Red Shift: Procedural Shift-Reduce Parsing (Vision Paper)

Nicolas Laurent
ICTEAM
Université catholique de Louvain, Belgium
nicolas.laurent@uclouvain.be

Abstract

Red Shift is a new design pattern for implementing parsers. The pattern draws ideas from traditional shift-reduce parsing as well as procedural PEG parsers. Red Shift parsers behave like shift-reduce parsers, but eliminate ambiguity by always prioritizing reductions over shifts. To compensate the resulting lack of expressivity, reducers are not simple reduction rules but full-blown procedures written in a general-purpose host language. I found many advantages to this style of parsing. In particular, we can generate high-quality error messages more easily; and compose different style of parsers. I also speculate about how Red Shift parsers may improve partial compilation in the context of an IDE.

CCS Concepts • **Software and its engineering → Parsers**;

Keywords parsing, parsers, pattern

ACM Reference Format:
Nicolas Laurent. 2017. Red Shift: Procedural Shift-Reduce Parsing (Vision Paper). In *Proceedings of 2017 ACM SIGPLAN International Conference on Software Language Engineering (SLE'17).* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3136014.3136036

1 Introduction

Parsing tools such as parser combinator libraries [3] and parser generators are well established among practicing programmers. Parsing combinator libraries are popular for simple parsing tasks that exceed the capabilities of regular expressions. Language workbenches [1] enable deriving whole development environments for simple domain specific languages, based on a set of specifications. Yet when it comes to writing compilers for general-purpose languages, programmers often eschew parsing tools in favour of ad-hoc parsers[1].

Yet ad-hoc parsers are far from perfect, and it certainly is less easy to write your own parser than it is to write a grammar for an existing tool. Why are programmers willing to put up with the additional pain? Based on my own experience and on many discussions with fellow programmers, I believe this is not due to lack of awareness about the tools, nor to the *not-invented-here* syndrome. Instead, there are two main factors that steer compiler authors away from parsing tools.

Error-reporting. Upon failure, most parsing tools report vague errors — indicating the best guess for the error location, but not much more.

Lack of control. Parsing tools are typically built around formalisms such as Context-Free Grammar (CFG) or Parsing Expression Grammar (PEG) [2], which leave little space for context-sensitive syntax such as significant whitespace, user-defined operators or length-delimited fields [5]. Lack of control also manifests as lack of flexibility in abstract parse tree construction.

Both problems have been recognized and tackled to some extent, albeit never in concert. Maidl et al. add labelled failure to the PEG formalism, as well as the ability to use these labels to influence the semantics of the choice operator [6]. These features allow retooling a PEG so that its parser generates errors more closely matching users' expectations. My own prior work extends the PEG formalism to enable context-sensitive parsing, while enforcing a strong safety contract. That paper also includes a discussion of the context-sensitive parsing literature [5]. Because Red Shift does not feature backtracking and makes the context available implicitly, it is exempt from the issues tackled in that paper.

This paper presents a new pattern for writing ad-hoc parsers. Using this pattern goes a long way to alleviate the pain incurred by the two factors identified above, while

Nicolas Laurent is a research fellow of the Belgian fund for scientific research (F.R.S.-FNRS).
For Manon, with love.

SLE'17, October 23–24, 2017, Vancouver, Canada
© 2017 Copyright held by the owner/author(s). Publication rights licensed to Association for Computing Machinery.
ACM ISBN 978-1-4503-5525-4/17/10...$15.00
https://doi.org/10.1145/3136014.3136036

[1]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.

avoiding the traditional pitfalls of traditional ad-hoc parsers (which we describe in the next sub-section).

1.1 Ad-Hoc Recursive Descent Parsers

Ad-hoc parsers are overwhelmingly written in the *top-down recursive descent* style. These parsers have well-known pitfalls: left-recursion handling is awkward, and handling operator precedence is thorny [7]. Ad-hoc recursive descent parsers enable writing better error reports, and implementing context-sensitive features; but they do not make it easy. We now review a few notable implications of using ad-hoc recursive descent parsers.

Error-reporting. Using ad-hoc parsing, programmers can write custom error-reporting code; but generating good parse errors requires keeping track of the context in which the error occurs. By default, recursive descent parsers encode this context inside call stack frames, which means that it is not immediately available at the error detection site.

Taking a step back, it is already non-trivial to determine the nature of the error, when faced with an invalid input. This notably occurs with disjunction constructs, for instance when we attempt to match a method declaration, *or* a field declaration, *or* a constructor declaration. If none of the alternatives succeed, what to report? Can we infer what alternative was meant by the user? This problem is endemic to parsing tools, and using an ad-hoc parser does not make it vanish magically. However, in some cases the programmer may now disambiguate the error's origin, using his knowledge of the language's grammar.

It is also not obvious what steps to take after an error has been detected. When an error occurs in a syntactic unit (e.g. an expression), this leaves all outer syntactic units (e.g. a statement, a function declaration, a class declaration) potentially hanging. There are two options for dealing with these situations.

The naive approach is to abort the parse as soon as an error is encountered. The alternative is to use recovery rules to skip the erroneous part of the input and *resynchronize* the parse, in order not to hinder the matching of the outer syntactic units. To do so, two main strategies exist. The first is to associate the recovery rules to the error detection code — in order to resynchronize the parse immediately after an error is encountered. The second strategy is to associate the recovery rules with syntactic units. For instance, when a compound statement delimited by curly braces is faced with an error in one of its components, it may skip ahead to the closing curly brace. These strategies work, but are hard to deploy sytematically — it takes a lot of finesse to determine how to implement the appropriate strategy. They also hurt transparency: explaining the observed behaviour of the parser now requires explaining all recovery rules.

Lack of control. With ad-hoc parsers, programmers are able to parse context-sensitive syntax, but doing so requires the use of a global context. Programmers must take great care to preserve the context's integrity in the presence of backtracking [5]; and recursive descent parsing makes heavy use of backtracking.

Performance. Ad-hoc parsing mostly ensures good performance, but some traps lurk for the unwary. For instance, it is easy to end up with an operator-handling algorithm that is exponential, if one tries to emulate how operators are handled in CFG[2], and disregards existing algorithms [7].

Reuse and composition. While ad-hoc recursive descent parsing has benefits, it also foregoes many benefits of traditional parsing tools. One of the appeals of parsing combinator libraries is the ability to write and combine reusable parsing components. Some libraries go further and enable the definition of custom combinators [3, 5]. Ad-hoc parsers, as the name implies, have no such capabilities. Because each parser has its own architecture and data structures, even manually porting components may prove difficult.

Red Shift parsers do not solve all the problems of ad-hoc parsers. For instance, they do not provide facilities to integrate with formal grammars. By definition, ad-hoc parsers aren't automatically derived from a formal grammar, even though such a grammar may exist and serve as reference. An ad-hoc parser is not an adequate description of the language syntax for an end-user; and establishing the adequation of the parser to the grammar is non-trivial.

1.2 Inspirations

The pattern we propose in the next section draws from three main sources of inspirations.

Shift-Reduce Parsing. Shift-reduce parsers consume an input stream (usually made of tokens) linearly, without backtracking. As the parse proceeds, the parser builds up a parse tree. A stack data structure is used as temporary storage for parse tree nodes. At each step of the parse, the parser must choose between a *shift* action: consuming an item from the stream and pushing it on the stack; or a *reduce* action: combining multiple items at the top of the stack. The parser makes this decision based on the content of the stack and a few tokens of lookahead.

Parsing Expression Grammars. PEGs are a formalization of top-down recursive descent parsers [2], making it somewhat of a trade-off betweeen ad-hoc recursive descent parsers and the CFG formalism. As a formalism, PEG has a few interesting properties. First, it is unambiguous: a PEG only ever admits a single derivation tree for each input. PEGs are also recognition-based, describing in essence a predicate that can be applied to any input to test its membership in the described language. CFGs, on the other hand, describe how to generate every input that belongs to the language.

[2]The author has been bitten by this in the past.

```
inputs
    reducers: List[Reducer]
    token: Queue[Token]
context = new Context()
stack = new Stack()
while !tokens.is_empty() do
    stack.push(tokens.dequeue())
    outer: while true do
        for reducer in reducers do
            if reducer.reduce(context, stack, tokens) then
                continue outer
        break
```

Algorithm 1: Core loop of a Red Shift parser.

Said otherwise, PEG is more procedural, while CFG is more declarative. The obvious implementaion of PEG's semantics is very close to recursive descent parsers, making it easy to extend with custom code [5].

A parser is not a recognizer. It is uncontroversial to say that the role of a parser is to superimpose structure over a linear input. However, the idea that a parser should only accept syntactically valid inputs is also prevalent. This idea was probably inherited from formal language theory. After all, the role of a grammar is to describe the sentences (inputs) that belong to a language. As parsers are often specified with grammars, it is not surprising that the idea seeped through.

However, I argue that the idea that a parser should reject syntactically invalid inputs is actually harmful to the first mission of parsers: imposing structure. If a perfectly recognizable syntactic unit (say a function definition) appears in a context where it is not allowed (say in another function definition, as in C or Java), it is perfectly reasonable to parse it anyway: the unit can be clearly recognized.

This emphasis on structure comes with two benefits. First, it may help writing simpler parsers, as most syntactic units may now be recognized globally, not only in specific contexts. Second, it leads to much better error messages: after generating a syntax tree, we can inspect it for conformity and use the context supplied by the rest of the tree in our report. The difference I have in mind here is the difference between the error messages "Unexpected token: abstract" and "The abstract modifier is not allowed for field declarations".

2 Description of the Pattern

This section describes the Red Shift design pattern. It is not an algorithm or a library, but rather a way to organize parsing code in a way that yields numerous benefits. After a succint description of the core idea, I will present a number of ways in which the pattern can be exploited.

Core loop. Algorithm 1 describes the core of a Red Shift parser in terms of a OO-style pseudo-code. Given a stream of tokens, we shift the first one onto the stack, then attempt to run each of our reducers in turn. A reducer will return `true` only if it successfully performed a reduction (entailing a modification of the stack). If a reducer succeeds, we attempt another reduction, re-starting from the top of the reducers list. When no further reductions can be applied, we shift the next token onto the stack, and repeat the same process. This continues until the token stream is exhausted.

There is no centralized reducer dispatch: each reducer is queried in a pre-defined order and given the opportunity to perform a reduction. To take this decision, reducers have access to the stack, the token stream, and a global context.

Reducer Aggregation. Attempting every reducer for every token is a clear-cut case of performance regression compared to traditional shift-reduce parsers. This can be mitigated by aggregating multiple reducers together: you can create a reducer that looks up the top of the stack and the next token, then calls the appropriate reducer based on a table lookup.

Note that this does not fundamentally change the nature of Red Shift: aggregation does not introduce backtracking. Replacing a sequence of reducers that match on the first token by an aggregated table-lookup reducer is simply a convenient semantics-preserving optimization.

Specialize using the global context. By default, every reducer can be applied at any position in the token stream. This is consistent with the idea that parsers should superimpose structure rather than check for correctness. This still leaves some irreducible ambiguity: the same syntax might mean different things depending on the context. To disambiguate, I suggest using a global context object, passed to every reducer. Reducers can collaborate through this object by indicating changes in the context (e.g. a special construct is entered / exited) and modifying their behaviour depending on the context.

Error detection a posteriori. The pattern does not make any special provision for error-reporting. Instead, error detection and reporting should be done after the parse completes (i.e. after all the tokens have been shifted). If the input was syntactically correct, the result of the parse should be a single object on the stack. Otherwise, the stack contains multiple objects (usually tree nodes and tokens). Each of these objects potentially indicate a missed opportunity for reduction.

To detect errors, I propose running a series of *error-reporters*, once for each item in the stack. These error-reporters are very similar to reducers: by looking at the stack around the given stack element, they try to determine a syntactic error that could have prevented reduction. In some case these errors are clear-cut, such as a prefix operator appearing in betweeen two otherwise well-formed expressions, or an illegal item appearing in a list (say a statement in a list of expressions). Other situations are ambiguous, and the reporter may use an heuristic to guess at the source of the issue.

Parser extension and composition. A Red Shift parser can be extended by adding new reducers. Appending to the reducer list is the safest option to introduce new syntax, although we should note it does not guarantee that existing valid inputs will continue to parse equivalently. Existing parsers can also be overriden by inserting new parsers ahead of them. As outlined earlier, reducer aggregation is also an interesting strategy, especially when disambiguation is required. Finally, a reducer may take advantage of its ability to manipulate the stack to second-guess the output of an earlier reducer.

It is theoretically possible to compose Red Shift parsers, but the resulting semantics is distinctively not obvious. Language embedding, on the other hand, is easier and far more likely to be useful. It can be accomplish by aggregating the embedded parser's reducers, and enabling them only in certain contexts.

Incremental construction. While writing reducers, you are likely to push *incomplete* objects onto the stack. These objects represent the information contained in part of the syntax, and are to be completed by a subsequent reduction. Although reducers are able to shift tokens out of the stream, using incomplete objects is often an easier way to structure the code. In the case where the object is never completed, it becomes straightforward to write an error-reporter to highlight the error. In the next section, I will describe how I used incomplete objects to implement operator parsing.

Generic components and combinators. While Red Shift parsers are ad-hoc, it is possible to write components that can be shared among parsers, assuming the reducer signature is identical. For most reducers, the difficulty is that they operate on parser-specific stack objects. This can be accomodated by writing two "adapter" reducers: one to translate the input of the reused reducer, one to translate its output. Writing generic reducer combinators (reducers that dispatch to other reducers, as in reducer aggregation) is even easier.

Combine parsing styles. Since a reducer has access to the stack and the token stream (and has the ability to perform shift actions), it can use them to parse a portion of the input any way it wants. This is most useful to embed small lookahead-based parsers into the Red Shift parser. One could also embed a recursive descent parser, or even another Red Shift parser that would be cleanly isolated from all outer reducers. The pattern gives us the flexibility to pick the style that works best.

3 Discussion

3.1 Prototype

I had the opportunity to test out some of the ideas outlined in this paper in a small compiler prototype. The compiler accepts a simple Java-like language.

One of the most pleasant uses of the pattern was dealing with operators. Grammars usually do not have explicit support for precedence or associativity, and encoding operator precedence can be a tedious affair, especially for PEGs, where it is easy to accidentally induce exponential execution times [4]. Here, I was free to define a table of operators which specified their type (prefix, suffix, binary), precedence and associativity, then to write a single reducer that took care of building up the expression tree. The operator table can be extended with user-defined operators, which will then be transparently handled by the reducer.

The operator reducer is triggered when an operator token has been shifted on the stack. It generates an operator parse tree node and, if applicable, merges it with the preceding node, carefully inserting it at the appropriate location given its precedence and associativity. As a result, operator trees are always correct with respect to precedence and associativity. However, it is possible that some operators' operands were omitted. It was trivial to build an error-reporter that verified that no operand was omitted. When we detect an omitted operand, the real error the user made might not have been to omit an operand, but to be mistaken about the precedence of some operator. At this point, the error-reporter is able to show a textual representation of the expression tree in order to clarify the situation. Another error-reporter was responsible to look for consecutive expressions, which might indicate a missing operator.

Otherwise, the parser's implementation is fairly straightforward. Some reducers are responsible to aggregate sequenced items (lists of expression parameters, statements, declarations) and associated error-reporters look for items that prevent these reductions from happening. Some reducers use lookahead for disambiguation purpose, for instance to disambiguate function calls from variable references and field access.

3.2 Prospects

Beyond the benefits that have already been outlined, I believe one major prospect of this technique is in enabling partial analysis of the source code by a compiler's semantic analysis step. This is particularly useful in an IDE setting, where we would like to derive as much information on the source as possible (most notably types and name resolutions), even when it contains syntax errors.

From this perspective, the big benefit of the pattern is that it produces a representation of the source that is *most reduced* given syntactic errors. Additional work might be necessary to exploit this representation. For instance, the presence of an odd item in a list of statements might prevent us to see that these statements form the body of a function. But if we are able to report these errors, it stands to reason that we can also build *error-removers* that eliminate (but not correct) the errors; so that the information passed to the semantic analysis step may be more accurate.

Acknowledgments

Nicolas Laurent is a research fellow of the Belgian fund for scientific research (F.R.S.-FNRS).

References

[1] Sebastian Erdweg, Tijs van der Storm, Markus Völter, Meinte Boersma, Remi Bosman, William R. Cook, Albert Gerritsen, Angelo Hulshout, Steven Kelly, Alex Loh, Gabriël D. P. Konat, Pedro J. Molina, Martin Palatnik, Risto Pohjonen, Eugen Schindler, Klemens Schindler, Riccardo Solmi, Vlad A. Vergu, Eelco Visser, Kevin van der Vlist, Guido H. Wachsmuth, and Jimi van der Woning. 2013. *The State of the Art in Language Workbenches.* Springer International Publishing, Cham, 197–217. DOI: http://dx.doi.org/10.1007/978-3-319-02654-1_11

[2] Bryan Ford. 2004. Parsing Expression Grammars: A Recognition-based Syntactic Foundation. *SIGPLAN Notices* 39, 1 (Jan. 2004), 111–122. DOI: http://dx.doi.org/10.1145/982962.964011

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

[4] Nicolas Laurent and Kim Mens. 2015. Parsing Expression Grammars Made Practical. In *Proceedings of the ACM SIGPLAN International Conference on Software Language Engineering (SLE 2015).* ACM, 167–172. DOI: http://dx.doi.org/10.1145/2814251.2814265

[5] Nicolas Laurent and Kim Mens. 2016. Taming context-sensitive languages with principled stateful parsing. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Software Language Engineering, Amsterdam, The Netherlands, October 31 - November 1, 2016.* 15–27. http://dl.acm.org/citation.cfm?id=2997370

[6] André Murbach Maidl, Fabio Mascarenhas, Sérgio Medeiros, and Roberto Ierusalimschy. 2016. Error reporting in Parsing Expression Grammars. *Science of Computer Programming* 132 (2016), 129 – 140. DOI: http://dx.doi.org/10.1016/j.scico.2016.08.004 Selected and extended papers from SBLP 2013.

[7] Theodore Norvell. 2001. Parsing expressions by recursive descent. (2001). https://www.engr.mun.ca/~theo/Misc/exp_parsing.htm