# Simple, efficient, sound-and-complete combinator parsing for all context-free grammars, using an oracle

OCaml 2014 workshop, talk proposal

Tom Ridge
University of Leicester, UK
tr61@le.ac.uk

*This proposal describes a parsing library that is based on current work due to be published as [3]. The talk will focus on the OCaml library and its usage.*

## 1. Combinator parsers

Parsers for context-free grammars can be implemented directly and naturally in a functional style known as "combinator parsing", using recursion following the structure of the grammar rules. Traditionally parser combinators have struggled to handle all features of context-free grammars, such as left recursion.

## 2. Sound and complete combinator parsers

Previous work [2] introduced novel parser combinators that could be used to parse all context-free grammars. A parser generator built using these combinators was proved both *sound* and *complete* in the HOL4 theorem prover.

This previous approach has all the traditional benefits of combinator parsing libraries. The additional advantages are:

- the ability to handle all context free grammars
- guaranteed termination for any constructed parser, whilst preserving completeness of parsed results
- good theoretical behaviour

The combinator parsing interface allows parsers to be combined easily, and parsers to be parameterized by other parsers. Guaranteed termination means that, however the parsers are constructed, parsing is guaranteed to terminate. "Good theoretical behaviour" means, for example, that equational reasoning about parsers is supported, and not subject to side conditions about the termination of the parsers involved (since all parsers terminate). Together, these features support modular definition and combination of parsers.

Unfortunately the performance of this approach is not as good as other parsing methods such as Earley parsing.

## 3. $O(n^3)$ parsing, using a combinator library

In this work, we build on this previous work, and combine it in novel ways with existing parsing techniques such as Earley parsing. The result is a sound-and-complete combinator parsing library that *can handle all context-free grammars*, and *has good performance*.

In outline, our approach consists of the following. We first define parser combinators that allow the user to define parsers much as with traditional combinator parsing (one difference is that, since we can handle all grammars, grammar features such as left-recursion are not a problem). From this we can extract a concrete representation of the grammar (as a list of rules). We give the grammar, the start symbol and the input to the back-end Earley parser, which parses the input and returns a parsing oracle. The oracle is then used to guide the action phase of the parser.

## 4. Real-world performance

In addition to $O(n^3)$ theoretical performance, we also optimize our back-end Earley parser. Extensive performance measurements on small grammars (described in [3]) indicate that the performance is arguably better than the Happy parser generator [1].

## 5. Online distribution

This work has led to the P3 parsing library for OCaml, freely available online[1]. The distribution includes extensive examples. Some of the examples are:

- Parsing and disambiguation of arithmetic expressions using precedence annotations. This shows that traditional features of parsers (precedence annotations) do not need to be integrated into the parsing implementation, but can be supported directly and naturally by appropriate actions on top of the basic P3 parser combinators (which have no notion of precedence or associativity built in).
- Parsing and disambiguation of arithmetic expressions using rewrite rules. One way to disambiguate arithmetic expressions is to use rewrite rules. For example, a rewrite rule $x+[y+z] \rightsquigarrow [x+y]+z$ might indicate that the $+$ operator should associate to the left. This approach relies on rewriting parse trees. Typically arithmetic expressions (without disambiguation) result in exponentially many parse trees, and thus this approach to disambiguation has not so far been feasible. P3 allows such an approach in polynomial time. This is possible because P3 uses an oracle to represent parse results, rather than actual parse trees. In addition, P3 can use memoization to avoid repeated computation.
- Modular definition of parsers. One of the examples is a modular development of parsers and evaluators for 3 languages (boolean expressions, arithmetic expressions and lambda calculus). These parsers and evaluators are then combined to give a parser and evaluator for the combined language.
- A parser for OCaml, based on the `ocamlyacc` grammar from the OCaml distribution.
- A CSV parser. The definition of the parser is parameterized by parsers for field delimiters (typically the comma), the field quotation character (typically a double quote). Even the record

[1] `https://github.com/tomjridge/p3`

terminator (typically a newline) is parameterized. This results in a very compact CSV parser (about 20 lines of code) that handles all variations of the CSV format.

## 6. Potential applications

Our library can parse general context-free grammars. As such, the performance is not competitive with specialized parsing techniques, such as LALR (the basis for the standard `ocamlyacc` OCaml parser generator). Traditional combinator parsers are also not competitive with these specialized techniques, however, their advantages mean that they have found widespread use. They are often used, for example, to prototype parsers. We believe that the primary use of our technique and library will be in the areas where combinator parsing has traditionally been strong, such as prototyping. In addition, the unique features enable several new applications.

- The ability to handle all grammars means that the parsing interface is extremely flexible and powerful. The interface is also very simple: users can forget about typical errors that are encountered with less general approaches (shift/reduce conflicts etc.), or the possibility of non-termination (as with traditional combinator parsers). The library can be used interactively, which facilitates parser development and debugging.
- The ability to parameterize parsers by other parsers allows parsers to be extended easily: A modular development of a parser and evaluator for lambda calculus expressions extended with booleans and arithmetic expressions is available in the online distribution.

  The parsers for boolean expressions, arithmetic expressions and lambda expressions are defined separately, but parameterized over another unknown parser. The actions for each parser are defined with each parser, using polymorphic variants. Each parser is usable on its own. A final step "ties the knot" and combines these three parsers to give a parser and evaluator for the combined language. The modular development and reuse of parsers and evaluator is important for the development of domain specific languages.
- The parsing library outperforms the popular Haskell Happy parser generator across all our example grammars, often dramatically so. The performance rests on a careful analysis and engineering of the back-end parser, based on Earley's algorithm. In practice, even for relatively large inputs, P3 parsers deliver reasonable performance. This means that the library may be usable not only for prototyping, but as the basis for real-world applications.

The talk will describe our approach and the main features of P3, detail the OCaml API for the library, and finally demonstrate some examples of P3 parsers.

## References


[1] Happy, a parser generator for Haskell. `http://www.haskell.org/happy/`.

[2] T. Ridge. Simple, functional, sound and complete parsing for all context-free grammars. In *CPP*, pages 103–118. Springer, 2011.

[3] T. Ridge. Simple, efficient, sound and complete combinator parsing for all context-free grammars, using an oracle. In *SLE*, 2014.