Iterator-Based Parser Combinators for Ambiguous Grammars

Param Desai, Devaansh Mann, Kyle Dewey
Department of Computer Science, California State University, Northridge
{ param-ridham.desai.167, devaansh.mann.836 }@my.csun.edu, kyle.dewey@csun.edu

Abstract

For most programming languages, context-free grammars form the basis for syntactic validity. Most of the literature is focused on how to parse unambiguous context-free grammars, but we observe that, in practice, real languages (including Swift) are sometimes ambiguous. This paper proposes a new technique for parsing ambiguous grammars: iterator-based parser combinators. Like traditional parser combinators, iterator-based parser combinators allow for users to write their own parsing operations, and they are handled at the library level. In fact, iterator-based parser combinators serve as a drop-in replacement for traditional parser combinators. However, iterator-based parser combinators can compute all parses, and they do so on demand, meaning that the only work that is performed is specific to the parses actually used. We implement an iterator-based parser combinator library in Java, and apply it to parsing several globally and locally ambiguous grammars. Our results show that the runtime of them is low, particularly for the first valid parse, demonstrating their promise for handling ambiguous grammars.

1. Introduction

Grammars are formal sets of rules which describe the structure of valid sentences in a language. In programming languages, grammars are used to determine if a given program is syntactically valid or not. Grammars can be *unambiguous*, meaning there is only one possible way to apply the rules to show that a sentence is synatically well-formed. Alternatively, they can be *ambiguous*, meaning there are multiple ways to apply the rules, each yielding a different successful parse of the input. For example, consider the following grammar, which is intended to represent sums of numbers:

```
exp ::= NUM | exp '+' exp
```

This grammar defines the `exp` (expression) rule, stating that an expression can either be a number (`NUM`), or two expressions separated by `+`. The `exp` rule is recursively defined, allowing expressions to be as deeply nested as necessary. Some example sentences this grammar accepts are `5`, `6 + 7`, and `1 + 2 + 3`. With `1 + 2 + 3`, this can be read either as `(1 + 2) + 3` (meaning the leftmost expression is `1 + 2`), or as `1 + (2 + 3)` (meaning the rightmost expression is `2 + 3`). This shows that this grammar is ambiguous.

In programming languages, ambiguous grammars are generally not ideal, as different possible parses can be different possible ways to execute the program. For example, this issue was seen early in computing with the *dangling else problem* [1], wherein the `else` portion of a nested `if` statement could be potentially matched with different `if`s depending on how it was parsed. However, it can be difficult to completely avoid ambiguity in a grammar's definition, even in programming languages. Furthermore, a grammar may merely be *locally ambiguous* [2], meaning there is only one valid parse overall, but subportions of a sentence may have multiple parses. For example, the grammar for Apple's Swift language [3] is ambiguous [4], though in practice only locally ambiguous.

We observe that there are already a variety of techniques and tools for writing parsers for unambiguous grammars, including recursive descent parsers [5], ANTLR [6], and parser combinators [7–10]. However, support for ambiguous grammars is comparatively limited, despite the fact that real programming languages make use of them. Additionally, some techniques force all parses to be computed and stored before any of them can be used, which is impractical on very ambiguous grammars.

To address these issues, we propose a new way to efficiently parse ambiguous grammars. Our approach is parser combinator variant which serves as a drop-in replacement for traditional parser combinators. Via careful use of iterators, we incrementally compute each parse on demand, unlike techniques which perform all parses at once. We implement our technique in Java as a library, and demonstrate its ability to parse unambiguous grammars. Overall, our contributions are as follows:

1. A new technique for parsing ambiguous grammars, based on parser combinators (Section 3)
2. An evaluation of this technique's runtime over several ambiguous grammars (Section 4)

2. Background and Related Work

This section focuses on parser combinators, as our approach is based on them. We use the following grammar in this section for illustrative purposes, where lowercase letters are tokens in the input:

```
G ::= a | bGc
```

Individual programs are parsed to *abstract syntax trees* (ASTs), which are a data structure representation of the input program. The specific node types in an AST correspond closely to the grammar. With respect to grammar `G` above, there likely would be two AST nodes:

1. A leaf node representing `a`, hereafter referred to as `ANode`.
2. An internal node representing `bGc`. This is an internal node recursively using `G`, which itself is represented by an AST node. We refer to this as `BNode(r)`, where `r` is the child node.

For example, the input `bbacc` would parse to the AST node `BNode(BNode(ANode))`.

2.1. Traditional Parser Combinators

Parser combinators (PCs) are a popular technique for writing parsers in functional languages [7, 8]. PCs behave much like recursive descent parsers, but abstract over common details. The basic primitive value with PCs is a *parser*, which is a single executable unit. A Java representation of a parser and related code is shown in Figure 1. As shown, parsers have only a single method: `parse`. `parse` takes a position `p`, indicating where in the input list of tokens it should start parsing from. From there, `parse` returns `Res`, where `Res` includes the specific value parsed in (`result`), and the next position to start parsing from (`next`). The value parsed in is intentionally encoded with a type variable (`A`), permitting different parsers to parse in values of different types. `Res` is wrapped in `Optional`, so upon calling `parse`, one might receive either a `Res` as expected, or an empty value. The empty value is used to indicate parse failure, i.e., `parse` could not parse its input. Conversely, on success, a single `Res` value is returned.

In contrast to recursive descent parsing, parsers exist as explicit types with PCs, making it possible to write operations that work directly on parsers. This is commonly exploited to construct larger parsers from smaller ones. The signatures of some common related operations and related utilities are shown in Figure 2.

Figure 2's `token` constructs a parser that reads in the given token `c`, which for expository reasons is respresented as a `char`. On success, the `Character` representation of `c` is returned. In practice, `token` will read in whatever the underlying type of the input tokens are instead of `char`.

```
public class Res<A> {
  public final A result;
  public final int next;
  public Res(A a, int n) {
    result = a; next = n;
  }
}
public interface Parser<A> {
  public Optional<Res<A>> parse(int p);
}
```

Figure 1. Definition of a parser and related code for traditional PCs.

```
@FunctionalInterface
public interface Thunk<A> {
  public A execute();
}
@FunctionalInterface
public interface Function<A, B> {
  public B execute(A a);
}
public class Pair<A, B> {
  public final A first;
  public final B second;
  public Pair(final A a, final B b) {
    first = a; second = b;
  }
}
public static Parser<Character>
  token(char c);
public static <A, B> Parser<Pair<A, B>>
  and(Parser<A> a, Thunk<Parser<B>> b);
public static <A> Parser<A>
  or(Parser<A> p1, Thunk<Parser<A>> p2);
public static <A> Parser<List<A>>
  star(Parser<A> p);
public static <A, B> Parser<B>
  map(Parser<A> p, Function<A, B> f);
```

Figure 2. Common PC-related types and operations.

Figure 2's `and` runs two parsers in sequence, where the second parser picks up where the first parser left off. The values read in by `and`'s input parsers (`A` and `B`, respectively) are grouped into a single `Pair<A, B>` if both succeed. The purpose of `Thunk` in `and` is because we only need to construct the second parser (`b`) in the event that the first parser (`a`) succeeded; if the first parser fails, then the second parser will never be executed.

Figure 2's `or`, like `and`, also executes the first parser (`p1`) first. However, unlike `and`, if the first parser fails, then `or` instead runs the second parser (`p2`). Phrased another way, `or` will run the first parser that succeeds,

whereas `and` runs all parsers in sequence.

Figure 2's `star` will repeatedly apply a given parser (`p`) to the input, building up a list of values that were parsed in. Once `p` fails, the list is returned. If `p` initially fails, then the returned list will be empty.

Figure 2's `map` converts a parser of one type to another type, using the provided function `f`. Specifically, if `map`'s input parser (`p`) succeeds, then `f` is applied to `p`'s return value. Otherwise, if `p` fails, then the parser returned by `map` also fails.

Putting all these aforementioned helpers together, we can define a parser for our example grammar `G`, shown in Figure 3. The `a()` helper reads in an `a` token. Upon success, `a()` uses `map` to create `a`'s AST representation (`ANode`). The `bGc()` helper handles the `bGc` part of the grammar; it first reads in a `b`, followed by a recursive call to `G` (chained with `and`), followed by reading in a `c`. If all of those subcomponents succeed, then `map` is used to convert what was parsed in into a `BNode`. The function passed to `map` extracts out the node from the recursive call, and puts it into the new `BNode`. Finally, `a()` and `bGc()` are combined together in the `G()` method via `or`. Here we can see the value in `Thunk`; `bGc()` calls `G()`, which itself calls `bGc()` recursively. Without `Thunk`, this would create infinite recursion upon calling `G`. However, with `Thunk`, the recursive call is only made when absolutely necessary to continue parsing the input. Since the input is of finite length, this breaks any infinitely recursive chains, assuming the grammar itself is not left-recursive (i.e., no rules recursively call themselves without first reading at least one token, thus forcing recursive progress).

These helpers all neatly correspond to what is representable with context-free grammars, making it relatively easy to mechanically apply these helpers to match what the grammar says. While we implement these helpers ourselves, in practice, multiple libraries provide them (e.g., `scala-parser-combinators` for Scala [10] and Parsec for Haskell [9], among many others).

2.2. List-based PCs

One issue with PCs as described is that they are inappropriate for parsing ambigiuous grammars. With ambiguous grammars, at best they will only generate a single parse. At worst they will fail on any valid inputs which require backtracking. This issue is rooted in `parse`'s return type (Figure 1): `Optional<Res<A>>`. `Optional` either wraps around one value, or is empty, meaning `Optional` can only encode 0 - 1 possible parses. Even if multiple parses are possible, `Optional` forces us to pick one, making traditional PCs unsuitable for ambiguous grammars.

Fortunately we can swap out `Optional` for a different type which can encode more than one possible parse. Wadler [11] and Koopman [12] do exactly this, and replace `Optional` with `List`. Lists can still encode parse failure (i.e., an empty list), but they can also encode any number of successes (i.e., a list of length n where $n > 0$). The helpers in Figure 2 admittedly need some tweaking to work with a `List`-based representation, but the changes are surprisingly minimal. With `token`, instead of returning a single `Character` wrapped in an `Optional` on success, we instead return a list holding a single value, namely the `Character` representation of the `char` we parsed in. On parse failure, `token` returns an empty list.

With `or`, instead of optionally executing a second parser, we instead *always* execute both parsers, producing two separate lists of successes. From there, the two lists are appended together into a single result list. In this case, failure means both parsers returned empty lists. Appending two empty lists together leads to an empty result list, naturally representing failure of both parsers.

`and` is more complex. Specifically, for every parse success of the first parser, we must run the second parser. Then, for every result of the second parser, we group it into a `Pair` with the corresponding result of the first parser. This is shown below in Python-like psuedocode, where `a` and `b` are functions representing the first and second parser, respectively. `Res` and `Pair` correspond to their definitions from the prior section, and `startPos` is the initial position we should start from for the `and`:

```
result = []
for resA in a(startPos):
  for resB in b(resA.next):
    p = Pair(resA.result, resB.result)
    r = Res(p, resB.next)
    result.append(r)
return result
```

There is, however, a major downside to switching to this `List`-based approach: before *any* parse can be accessed, the whole list must be constructed, so *all* parses must be complete before any of them can be used. This makes lists an impractical representation whenever many possible parses are expected.

3. Iterator-based PCs

Here is where we introduce our major innovation over PCs for ambiguous grammars, which builds directly off of the list-based PCs described in Section 2.2. We observe that while `List` is one potential encoding of 0 - n values, other possible encodings exist. In our case, *iterators* are instead a much more suitable representation. One possible basic iterator definition in Java is shown below:

```
public interface Iterator<A> {
  public Optional<A> next();
}
```

```
public static Parser<Node> a() {
  return map(token('a'), (Character c) -> new ANode());
}
public static Parser<Node> bGc() {
  return map(and(token('b'),
                 () -> and(G(),
                           () -> token('c'))),
             (Pair<Character, Pair<Node, Character>> pair) ->
               new BNode(pair.second.first));
}
public static Parser<Node> G() {
  return or(a(), () -> bGc());
}
```

Figure 3. Parser for grammar `G` using PCs.

With iterators, one would call the `next` method in order to retrieve an element. Only a single element is returned for each call to `next`. Once all elements have been iterated over, `next` returns an empty result, hence `Optional` appearing in the code above.

Importantly, iterators separate the creation of the iterator from the call to `next`. For example, one can define an iterator that iterates over all possible integers (mathematically speaking), or some other infinitely large space. While actually iterating over that space takes an infinite amount of time, this does not automatically mean that constructing the iterator will take an infinite amount of time. Specific to parsing, this means if we define `parse` like so:

```
public Iterator<Res<A>> parse(int p);
```

...then we do not need to construct all possible parses in `parse`. We can instead delay parse construction until `next` is called. Better yet, each call to `next` only needs to do the work necessary for whatever the next parse is.

Overall, with iterators, this means we can construct parses on demand. Depending on the application in play, if only a subset of parses are needed, then only the work necessary for that subset needs to be performed. In the domain of compilers, generally only one parse is used even if the grammar is ambiguous, so an iterator-based representation will only ever need to perform the work necessary for this one parse.

Swapping out list-based PCs with iterator-based versions is surprisingly straightforward. From Dewey et al. [13], both lists and iterators are *additive monads*, meaning they share a common set of four key operations obeying certain properties. The details of additive monads are beyond the scope of this paper, but we found that PCs can be implemented using only these four operations, greatly simplifying the switch to iterators. Most importantly, just as moving from `Optional`-based to list-based PCs did not change the user-facing interface, switching from list-based to iterator-based PCs does not change anything in the user-facing interface. In other words, the exact same code in Figure 3 works for all three PC implementations, and the different implementations only change internal implementation details within the helper functions (e.g., `and` and `or`). Putting all this information together, we implemented an iterator-based PC library in Java.

4. Evaluation

In this section, we evaluate iterator-based PCs on a mix of globally and locally ambiguous grammars. These grammars are shown in the first column of Table 1. For each grammar, we generate a list of guaranteed parsable tokens via *stochastic grammars* [14]. With stochastic grammars, we perform a random recursive walk over the grammar, generating tokens according to the path taken. The parsing time can vary widely between grammars based on the number of tokens in the input, so the number of tokens generated for each grammar was adjusted to try to keep overall parsing time under 400 ms. After using stochastic grammars to make a parsable token string for each grammar, we parsed the same string of tokens 1,000 times using parsers written with our approach. We separately record the average time taken for the first parse, as well as all parses. Table 1 details our results.

For grammar `S`, it takes approximately 8,812X less time to compute the first parse compared to all parses. This is because `S` is highly ambiguous, and the number of possible parses grows exponentially with the number of tokens. For example, while `a` and `aa` only have one possible parse apiece, a string of 25 `a`'s has over 75,000 possible parses. A list-based PC must compute all of these, even if only a single one is desired. In contrast, our iterator-based PCs only need to do an amount of work proportional to the number of parses needed. For the remaining grammars, while all are at least somewhat locally ambiguous due to all production rules sharing a prefix, none are globally ambiguous, so only one parse is possible. However,

Grammar	Locally Ambiguous	Globally Ambiguous	Number of Tokens	First Parse (ms)	Total Runtime (ms)
`S ::= a | aS | aaS`	Yes	Yes	25	0.03	264.36
`D ::= ab | abD | aDb`	Yes	No	100,000	19.61	39.75
`E ::= ab | abE | aEb | aEa`	Yes	No	50	70.16	341.68
`F ::= a | abF`	Yes	No	2,000	26.47	26.58

Table 1. Grammars used in our evaluation. **First Parse (ms)** is the average amount of time taken to get the first parse over 1,000 iterations, and **Runtime (ms)** is the average time taken to produce all parses for the same number of iterations.

we may still end up searching for a second parse which does not exist, hence the total runtime is still often significantly greater than the first parse. While we have not compared our PC implementation to other ambiguous grammar parsing techniques, the relatively short runtimes in Table 1, particularly for the first parse, shows that our iterator-based approach is nonetheless a viable parsing strategy.

5. Conclusion

Most parsing tools and techniques assume that a given language's grammar is unambiguous, but real programming languages like Swift can nonetheless have ambiguous grammars. As such, we still need a way to handle ambiguous grammars. To that end, in this paper we have presented iterator-based PCs, a new approach to parsing ambiguous grammars. Iterator-based PCs offer all the advantages of traditional PCs, and serve as a drop-in replacement. However, unlike traditional PCs, iterator-based PCs can produce all possible parses. Moreover, these parses are constructed as needed, saving significant resources in the event that not all parses need to be generated. We have demonstrated that these work over several unambiguous and ambiguous grammars, and that the overall runtime seems low. For future work, we hope to evaluate these against other parsing techniques for handling both ambiguous and unambiguous grammars, to get a better sense of how they perform. We also wish to use these to handle larger grammars, including those of popular programming languages.

References

[1] Abrahams PW. A final solution to the dangling else of algol 60 and related languages. Commun ACM September 1966; 9(9):679–682. ISSN 0001-0782. URL https://doi.org/10.1145/365813.365821.

[2] Syntactic ambiguity; locally ambiguous. URL https://en.wikipedia.org/wiki/Syntactic_ambiguity#Locally_ambiguous.

[3] Apple. Summary of the grammar, 2025. URL https://docs.swift.org/swift-book/documentation/the-swift-programming-language/summaryofthegrammar/.

[4] Canto Hyatt S, Dewey K. Mutation-based Fuzzing of the Swift Compiler With Incomplete Type Information. In 2025 IEEE Conference on Software Testing, Verification and Validation (ICST). IEEE Computer Society, 2025; .

[5] Recursive descent parser, 2025. URL https://en.wikipedia.org/wiki/Recursive_descent_parser.

[6] Parr T. The Definitive ANTLR 4 Reference. 2nd edition. Pragmatic Bookshelf, 2013. ISBN 1934356999.

[7] Frost R, Launchbury J. Constructing natural language interpreters in a lazy functional language. The Computer Journal 01 1989;32(2):108–121. ISSN 0010-4620. URL https://doi.org/10.1093/comjnl/32.2.108.

[8] Hutton G, Meijer H. Monadic parser combinators. Technical Report NOTTCS-TR-96-4, University of Nottingham, 1996.

[9] Leijen D, Meijer E. Parsec: Direct style monadic parser combinators for the real world. Technical Report UU-CS-2001-27, July 2001. User Modeling 2007, 11th International Conference, UM 2007, Corfu, Greece, June 25-29, 2007.

[10] scala-parser-combinators, 2025. URL https://github.com/scala/scala-parser-combinators.

[11] Wadler P. How to replace failure by a list of successes a method for exception handling, backtracking, and pattern matching in lazy functional languages. In Jouannaud JP (ed.), Functional Programming Languages and Computer Architecture. Berlin, Heidelberg: Springer Berlin Heidelberg. ISBN 978-3-540-39677-2, 1985; 113–128.

[12] Koopman P, Plasmeijer R. A new view on parser combinators. In Proceedings of the 31st Symposium on Implementation and Application of Functional Languages, IFL '19. New York, NY, USA: Association for Computing Machinery. ISBN 9781450375627, 2021; URL https://doi.org/10.1145/3412932.3412938.

[13] Dewey K, Hairapetian S, Gavrilov M. Mimis: Simple, efficient, and fast bounded-exhaustive test case generators. In 2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST). 2020; 51–62.

[14] McKeeman WM. Differential testing for software. Digital Technical Journal December 1998;10(1):100–107.