: Practice and Experience 39*, 3 (2009), 221–258.

[37] ISO/IEC 14977:1996(E). EBNF Syntax Specification. Standard, International Organization for Standardization, 1996.

[38] Izmaylova, A., Afroozeh, A., and van der Storm, T. Practical, General Parser Combinators. In *ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation (PEPM)* (2016), PEPM '16, ACM, pp. 1–12.

[39] Jeffery, C. L. Generating LR Syntax Error Messages from Examples. *ACM Transactions on Programming Languages and Systems (TOPLAS) 25*, 5 (Sept. 2003), 631–640.

[40] Jim, T., and Mandelbaum, Y. A New Method for Dependent Parsing. In *European Conference on Programming Languages and Systems (ESOP)* (2011), Springer-Verlag, pp. 378–397.

[41] Jim, T., Mandelbaum, Y., and Walker, D. Semantics and Algorithms for Data-dependent Grammars. *SIGPLAN Notices 45*, 1 (2010), 417–430.

[42] Johnson, D. B. Finding All the Elementary Circuits of a Directed Graph. *SIAM Journal on Computing 4*, 1 (1975), 77–84.

[43] Johnson, S. C. Yacc: Yet Another Compiler-Compiler. Tech. rep., AT&T Bell Laboratories Murray Hill, 1976.

[44] Kallmeyer, L. *Parsing Beyond Context-Free Grammars*. Springer-Verlag, 2010.

[45] Kegler, J. Marpa, a Practical General Parser: The Recognizer, 2012.

[46] Kegler, J. Marpa and combinator parsing 2, 2018. http://jeffreykegler.github.io/Ocean-of-Awareness-blog/individual/2018/05/combinator2.html.

[47] Kegler, J. Parsing: a Timeline, 2018. https://jeffreykegler.github.io/personal/timeline_v3.

[48] Kipps, J., and Tomita, M. *Generalized LR Parsing*. Springer-Verlag, 1991.

[49] Klint, P., Lämmel, R., and Verhoef, C. Toward an Engineering Discipline for Grammarware. *ACM Transactions on Software Engineering Methodology 14*, 3 (2005), 331–380.

[50] Knuth, D. E. On the Translation of Languages From Left to Right. *Information and Control 8*, 6 (1965), 607–639.

[51] Knuth, D. E. Semantics of Context-Free Languages. *Mathematical Systems Theory 2*, 2 (1968), 127–145.

[52] Knuth, D. E. The genesis of attribute grammars. In *Attribute Grammars and Their Applications*. Springer-Verlag, 1990, pp. 1–12.

[53] Kuroda, S.-Y. Classes of Languages and Linear-Bounded Automata. *Information and Control 7*, 2 (1964), 207 – 223.

[54] Laurent, N. Prolog Served DRY, 2017. https://github.com/norswap/prolog-dry.

[55] Laurent, N. Red Shift: Procedural Shift-Reduce Parsing. In

ACM SIGPLAN International Conference on Software Language Engineering (SLE) (2017), pp. 38–42.

[56] Laurent, N. Autumn Source Code Repository, Thesis Version, 2019. https://github.com/norswap/autumn.

[57] Laurent, N., and Mens, K. Parsing Expression Grammars Made Practical. In *ACM SIGPLAN International Conference on Software Language Engineering (SLE)* (2015), ACM, pp. 167–172.

[58] Laurent, N., and Mens, K. Taming Context-sensitive Languages with Principled Stateful Parsing. In *ACM SIGPLAN International Conference on Software Language Engineering (SLE)* (2016), ACM, pp. 15–27.

[59] Laurent, N., and Mens, K. Taming Context-Sensitive Languages with Principled Stateful Parsing: Artifacts. *Software Language Engineering: Artifacts Track* (2016). https://github.com/ncellar/sle2016.

[60] Le Charlier, B. Languages et Traducteur (INGI2132) - Course Syllabus, 2012.

[61] Leijen, D., and Meijer, E. Parsec: Direct Style Monadic Parser Combinators for the Real World. Tech. Rep. UU-CS-2001-35, Department of Information and Computing Sciences, Utrecht University, 2001.

[62] Leo, J. M. I. M. A General Context-free Parsing Algorithm Running in Linear Time on Every LR(K) Grammar Without Using Lookahead. *Theoretical Computer Science 82*, 1 (1991), 165–176.

[63] Maidl, A. M., Mascarenhas, F., Medeiros, S., and Ierusalimschy, R. Error Reporting in Parsing Expression Grammars. *Science of Computer Programming 132*, P1 (Dec. 2016), 129–140.

[64] Medeiros, S., Mascarenhas, F., and Ierusalimschy, R. Left Recursion in Parsing Expression Grammars. *Science of Computer Programming 96*, P2 (Dec. 2014), 177–190.

[65] Mizushima, K., Maeda, A., and Yamaguchi, Y. Packrat

Parsers Can Handle Practical Grammars in Mostly Constant Space. In *ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE)* (2010), ACM, pp. 29–36.

[66] Moonen, L. Generating Robust Parsers Using Island Grammars. In *Working Conference on Reverse Engineering (WCRE)* (2001), IEEE.

[67] Moors, A., Piessens, F., and Odersky, M. Parser Combinators in Scala. Tech. Rep. CW491, Department of Computer Science, KU Leuven, Feb. 2008.

[68] Nilsson-Nyman, E., Ekman, T., and Hedin, G. Practical Scope Recovery Using Bridge Parsing. In *International Conference on Software Language Engineering (SLE)* (2008), pp. 95–113.

[69] Oliveira, B. C. d. S., and Cook, W. R. Extensibility for the Masses - Practical Extensibility with Object Algebras. In *European Conference on Object-Oriented Programming (ECOOP)* (2012), pp. 2–27.

[70] Oliveira, B. C. d. S., van der Storm, T., Loh, A., and Cook, W. R. Feature-Oriented Programming with Object Algebras. In *European Conference on Object-Oriented Programming (ECOOP)* (2013), Springer-Verlag, pp. 27–51.

[71] Parr, T., and Fisher, K. LL(*): The Foundation of the ANTLR Parser Generator. In *ACM SIGPLAN Conference on Programming Language Design and Implementation* (2011), ACM, pp. 425–436.

[72] Parr, T., Harwell, S., and Fisher, K. Adaptive LL(*) Parsing: The Power of Dynamic Analysis. In *ACM International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA)* (2014), ACM, pp. 579–598.

[73] Partridge, A., and Wright, D. Predictive Parser Combinators Need Four Values To Report Errors. *Journal of Functional Programming 6*, 2 (1996), 355–364.

[74] Pottier, F. Reachability and Error Diagnosis in LR(1) Parsers.

In *International Conference on Compiler Construction (CC)* (2016), ACM, pp. 88–98.

[75] Pratt, V. R. Top Down Operator Precedence. In *ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL)* (1973), ACM, pp. 41–51.

[76] Redziejowski, R. R. Parsing Expression Grammar As a Primitive Recursive-Descent Parser with Backtracking. *Fundamenta Informaticae - Special Issue on Concurrency Specification and Programming (CS&P) 79*, 3-4 (Aug. 2007), 513–524.

[77] Redziejowski, R. R. Mouse: From Parsing Expressions to a practical parser. In *CS&P Workshop 2* (2009), Warsaw University, pp. 514–525.

[78] Redziejowski, R. R. BITES instead of FIRST for Parsing Expression Grammar. *Fundamenta Informaticae 109*, 3 (2011), 323–337.

[79] Richter, H. Noncorrecting Syntax Error Recovery. *ACM Transactions on Programming Languages and Systems (TOPLAS) 7*, 3 (1985), 478–489.

[80] Rosenkrantz, D. J., and Stearns, R. E. Properties of Deterministic Top Down Grammars. In *ACM Symposium on Theory of Computing (STOC)* (1969), ACM, pp. 165–180.

[81] Schmitz, S. Modular Syntax Demands Verification. Tech. rep., LABORATOIRE I3S, UNIVERSITÉ DE NICE - SOPHIA ANTIPOLIS, 2006.

[82] Scott, E., and Johnstone, A. GLL Parsing. *Electronic Notes in Theoretical Computer Science 253*, 7 (Sept. 2010), 177–189.

[83] Seaton, C. A Programming Language Where the Syntax and Semantics Are Mutable at Runtime. Master's thesis, University of Bristol, 2007.

[84] Shipilëv, A. The Black Magic of (Java) Method Dispatch, 2015. https:

//shipilev.net/blog/2015/black-magic-method-dispatch/.

[85] Spiewak, D. Generalized Parser Combinators, 2010.

[86] Spivey, J. M., and Abrial, J. *The Z notation*. Prentice Hall, 1992.

[87] Stackexchange user Babou. Example of context-free grammar that triggers exponential behaviour without memoization in RD parsers.
https://cstheory.stackexchange.com/questions/22520/.

[88] Steindorfer, M. J., and Vinju, J. J. Optimizing Hash-array Mapped Tries for Fast and Lean Immutable JVM Collections. In *ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)* (2015), ACM, pp. 783–800.

[89] Sterling, L. S., and Shapiro, E. Y. *The Art of Prolog: Advanced Programming Techniques*. MIT Press, 1994.

[90] Tarjan, R. Depth-First Search and Linear Graph Algorithms. *SIAM Journal on Computing 1* (1972), 146–160.

[91] The Free Software Foundation. The GNU Bison Homepage, 2014. http://www.gnu.org/software/bison/.

[92] Thurston, A. D. *A Computer Language Transformation System Capable of Generalized Context-dependent Parsing*. PhD thesis, Queen's University, Kingston, Ontario, Canada, 2009.

[93] Thurston, A. D., and Cordy, J. R. A Backtracking LR Algorithm for Parsing Ambiguous Context-Dependent Languages. In *Conference of the Centre for Advanced Studies on Collaborative Research* (2006), pp. 39–53.

[94] Tomita, M. *Efficient Parsing for Natural Language: A Fast Algorithm for Practical Systems*. Kluwer, 1985.

[95] Torgersen, M. The Expression Problem Revisited. In *European Conference on Object-Oriented Programming (ECOOP)* (2004), pp. 123–143.

[96] Tratt, L. Parsing: The Solved Problem That Isn't, 2011. http://tratt.net/laurie/blog/entries/parsing_the_solved_problem_that_isnt.

[97] Tratt, L. grammars-v4, The ANTLR Grammar Repository, 2019. https://github.com/antlr/grammars-v4.

[98] van Binsbergen, L. T., Scott, E., and Johnstone, A. GLL Parsing with Flexible Combinators. In *ACM SIGPLAN International Conference on Software Language Engineering (SLE)* (2018), pp. 16–28.

[99] Van Wyk, E., and Schwerdfeger, A. Context-Aware Scanning for Parsing Extensible Languages. In *International Conference on Generative Programming and Component Engineering (GPCE)* (2007), ACM.

[100] Visser, E. Scannerless Generalized-LR Parsing. Tech. Rep. P9707, Programming Research Group, University of Amsterdam, 1997.

[101] Wadler, P. Monads for Functional Programming. In *International Spring School on Advanced Functional Programming* (1995), Springer-Verlag, pp. 24–52.

[102] Wadler, P., et al. The Expression Problem, 1998. Posted on the Java Genericity mailing list. http://homepages.inf.ed.ac.uk/wadler/papers/expression/expression.txt.

[103] Wang, Y., and Oliveira, B. C. d. S. The Expression Problem, Trivially! In *International Conference on Modularity (MODULARITY)* (2016), ACM, pp. 37–41.

[104] Warburton, R. Too Fast, Too Megamorphic: what influences method call performance in Java?, 2014. http://insightfullogic.com/2014/May/12/fast-and-megamorphic-what-influences-method-invoca/.

[105] Warth, A., Douglass, J. R., and Millstein, T. Packrat Parsers Can Support Left Recursion. In *ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation (PEPM)* (2008), ACM, pp. 103–110.

[106] WARTH, A., AND PIUMARTA, I. OMeta: an Object-Oriented Language for Pattern Matching. In *Symposium on Dynamic Languages (DLS)* (2007), ACM, pp. 11–19.

[107] WÜRTHINGER, T., WIMMER, C., WÖSS, A., STADLER, L., DUBOSCQ, G., HUMER, C., RICHARDS, G., SIMON, D., AND WOLCZKO, M. One VM to Rule Them All. In *ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software (Onward!)* (2013), ACM, pp. 187–204.

[108] ZAYTSEV, V. Formal Foundations for Semi-parsing. In *Software Evolution Week - IEEE Conference on Software Maintenance, Reengineering and Reverse Engineering (CSMR-WCRE)* (Feb. 2014), IEEE, pp. 313–317.

[109] ZENGER, M., AND ODERSKY, M. Independently Extensible Solutions To The Expression Problem. In *International Workshop on Foundations of Object-Oriented Languages (FOOL)* (2005).

Appendices

Appendix A

Autumn Java 8 Grammar

This appendix contains a complete listing of one of Autumn's Java 8 grammar, namely the one which uses Autumns's support for lexical analysis emulation (cf. Section 6.3). This grammar was used to obtain Autumn's performance measurements in Section 6.2.

```
1   package norswap.lang.java;
2
3   import norswap.autumn.DSL;
4   import norswap.autumn.StackAction;
5   import norswap.lang.java.ast.*;
6   import norswap.lang.java.ast.TypeDeclaration.Kind;
7   import norswap.utils.Pair;
8
9   import static java.util.Collections.emptyList;
10  import static norswap.lang.java.LexUtils.*;
11  import static norswap.lang.java.ast.BinaryOperator.*;
12  import static norswap.lang.java.ast.UnaryOperator.*;
13
14  public final class Grammar extends DSL
15  {
16    /// LEXICAL ======================================================================
17
18    // Whitespace --------------------------------------------------------------------
19
20    public rule space_char         = cpred(Character::isWhitespace);
21    public rule not_line           = seq(str("\n").not(), any);
22    public rule line_comment       = seq("//", not_line.at_least(0), str("\n").opt());
23
24    public rule not_comment_term    = seq(str("*/").not(), any);
25    public rule multi_comment       = seq("/*", not_comment_term.at_least(0), "*/");
26
27    { ws = choice(space_char, line_comment, multi_comment).at_least(0); }
```

```
28
29   // Keywords and Operators -----------------------------------------------------------
30
31   public rule _boolean       = word("boolean")      .token();
32   public rule _byte          = word("byte")         .token();
33   public rule _char          = word("char")         .token();
34   public rule _double        = word("double")       .token();
35   public rule _float         = word("float")        .token();
36   public rule _int           = word("int")          .token();
37   public rule _long          = word("long")         .token();
38   public rule _short         = word("short")        .token();
39   public rule _void          = word("void")         .token();
40   public rule _abstract      = word("abstract")     .token();
41   public rule _default       = word("default")      .token();
42   public rule _final         = word("final")        .token();
43   public rule _native        = word("native")       .token();
44   public rule _private       = word("private")      .token();
45   public rule _protected     = word("protected")    .token();
46   public rule _public        = word("public")       .token();
47   public rule _static        = word("static")       .token();
48   public rule _strictfp      = word("strictfp")     .token();
49   public rule _synchronized  = word("synchronized") .token();
50   public rule _transient     = word("transient")    .token();
51   public rule _volatile      = word("volatile")     .token();
52   public rule _assert        = word("assert")       .token();
53   public rule _break         = word("break")        .token();
54   public rule _case          = word("case")         .token();
55   public rule _catch         = word("catch")        .token();
56   public rule _class         = word("class")        .token();
57   public rule _const         = word("const")        .token();
58   public rule _continue      = word("continue")     .token();
59   public rule _do            = word("do")           .token();
60   public rule _else          = word("else")         .token();
61   public rule _enum          = word("enum")         .token();
62   public rule _extends       = word("extends")      .token();
63   public rule _finally       = word("finally")      .token();
64   public rule _for           = word("for")          .token();
65   public rule _goto          = word("goto")         .token();
66   public rule _if            = word("if")           .token();
67   public rule _implements    = word("implements")   .token();
68   public rule _import        = word("import")       .token();
69   public rule _interface     = word("interface")    .token();
70   public rule _instanceof    = word("instanceof")   .token();
71   public rule _new           = word("new")          .token();
72   public rule _package       = word("package")      .token();
73   public rule _return        = word("return")       .token();
74   public rule _super         = word("super")        .token();
75   public rule _switch        = word("switch")       .token();
76   public rule _this          = word("this")         .token();
77   public rule _throws        = word("throws")       .token();
78   public rule _throw         = word("throw")        .token();
79   public rule _try           = word("try")          .token();
80   public rule _while         = word("while")        .token();
81
82   // Names are taken from the javac8 lexer.
83   // Ordering matters when there are shared prefixes!
84
85   public rule BANG           = word("!")            .token();
```

```
86    public rule BANGEQ          = word("!=")          .token();
87    public rule PERCENT         = word("%")           .token();
88    public rule PERCENTEQ       = word("%=")          .token();
89    public rule AMP             = word("&")           .token();
90    public rule AMPAMP          = word("&&")          .token();
91    public rule AMPEQ           = word("&=")          .token();
92    public rule LPAREN          = word("(")           .token();
93    public rule RPAREN          = word(")")           .token();
94    public rule STAR            = word("*")           .token();
95    public rule STAREQ          = word("*=")          .token();
96    public rule PLUS            = word("+")           .token();
97    public rule PLUSPLUS        = word("++")          .token();
98    public rule PLUSEQ          = word("+=")          .token();
99    public rule COMMA           = word(",")           .token();
100   public rule SUB             = word("-")           .token();
101   public rule SUBSUB          = word("--")          .token();
102   public rule SUBEQ           = word("-=")          .token();
103   public rule EQ              = word("=")           .token();
104   public rule EQEQ            = word("==")          .token();
105   public rule QUES            = word("?")           .token();
106   public rule CARET           = word("^")           .token();
107   public rule CARETEQ         = word("^=")          .token();
108   public rule LBRACE          = word("{")           .token();
109   public rule RBRACE          = word("}")           .token();
110   public rule BAR             = word("|")           .token();
111   public rule BARBAR          = word("||")          .token();
112   public rule BAREQ           = word("|=")          .token();
113   public rule TILDE           = word("~")           .token();
114   public rule MONKEYS_AT      = word("@")           .token();
115   public rule DIV             = word("/")           .token();
116   public rule DIVEQ           = word("/=")          .token();
117   public rule GTEQ            = word(">=")          .token();
118   public rule LTEQ            = word("<=")          .token();
119   public rule LTLTEQ          = word("<<=")         .token();
120   public rule LTLT            = word("<<")          .token();
121   public rule GTGTEQ          = word(">>=")         .token();
122   public rule GTGTGTEQ        = word(">>>=")        .token();
123   public rule GT              = word(">")           .token();
124   public rule LT              = word("<")           .token();
125   public rule LBRACKET        = word("[")           .token();
126   public rule RBRACKET        = word("]")           .token();
127   public rule ARROW           = word("->")          .token();
128   public rule COL             = word(":")           .token();
129   public rule COLCOL          = word("::")          .token();
130   public rule SEMI            = word(";")           .token();
131   public rule DOT             = word(".")           .token();
132   public rule ELLIPSIS        = word("...")         .token();
133
134   // These two are not tokens, because they would cause issue with
135   // nested generic types.
136   // e.g. in List<List<String>>, you want ">>" to lex as [GT, GT]
137
138   public rule GTGT      = word(">>");
139   public rule GTGTGT    = word(">>>");
140
141   public rule _false    = word("false") .as_val(false)      .token();
142   public rule _true     = word("true")  .as_val(true)       .token();
143   public rule _null     = word("null")  .as_val(Null.NULL)  .token();
```

```
144
145    // Identifiers -------------------------------------------------------------------
146
147    public rule id_start    = cpred(Character::isJavaIdentifierStart);
148    public rule id_part     = cpred(c -> c != 0 && Character.isJavaIdentifierPart(c));
149
150    public rule iden = seq(id_start, id_part.at_least(0))
151        .push(with_string((p,xs,str) -> Identifier.mk(str)))
152        .word()
153        .token();
154
155    // Numerals - Common Parts -------------------------------------------------------
156
157    public rule underscore  = str("_");
158    public rule dlit        = str(".");
159    public rule hex_prefix  = choice("0x", "0X");
160    public rule underscores = underscore.at_least(0);
161    public rule digits1     = digit.sep(1, underscores);
162    public rule digits0     = digit.sep(0, underscores);
163    public rule hex_digits  = hex_digit.sep(1, underscores);
164    public rule hex_num     = seq(hex_prefix, hex_digits);
165
166    // Numerals - Floating Point -----------------------------------------------------
167
168    public rule hex_significand = choice(
169        seq(hex_prefix, hex_digits.opt(), dlit, hex_digits),
170        seq(hex_num, dlit.opt()));
171
172    public rule exp_sign_opt      = set("+-").opt();
173    public rule exponent          = seq(set("eE"), exp_sign_opt, digits1);
174    public rule binary_exponent   = seq(set("pP"), exp_sign_opt, digits1);
175    public rule float_suffix      = set("fFdD");
176    public rule float_suffix_opt  = float_suffix.opt();
177
178    public rule hex_float_lit =
179        seq(hex_significand, binary_exponent, float_suffix_opt);
180
181    public rule decimal_float_lit = choice(
182        seq(digits1, dlit, digits0, exponent.opt(), float_suffix_opt),
183        seq(dlit, digits1, exponent.opt(), float_suffix_opt),
184        seq(digits1, exponent, float_suffix_opt),
185        seq(digits1, exponent.opt(), float_suffix));
186
187    public rule float_literal = choice(hex_float_lit, decimal_float_lit)
188        .push(with_string((p,xs,str) -> parse_floating(str).unwrap()))
189        .token();
190
191    // Numerals - Integral -----------------------------------------------------------
192
193    public rule bit             = set("01");
194    public rule binary_prefix   = choice("0b", "0B");
195
196    public rule binary_num =
197        seq(binary_prefix, bit.at_least(1).sep(1, underscores));
198
199    public rule octal_num =
200        seq("0", seq(underscores, octal_digit).at_least(1));
201
```

```
202    public rule decimal_num =
203        choice("0", digits1);
204
205    public rule integer_num =
206        choice(hex_num, binary_num, octal_num, decimal_num);
207
208    public rule integer_literal = seq(integer_num, set("lL").opt())
209        .push(with_string((p,xs,str) -> parse_integer(str).unwrap()))
210        .token();
211
212    // Characters and Strings ----------------------------------------------------------
213
214    public rule octal_code_3    = seq(range('0', '3'), octal_digit, octal_digit);
215    public rule octal_code_2    = seq(octal_digit, octal_digit.opt());
216    public rule octal_code      = choice(octal_code_3, octal_code_2);
217    public rule unicode_code    = seq(str("u").at_least(1), hex_digit.repeat(4));
218    public rule escape_suffix   = choice(set("btnfr\"'\\"), octal_code, unicode_code);
219    public rule escape          = seq("\\", escape_suffix);
220    public rule naked_char      = choice(escape, seq(set("'\\\n\r").not(), any));
221    public rule nake_str_char   = choice(escape, seq(set("\"\\\n\r").not(), any));
222
223    public rule char_literal = seq("'", naked_char, "'")
224        .push(with_string((p,xs,str) -> parse_char(str).unwrap()))
225        .token();
226
227    public rule string_literal = seq("\"", nake_str_char.at_least(0), "\"")
228        .push(with_string((p,xs,str) -> parse_string(str).unwrap()))
229        .token();
230
231    // Literal -------------------------------------------------------------------------
232
233    public rule literal = token_choice(
234            integer_literal, string_literal, _null,
235            float_literal, _true, _false, char_literal)
236        .word()
237        .push(xs -> Literal.mk(xs[0]));
238
239    //// LAZY FORWARD REFS =============================================================
240
241    public rule _stmt =
242        lazy(() -> this.stmt);
243
244    public rule _expr =
245        lazy(() -> this.expr);
246
247    public rule _block =
248        lazy(() -> this.block);
249
250    /// ANNOTATIONS ====================================================================
251
252    public rule annotation_element = choice(
253        lazy(() -> this.ternary_expr),
254        lazy(() -> this.annotation_element_list),
255        lazy(() -> this.annotation));
256
257    public rule annotation_inner_list =
258        lazy(() -> this.annotation_element).sep_trailing(0, COMMA);
259
```

```
260    public rule annotation_element_list =
261        seq(LBRACE, annotation_inner_list, RBRACE)
262        .push(xs -> AnnotationElementList.mk(list(xs)));
263
264    public rule annotation_element_pair =
265        seq(iden, EQ, annotation_element)
266        .push(xs -> new Pair<Identifier, AnnotationElement>($(xs,0), $(xs,1)));
267
268    public rule normal_annotation_suffix =
269        seq(LPAREN, annotation_element_pair.sep(1, COMMA), RPAREN)
270        .push(with_parse((p,xs) -> NormalAnnotation.mk($(p.stack.pop()), list(xs))));
271
272    public rule single_element_annotation_suffix =
273        seq(LPAREN, annotation_element, RPAREN)
274        .collect().lookback(1).push(xs -> SingleElementAnnotation.mk($(xs,0), $(xs,1)));
275
276    public rule marker_annotation_suffix =
277        seq(LPAREN, RPAREN).opt()
278         .collect().lookback(1).push(xs -> MarkerAnnotation.mk($(xs,0)));
279
280    public rule annotation_suffix = choice(
281        normal_annotation_suffix,
282        single_element_annotation_suffix,
283        marker_annotation_suffix);
284
285    public rule qualified_iden =
286        iden.sep(1, DOT)
287        .collect().as_list(Identifier.class);
288
289    public rule annotation =
290        seq(MONKEYS_AT, qualified_iden, annotation_suffix);
291
292    public rule annotations =
293        annotation.at_least(0)
294        .collect().as_list(TAnnotation.class);
295
296    /// TYPES ==========================================================================
297
298    public rule basic_type =
299        token_choice(_byte, _short, _int, _long, _char, _float, _double, _boolean, _void)
300        .push(with_string(
301            (p,xs,str) -> BasicType.valueOf("_" + trim_trailing_whitespace(str))));
302
303    public rule primitive_type =
304        seq(annotations, basic_type)
305        .push(xs -> PrimitiveType.mk($(xs,0), $(xs,1)));
306
307    public rule extends_bound =
308        seq(_extends, lazy(() -> this.type))
309        .push(xs -> ExtendsBound.mk($(xs,0)));
310
311    public rule super_bound =
312        seq(_super, lazy(() -> this.type))
313        .push(xs -> SuperBound.mk($(xs,0)));
314
315    public rule type_bound =
316        choice(extends_bound, super_bound).maybe();
317
```

```
318    public rule wildcard =
319        seq(annotations, QUES, type_bound)
320        .push(xs -> Wildcard.mk($(xs,0), $(xs,1)));
321
322    public rule opt_type_args =
323        seq(LT, choice(lazy(() -> this.type), wildcard).sep(0, COMMA), GT).opt()
324        .collect().as_list(TType.class);
325
326    public rule class_type_part =
327        seq(annotations, iden, opt_type_args)
328        .push(xs -> ClassTypePart.mk($(xs, 0), $(xs, 1), $(xs, 2)));
329
330    public rule class_type =
331        class_type_part.sep(1, DOT)
332        .push(xs -> ClassType.mk(list(xs)));
333
334    public rule stem_type =
335        choice(primitive_type, class_type);
336
337    public rule dim =
338        seq(annotations, seq(LBRACKET, RBRACKET))
339        .push(xs -> Dimension.mk($(xs,0)));
340
341    public rule dims =
342        dim.at_least(0)
343        .collect().as_list(Dimension.class);
344
345    public rule dims1 =
346        dim.at_least(1)
347        .collect().as_list(Dimension.class);
348
349    public rule type_dim_suffix =
350        dims1
351        .collect().lookback(1).push(xs -> ArrayType.mk($(xs,0), $(xs,1)));
352
353    public rule type =
354        seq(stem_type, type_dim_suffix.opt());
355
356    public rule type_union_syntax =
357        lazy(() -> this.type).sep(1, AMP);
358
359    public rule type_union =
360        type_union_syntax
361        .collect().as_list(TType.class);
362
363    public rule type_bounds =
364        seq(_extends, type_union_syntax).opt()
365        .collect().as_list(TType.class);
366
367    public rule type_param =
368        seq(annotations, iden, type_bounds)
369        .push(xs -> TypeParameter.mk($(xs,0), $(xs,1), $(xs,2)));
370
371    public rule type_params =
372        seq(LT, type_param.sep(0, COMMA), GT).opt()
373        .collect().as_list(TypeParameter.class);
374
375    /// EXPRESSIONS ==================================================================
```

```
376
377     // Initializers -------------------------------------------------------------------
378
379     public rule var_init =
380         choice(_expr, lazy(() -> this.array_init));
381
382     public rule array_init =
383         seq(LBRACE, var_init.sep_trailing(0, COMMA), RBRACE)
384         .push(xs -> ArrayInitializer.mk(list(xs)));
385
386     // Array Constructor --------------------------------------------------------------
387
388     public rule dim_expr =
389         seq(annotations, LBRACKET, _expr, RBRACKET)
390         .push(xs -> DimExpression.mk($(xs,0), $(xs,1)));
391
392     public rule dim_exprs =
393         dim_expr.at_least(1)
394         .collect().as_list(DimExpression.class);
395
396     public rule dim_expr_array_creator =
397         seq(stem_type, dim_exprs, dims)
398         .push(xs -> ArrayConstructorCall.mk($(xs,0), $(xs,1), $(xs,2), null));
399
400     public rule init_array_creator =
401         seq(stem_type, dims1, array_init)
402         .push(xs -> ArrayConstructorCall.mk($(xs,0), emptyList(), $(xs,1), $(xs,2)));
403
404     public rule array_ctor_call =
405         seq(_new, choice(dim_expr_array_creator, init_array_creator));
406
407     // Lambda Expression --------------------------------------------------------------
408
409     public rule lambda = lazy(() ->
410         seq(this.lambda_params, ARROW, choice(this.block, this.expr)))
411         .push(xs -> Lambda.mk($(xs,0), $(xs,1)));
412
413     // Expression - Primary -----------------------------------------------------------
414
415     public rule args =
416         seq(LPAREN, _expr.sep(0, COMMA), RPAREN)
417         .collect().as_list(Expression.class);
418
419     public rule par_expr =
420         seq(LPAREN, _expr, RPAREN)
421         .push(xs -> ParenExpression.mk($(xs,0)));
422
423     public rule ctor_call =
424         seq(_new, opt_type_args, stem_type, args, lazy(() -> this.type_body).maybe())
425         .push(xs -> ConstructorCall.mk($(xs,0), $(xs,1), $(xs,2), $(xs,3)));
426
427     public rule new_ref_suffix =
428         _new
429         .collect().lookback(2).push(xs -> NewReference.mk($(xs,0), $(xs,1)));
430
431     public rule method_ref_suffix =
432         iden
433         .collect().lookback(2)
```

```
434         .push(xs -> TypeMethodReference.mk($(xs,0), $(xs,1), $(xs,2)));
435 
436     public rule ref_suffix =
437         seq(COLCOL, opt_type_args, choice(new_ref_suffix, method_ref_suffix));
438 
439     public rule class_expr_suffix =
440         seq(DOT, _class)
441         .collect().lookback(1).push(xs -> ClassExpression.mk($(xs,0)));
442 
443     public rule type_suffix_expr =
444         seq(type, choice(ref_suffix, class_expr_suffix));
445 
446     public rule iden_or_method_expr =
447         seq(iden, args.maybe())
448         .push(xs -> $(xs,1) == null
449             ? $(xs,
450             : MethodCall.mk(null, list(), $(xs,0), $(xs,1)));
451 
452     public rule this_expr =
453         seq(_this, args.maybe())
454         .push(xs -> $(xs,0) == null ? This.mk() : ThisCall.mk($(xs,0)));
455 
456     public rule super_expr =
457         seq(_super, args.maybe())
458         .push(xs -> $(xs,0) == null ? Super.mk() : SuperCall.mk($(xs,0)));
459 
460     public rule primary_expr = choice(
461         lambda, par_expr, array_ctor_call, ctor_call, type_suffix_expr,
462         iden_or_method_expr, this_expr, super_expr, literal);
463 
464     // Expression - Postfix & Prefix --------------------------------------------------
465 
466     public rule prefix_op = choice(
467         PLUSPLUS    .as_val(PREFIX_INCREMENT),
468         SUBSUB      .as_val(PREFIX_DECREMENT),
469         PLUS        .as_val(UNARY_PLUS),
470         SUB         .as_val(UNARY_MINUS),
471         TILDE       .as_val(BITWISE_COMPLEMENT),
472         BANG        .as_val(LOGICAL_COMPLEMENT));
473 
474     public rule postfix_expr = left_expression()
475         .left(primary_expr)
476         .suffix(seq(DOT, opt_type_args, iden, args),
477             xs -> MethodCall.mk($(xs,0), $(xs,1), $(xs,2), $(xs,3)))
478         .suffix(seq(DOT, iden),
479             xs -> DotIden.mk($(xs,0), $(xs,1)))
480         .suffix(seq(DOT, _this),
481             xs -> UnaryExpression.mk(DOT_THIS, $(xs,0)))
482         .suffix(seq(DOT, _super),
483             xs -> UnaryExpression.mk(DOT_SUPER, $(xs,0)))
484         .suffix(seq(DOT, ctor_call),
485             xs -> DotNew.mk($(xs,0), $(xs,1)))
486         .suffix(seq(LBRACKET, _expr, RBRACKET),
487             xs -> ArrayAccess.mk($(xs,0), $(xs,1)))
488         .suffix(PLUSPLUS,
489             xs -> UnaryExpression.mk(POSTFIX_INCREMENT, $(xs,0)))
490         .suffix(SUBSUB,
491             xs -> UnaryExpression.mk(POSTFIX_DECREMENT, $(xs,0)))
```

```
492         .suffix(seq(COLCOL, opt_type_args, iden),
493             xs -> BoundMethodReference.mk($(xs,0), $(xs,1), $(xs,2)))
494         .get();
495
496     public rule prefix_expr = recursive(self -> choice(
497         seq(prefix_op, self)
498             .push(xs -> UnaryExpression.mk($(xs,0), $(xs,1))),
499         seq(LPAREN, type_union, RPAREN, self)
500             .push(xs -> Cast.mk($(xs,0), $(xs,1))),
501         postfix_expr));
502
503     // Expression - Binary ------------------------------------------------------------
504
505     StackAction.Push binary_push =
506         xs -> BinaryExpression.mk($(xs,1), $(xs,0), $(xs,2));
507
508     public rule mult_op = choice(
509         STAR        .as_val(MULTIPLY),
510         DIV         .as_val(DIVIDE),
511         PERCENT     .as_val(REMAINDER));
512
513     public rule add_op = choice(
514         PLUS        .as_val(ADD),
515         SUB         .as_val(SUBTRACT));
516
517     public rule shift_op = choice(
518         LTLT        .as_val(LEFT_SHIFT),
519         GTGTGT      .as_val(UNSIGNED_RIGHT_SHIFT),
520         GTGT        .as_val(RIGHT_SHIFT));
521
522     public rule order_op = choice(
523         LT          .as_val(LESS_THAN),
524         LTEQ        .as_val(LESS_THAN_EQUAL),
525         GT          .as_val(GREATER_THAN),
526         GTEQ        .as_val(GREATER_THAN_EQUAL));
527
528     public rule eq_op = choice(
529         EQEQ        .as_val(EQUAL_TO),
530         BANGEQ      .as_val(NOT_EQUAL_TO));
531
532     public rule assignment_op = choice(
533         EQ          .as_val(ASSIGNMENT),
534         PLUSEQ      .as_val(ADD_ASSIGNMENT),
535         SUBEQ       .as_val(SUBTRACT_ASSIGNMENT),
536         STAREQ      .as_val(MULTIPLY_ASSIGNMENT),
537         DIVEQ       .as_val(DIVIDE_ASSIGNMENT),
538         PERCENTEQ   .as_val(REMAINDER_ASSIGNMENT),
539         LTLTEQ      .as_val(LEFT_SHIFT_ASSIGNMENT),
540         GTGTEQ      .as_val(RIGHT_SHIFT_ASSIGNMENT),
541         GTGTGTEQ    .as_val(UNSIGNED_RIGHT_SHIFT_ASSIGNMENT),
542         AMPEQ       .as_val(AND_ASSIGNMENT),
543         CARETEQ     .as_val(XOR_ASSIGNMENT),
544         BAREQ       .as_val(OR_ASSIGNMENT));
545
546     public rule mult_expr = left_expression()
547         .operand(prefix_expr)
548         .infix(mult_op, binary_push).get();
549
```

```
550   public rule add_expr = left_expression()
551       .operand(mult_expr)
552       .infix(add_op, binary_push).get();
553
554   public rule shift_expr = left_expression()
555       .operand(add_expr)
556       .infix(shift_op, binary_push).get();
557
558   public rule order_expr = left_expression()
559       .operand(shift_expr)
560       .suffix(seq(_instanceof, type),
561           xs -> InstanceOf.mk($(xs,0), $(xs,1)))
562       .infix(order_op, binary_push)
563       .get();
564
565   public rule eq_expr = left_expression()
566       .operand(order_expr)
567       .infix(eq_op, binary_push).get();
568
569   public rule binary_and_expr = left_expression()
570       .operand(eq_expr)
571       .infix(AMP.as_val(AND), binary_push).get();
572
573   public rule xor_expr = left_expression()
574       .operand(binary_and_expr)
575       .infix(CARET.as_val(XOR), binary_push).get();
576
577   public rule binary_or_expr = left_expression()
578       .operand(xor_expr)
579       .infix(BAR.as_val(OR), binary_push).get();
580
581   public rule conditional_and_expr = left_expression()
582       .operand(binary_or_expr)
583       .infix(AMPAMP.as_val(CONDITIONAL_AND), binary_push).get();
584
585   public rule conditional_or_expr = left_expression()
586       .operand(conditional_and_expr)
587       .infix(BARBAR.as_val(CONDITIONAL_OR), binary_push).get();
588
589   public rule ternary_expr = right_expression()
590       .operand(conditional_or_expr)
591       .infix(seq(QUES, _expr, COL),
592           xs -> TernaryExpression.mk($(xs,0), $(xs,1), $(xs,2)))
593       .get();
594
595   public rule expr = right_expression()
596       .operand(ternary_expr)
597       .infix(assignment_op, binary_push).get();
598
599   /// MODIFIERS ======================================================================
600
601   public rule keyword_modifier = token_choice(
602           _public, _protected, _private, _abstract, _static, _final, _synchronized,
603           _native, _strictfp, _default, _transient, _volatile)
604       .push(with_string(
605           (p,xs,str) -> Keyword.valueOf("_" + trim_trailing_whitespace(str))));
606
607   public rule modifier =
```

```
608         choice(annotation, keyword_modifier);
609 
610     public rule modifiers =
611         modifier.at_least(0)
612         .collect().as_list(Modifier.class);
613 
614     /// PARAMETERS ====================================================================
615 
616     public rule this_parameter_qualifier =
617         seq(iden, DOT).at_least(0)
618         .collect().as_list(String.class);
619 
620     public rule this_param_suffix =
621         seq(this_parameter_qualifier, _this)
622         .collect().lookback(2)
623         .push(xs -> ThisParameter.mk($(xs,0), $(xs,1), $(xs,2)));
624 
625     public rule iden_param_suffix =
626         seq(iden, dims)
627         .collect().lookback(2)
628         .push(xs -> IdenParameter.mk($(xs,0), $(xs,1), $(xs,2), $(xs,3)));
629 
630     public rule variadic_param_suffix =
631         seq(annotations, ELLIPSIS, iden)
632         .collect().lookback(2)
633         .push(xs -> VariadicParameter.mk($(xs,0), $(xs,1), $(xs,2), $(xs,3)));
634 
635     public rule formal_param_suffix =
636         choice(iden_param_suffix, this_param_suffix, variadic_param_suffix);
637 
638     public rule formal_param =
639         seq(modifiers, type, formal_param_suffix);
640 
641     public rule formal_params =
642         seq(LPAREN, formal_param.sep(0, COMMA), RPAREN)
643         .push(xs -> FormalParameters.mk(list()));
644 
645     public rule untyped_params =
646         seq(LPAREN, iden.sep(1, COMMA), RPAREN)
647         .push(xs -> UntypedParameters.mk(list()));
648 
649     public rule single_param =
650         iden
651         .push(xs -> UntypedParameters.mk(list(xs)));
652 
653     public rule lambda_params =
654         choice(formal_params, untyped_params, single_param);
655 
656     /// NON-TYPE DECLARATIONS ==========================================================
657 
658     public rule var_declarator_id =
659         seq(iden, dims)
660         .push(xs -> VarDeclaratorID.mk($(xs,0), $(xs,1)));
661 
662     public rule var_declarator =
663         seq(var_declarator_id, seq(EQ, var_init).maybe())
664         .push(xs -> VarDeclarator.mk($(xs,0), $(xs,1)));
665 
```

```
666     public rule var_declarators =
667         var_declarator.sep(1, COMMA)
668         .collect().as_list(VarDeclarator.class);
669
670     public rule var_decl_suffix_no_semi =
671         seq(type, var_declarators)
672         .collect().lookback(1)
673         .push(xs -> VarDeclaration.mk($(xs,0), $(xs,1), $(xs,2)));
674
675     public rule var_decl_suffix =
676         seq(var_decl_suffix_no_semi, SEMI);
677
678     public rule var_decl =
679         seq(modifiers, var_decl_suffix);
680
681     public rule throws_clause =
682         seq(_throws, type.sep(1, COMMA)).opt()
683         .collect().as_list(TType.class);
684
685     public rule block_or_semi =
686         choice(_block, SEMI.as_val(null));
687
688     public rule method_decl_suffix =
689         seq(type_params, type, iden, formal_params, dims, throws_clause, block_or_semi)
690         .collect().lookback(1)
691         .push(xs -> MethodDeclaration.mk(
692             $(xs,0), $(xs,1), $(xs,2), $(xs,3), $(xs,4), $(xs,5), $(xs,6), $(xs,7)));
693
694     public rule constructor_decl_suffix =
695         seq(type_params, iden, formal_params, throws_clause, _block)
696         .collect().lookback(1)
697         .push(xs -> ConstructorDeclaration.mk(
698             $(xs,0), $(xs,1), $(xs,2), $(xs,3), $(xs,4), $(xs,5)));
699
700     public rule init_block =
701         seq(_static.as_bool(), _block)
702         .push(xs -> InitBlock.mk($(xs,0), $(xs,1)));
703
704     /// TYPE DECLARATIONS ==============================================================
705
706     // Common -------------------------------------------------------------------------
707
708     public rule extends_clause =
709         seq(_extends, type.sep(0, COMMA)).opt()
710         .collect().as_list(TType.class);
711
712     public rule implements_clause =
713         seq(_implements, type.sep(0, COMMA)).opt()
714         .collect().as_list(TType.class);
715
716     public rule type_sig =
717         seq(iden, type_params, extends_clause, implements_clause);
718
719     public rule class_modifierized_decl = seq(
720         modifiers,
721         choice(
722             var_decl_suffix,
723             method_decl_suffix,
```

```
724            constructor_decl_suffix,
725            lazy(() -> this.type_decl_suffix)));
726
727     public rule class_body_decl =
728         choice(class_modifierized_decl, init_block, SEMI);
729
730     public rule class_body_decls =
731         class_body_decl.at_least(0)
732         .collect().as_list(Declaration.class);
733
734     public rule type_body =
735         seq(LBRACE, class_body_decls, RBRACE);
736
737     // Enum ----------------------------------------------------------------------------
738
739     public rule enum_constant =
740         seq(annotations, iden, args.maybe(), type_body.maybe())
741         .push(xs -> EnumConstant.mk($(xs,0), $(xs,1), $(xs,2), $(xs,3)));
742
743     public rule enum_class_decls =
744         seq(SEMI, class_body_decl.at_least(0)).opt();
745
746     public rule enum_constants =
747         enum_constant.sep(1, COMMA).opt();
748
749     public rule enum_body =
750         seq(LBRACE, enum_constants, enum_class_decls, RBRACE)
751         .collect().as_list(Declaration.class);
752
753     public rule enum_decl_suffix =
754         seq(_enum, type_sig, enum_body)
755         .collect().lookback(1)
756         .push(xs -> TypeDeclaration.mk(Kind.ENUM,
757             $(xs,0), $(xs,1), $(xs,2), $(xs,3), $(xs,4), $(xs,5)));
758
759     // Annotations ---------------------------------------------------------------------
760
761     public rule annot_default_clause =
762         seq(_default, annotation_element)
763         .push(xs -> $(xs,0));
764
765     public rule annot_elem_decl =
766         seq(modifiers, type, iden, LPAREN, RPAREN, dims,
767             annot_default_clause.maybe(), SEMI)
768         .push(xs -> AnnotationElementDeclaration.mk(
769             $(xs,0), $(xs,1), $(xs,2), $(xs,3), $(xs,4)));
770
771     public rule annot_body_decls =
772         choice(annot_elem_decl, class_body_decl).at_least(0)
773         .collect().as_list(Declaration.class);
774
775     public rule annotation_decl_suffix =
776         seq(MONKEYS_AT, _interface, type_sig, LBRACE, annot_body_decls, RBRACE)
777         .collect().lookback(1)
778         .push(xs -> TypeDeclaration.mk(Kind.ANNOTATION,
779             $(xs,0), $(xs,1), $(xs,2), $(xs,3), $(xs,4), $(xs,5)));
780
781     //// ------------------------------------------------------------------------------
```

```
782
783    public rule class_decl_suffix =
784        seq(_class, type_sig, type_body)
785        .collect().lookback(1)
786        .push(xs -> TypeDeclaration.mk(Kind.CLASS,
787            $(xs,0), $(xs,1), $(xs,2), $(xs,3), $(xs,4), $(xs,5)));
788
789    public rule interface_declaration_suffix =
790        seq(_interface, type_sig, type_body)
791        .collect().lookback(1)
792        .push(xs -> TypeDeclaration.mk(Kind.INTERFACE,
793            $(xs,0), $(xs,1), $(xs,2), $(xs,3), $(xs,4), $(xs,5)));
794
795    public rule type_decl_suffix = choice(
796        class_decl_suffix,
797        interface_declaration_suffix,
798        enum_decl_suffix,
799        annotation_decl_suffix);
800
801    public rule type_decl =
802        seq(modifiers, type_decl_suffix);
803
804    public rule type_decls =
805        choice(type_decl, SEMI).at_least(0)
806        .collect().as_list(Declaration.class);
807
808    /// STATEMENTS ===================================================================
809
810    public rule if_stmt =
811        seq(_if, par_expr, _stmt, seq(_else, _stmt).maybe())
812        .push(xs -> IfStatement.mk($(xs,0), $(xs,1), $(xs,2)));
813
814    public rule expr_stmt_list =
815        expr.sep(0, COMMA)
816        .collect().as_list(Statement.class);
817
818    public rule for_init_decl =
819        seq(modifiers, var_decl_suffix_no_semi)
820        .collect().as_list(Statement.class);
821
822    public rule for_init =
823        choice(for_init_decl, expr_stmt_list);
824
825    public rule basic_for_paren_part =
826        seq(for_init, SEMI, expr.maybe(), SEMI, expr_stmt_list.opt());
827
828    public rule basic_for_stmt =
829        seq(_for, LPAREN, basic_for_paren_part, RPAREN, _stmt)
830        .push(xs -> BasicForStatement.mk($(xs,0), $(xs,1), $(xs,2), $(xs,3)));
831
832    public rule for_val_decl =
833        seq(modifiers, type, var_declarator_id, COL, expr);
834
835    public rule enhanced_for_stmt =
836        seq(_for, LPAREN, for_val_decl, RPAREN, _stmt)
837        .push(xs ->
838            EnhancedForStatement.mk($(xs,0), $(xs,1), $(xs,2), $(xs,3), $(xs,4)));
839
```

```
840     public rule while_stmt =
841         seq(_while, par_expr, _stmt)
842         .push(xs -> WhileStatement.mk($(xs,0), $(xs,1)));
843
844     public rule do_while_stmt =
845         seq(_do, _stmt, _while, par_expr, SEMI)
846         .push(xs -> DoWhileStatement.mk($(xs,0), $(xs,1)));
847
848     public rule catch_parameter_types =
849         type.sep(0, BAR)
850         .collect().as_list(TType.class);
851
852     public rule catch_parameter =
853         seq(modifiers, catch_parameter_types, var_declarator_id);
854
855     public rule catch_clause =
856         seq(_catch, LPAREN, catch_parameter, RPAREN, _block)
857         .push(xs -> CatchClause.mk($(xs,0), $(xs,1), $(xs,2), $(xs,3)));
858
859     public rule catch_clauses =
860         catch_clause.at_least(0)
861         .collect().as_list(CatchClause.class);
862
863     public rule finally_clause =
864         seq(_finally, _block);
865
866     public rule resource =
867         seq(modifiers, type, var_declarator_id, EQ, expr)
868         .push(xs -> TryResource.mk($(xs,0), $(xs,1), $(xs,2), $(xs,3)));
869
870     public rule resources =
871         seq(LPAREN, resource.sep(1, SEMI), RPAREN).opt()
872         .collect().as_list(TryResource.class);
873
874     public rule try_stmt =
875         seq(_try, resources, _block, catch_clauses, finally_clause.maybe())
876         .push(xs -> TryStatement.mk($(xs,0), $(xs,1), $(xs,2), $(xs,3)));
877
878     public rule default_label =
879         seq(_default, COL)
880         .push(xs -> DefaultLabel.mk());
881
882     public rule case_label =
883         seq(_case, expr, COL)
884         .push(xs -> CaseLabel.mk($(xs,0)));
885
886     public rule switch_label =
887         choice(case_label, default_label);
888
889     public rule switch_clause =
890         seq(switch_label, lazy(() -> this.statements))
891         .push(xs -> SwitchClause.mk($(xs,0), $(xs,1)));
892
893     public rule switch_stmt =
894         seq(_switch, par_expr, LBRACE, switch_clause.at_least(0), RBRACE)
895         .push(xs -> SwitchStatement.mk($(xs,0), list(1, xs)));
896
897     public rule synchronized_stmt =
```

```
898         seq(_synchronized, par_expr, _block)
899         .push(xs -> SynchronizedStatement.mk($(xs,0), $(xs,1)));
900
901     public rule return_stmt =
902         seq(_return, expr.maybe(), SEMI)
903         .push(xs -> ReturnStatement.mk($(xs,0)));
904
905     public rule throw_stmt =
906         seq(_throw, expr, SEMI)
907         .push(xs -> ThrowStatement.mk($(xs,0)));
908
909     public rule break_stmt =
910         seq(_break, iden.maybe(), SEMI)
911         .push(xs -> BreakStatement.mk($(xs,0)));
912
913     public rule continue_stmt =
914         seq(_continue, iden.maybe(), SEMI)
915         .push(xs -> ContinueStatement.mk($(xs,0)));
916
917     public rule assert_stmt =
918         seq(_assert, expr, seq(COL, expr).maybe(), SEMI)
919         .push(xs -> AssertStatement.mk($(xs,0), $(xs,1)));
920
921     public rule semi_stmt =
922         SEMI
923         .push(xs -> SemiStatement.mk());
924
925     public rule expr_stmt =
926         seq(expr, SEMI);
927
928     public rule labelled_stmt =
929         seq(iden, COL, _stmt)
930         .push(xs -> LabelledStatement.mk($(xs,0), $(xs,1)));
931
932     public rule stmt = choice(
933         _block,
934         if_stmt,
935         basic_for_stmt,
936         enhanced_for_stmt,
937         while_stmt,
938         do_while_stmt,
939         try_stmt,
940         switch_stmt,
941         synchronized_stmt,
942         return_stmt,
943         throw_stmt,
944         break_stmt,
945         continue_stmt,
946         assert_stmt,
947         semi_stmt,
948         expr_stmt,
949         labelled_stmt,
950         var_decl,
951         type_decl);
952
953     public rule block =
954         seq(LBRACE, stmt.at_least(0), RBRACE)
955         .push(xs -> Block.mk(list(xs)));
```

```
956
957     public rule statements =
958         stmt.at_least(0)
959         .collect().as_list(Statement.class);
960
961     /// TOP-LEVEL ===================================================================
962
963     public rule package_decl =
964         seq(annotations, _package, qualified_iden, SEMI)
965         .push(xs -> PackageDeclaration.mk($(xs,0), $(xs,1)));
966
967     public rule import_decl =
968         seq(_import, _static.as_bool(), qualified_iden, seq(DOT, STAR).as_bool(), SEMI)
969         .push(xs -> ImportDeclaration.mk($(xs,0), $(xs,1), $(xs,2)));
970
971     public rule import_decls =
972         import_decl.at_least(0)
973         .collect().as_list(ImportDeclaration.class);
974
975     public rule root =
976         seq(ws, package_decl.maybe(), import_decls, type_decls)
977         .push(xs -> JavaFile.mk($(xs,0), $(xs,1), $(xs,2)));
978
979     // ============================================================================
980
981     { make_rule_names(); }
982 }
```