Practical, General Parser Combinators

Anastasia Izmaylova Ali Afroozeh Tijs van der Storm
Centrum Wiskunde & Informatica, Amsterdam, The Netherlands
{anastasia.izmaylova, ali.afroozeh, tijs.van.der.storm}@cwi.nl

Abstract

Parser combinators are a popular approach to parsing where context-free grammars are represented as executable code. However, conventional parser combinators do not support left recursion, and can have worst-case exponential runtime. These limitations hinder the expressivity and performance predictability of parser combinators when constructing parsers for programming languages.

In this paper we present general parser combinators that support all context-free grammars and construct a parse forest in cubic time and space in the worst case, while behaving nearly linearly on grammars of real programming languages. Our general parser combinators are based on earlier work on memoized Continuation-Passing Style (CPS) recognizers. First, we extend this work to achieve recognition in cubic time. Second, we extend the resulting cubic CPS recognizers to parsers that construct a binarized Shared Packed Parse Forest (SPPF).

Our general parser combinators bring the best of both worlds: the flexibility and extensibility of conventional parser combinators and the expressivity and performance guarantees of general parsing algorithms. We used the approach presented in this paper as the basis for Meerkat, a general parser combinator library for Scala.

Categories and Subject Descriptors D.3.1 [*Programming Languages*]: Formal Definitions and Theory—Syntax; D.3.4 [*Programming Languages*]: Processors—Parsing

Keywords Parser combinators, general parsing, left recursion, continuation-passing style, memoization, higher-order functions

1. Introduction

Parsing is a well-researched topic in computer science. However, there is no "one size fits all" solution for all parsing problems. In particular, all solutions have to find a balance among trade-offs such as expressivity, performance, ease of use, and flexibility. Syntax of programming languages has traditionally been specified using context-free grammars. In parser generators, a grammar is written in a (E)BNF-like notation, which is transformed to parse tables or code. In parser combinators [9, 10], on the other hand, a grammar is encoded using higher-order functions in a programming language, and thus is directly executable.

PEPM'16, January 18–19, 2016, St. Petersburg, FL, USA
ACM. 978-1-4503-4097-7/16/01...
http://dx.doi.org/10.1145/2847538.2847539

Parser combinators are higher-order functions used to define grammars in terms of constructs such as sequence and alternation. The seamless integration with the host programming language makes parser combinators flexible and extensible, compared to parser generators that use a fixed notation for syntax definition. The language developer can define custom combinators using the features of the host programming language. It is also possible to perform data-dependent tasks, such as parsing network protocols and indentation-sensitive languages, by allowing composition of functions that produce parsers based on the result of the previous parse. Monadic parser combinators [9, 10] are often used for this.

Conventional parser combinators are recursive-descent like, and thus have an intuitive execution model, which makes them easy to debug. However, recursive-descent parsers fail to terminate in face of left recursion, and can have worst-case exponential runtime if implemented naively. The lack of support for left recursion is a major problem in expressing natural syntax of programming languages. Most notably, expression grammars, when written in their natural form, are left-recursive.

Grammars of most programming languages do not fit deterministic classes of context-free grammars, such as LR(k) or LL(k), and transforming a grammar to such forms is a tedious process. In addition, maintenance and evolution of deterministic grammars is difficult. There has been extensive research in general parsing algorithms [4, 21, 24], which accept the full class of context-free grammars and deliver all derivation trees in form of a parse forest. There exist worst-case cubic general parsers which are near linear on near-deterministic grammars. Therefore, it is practical to build parsers for programming languages using general parsing, especially in areas where more expressivity is needed, e.g., for developing domain-specific languages (DSLs) and source code analysis.

For decades, general parsing algorithms, more specifically Generalized LR (GLR), have been used in parser generators. Besides standalone GLR parser generators, such as SGLR (used in ASF+SDF Meta-Environment [26]) and Elkhound [15], the popular GNU Bison has also a GLR mode. However, general parsing has not become popular in the world of parser combinators. The main reason is the technical difficulty in realizing general parsers in a combinator style. The underlying machinery of traditional parsing algorithms such as GLR is not composable using the sequence and alternation operators: GLR parsers operate on LR automaton, and each LR state corresponds to multiple positions in grammar rules, and therefore, parsers cannot be directly defined using sequence and alternation.

Earley [4] and Generalized LL (GLL) parsing [21] are different from GLR in the sense that the parser directly operates on grammar rules, rather than an automaton. In particular, GLL parsing has a close relationship with the grammar, similar to recursive-descent parsing. Using Earley's algorithm or GLL, it is possible to define a grammar in a combinator style, and then interpret such a grammar. Such interpretive version of Earley parsing is provided in [20]. In an earlier work [2], we provided an interpretive formulation of GLL

parsing. Such formulations of context-free grammars provide a deep embedding, as the grammar is represented by an algebraic data type.

Deep embedding can benefit from under-the-hood transformations and optimizations. For example, it is possible to calculate first/follow sets and use these information for pruning the search space. However, such parser combinators still have the flavor of a parser generator. Moreover, an extension to such parser combinators may also require modification to the underlying parsing algorithm, for example, the modified Earley sets [11] and modified GLL algorithm [2] to support data dependency. In contrast, parser combinators that provide shallow embedding enable directly executable parsers, as grammars are represented directly as functions. Although certain optimizations are difficult and require access to meta-syntax of the host language, shallow embedding is attractive, since it eases extension and modification through seamless integration with the host programming language.

In this paper we present general parser combinators that combine the expressivity and performance guarantees of state-of-the-art general parsing algorithms with the flexibility and ease of use of conventional parser combinators. Our general parser combinators support the full class of context-free grammars and produce a parse forest in $O(n^3)$ time and space. One key distinction of our approach, compared to interpreter-based general parsing solutions, is that in our approach, like in conventional parser combinators, parsers are directly executable. We believe such a model is a more natural generalization of conventional parser combinators.

The main contribution of this paper is a technique for realizing general parser combinators as a direct embedding in a programming language. We base our general parser combinators on earlier work on memoized Continuation-Passing Style (CPS) recognizers by Johnson [13]. Johnson's approach is a functional formulation of recursive-descent parsing which also provides an elegant solution to the problem of left recursion.

More specifically, our contributions are:

- We modify Johnson's CPS recognizers [13] to obtain the worst-case cubic complexity (Section 2). We show that Johnson's original formulation may require unbounded polynomial time in Appendix A.
- We extend cubic CPS recognizers to fully general parsers by constructing binarized SPPFs [21, 22] (Section 3). These parsers are cubic in time and space, which we prove in Appendix B.
- We evaluate the performance of the resulting parsers using a highly ambiguous grammar and the grammar of Java 7 [8]. The results show worst-case cubic runtime performance on the highly-ambiguous grammar and near linear runtime performance on the grammar of Java (Section 4).

Our general parser combinators are implemented as part of the Meerkat parser combinator library[1]. The Meerkat library provides combinators for lexical disambiguation, layout (whitespace and comment) insertion, EBNF, operator precedence, and execution of semantic actions. Figure 1 shows how an expression grammar can be written very naturally using the Meerkat library. The grammar is disambiguated using declarative disambiguation combinators for operator precedence, such as `|>`, `left`, and `right`. In addition, semantic actions for AST generation are defined using the `&` and `^` combinators. In this paper, however, we do not discuss how Meerkat can be used but instead focus on the underlying parsing technique.

The rest of this paper is organized as follows. Section 2 introduces Johnson's CPS recognizers, and our extension that makes them cubic. In Section 3 we introduce binarized SPPFs, and extend the cubic CPS recognizers to parsers that construct binarized SPPFs

[1] `https://github.com/meerkat-parser`

```
val E
= syn (  right(E ~ "^" ~ E)  & { case x~y => Pow(x,y) }
      |>       "-" ~ E       & { Neg(_) }
      |> left (E ~ "*" ~ E   & { case x~y => Mul(x,y) }
      |        E ~ "/" ~ E   & { case x~y => Div(x,y) })
      |> left (E ~ "+" ~ E   & { case x~y => Add(x,y) }
      |        E ~ "-" ~ E   & { case x~y => Sub(x,y) })
      |       "(" ~ E ~ ")"
      |       "[0-9]".r      ^ { s => Num(toInt(s))   })
```

Figure 1. A natural expression grammar directly encoded in Scala using the Meerkat library.

in cubic time and space. In Section 4 we present the performance results of CPS parsers using a highly ambiguous grammar and the grammar of Java. Section 5 provides a discussion of related work, and Section 6 concludes.

2. General Cubic CPS Recognizers

2.1 Basic Recursive-Descent Recognizers

In this section we introduce *basic* recursive-descent recognizers that use a simple backtracking strategy: the alternatives of a nonterminal are tried in order, and the next alternative is tried only if the current one fails. Figure 2 shows such a formulation[2]. We use basic Scala to explain the semantics of parser combinators as it is expressive enough to enable a concise, executable specification.

A basic recognizer is a function of type `Recognizer`, which is defined as a type alias to function type `Int => Result[Int]` (using the **`type`** keyword) with `Int` as a parameter type and `Result[Int]` as a return type. `Result[T]` is a generic type instantiated with `Int` to represent the result of a recognizer. In essence, a basic recognizer is a *partial* function: it takes an input position and either succeeds, returning the next input position, or fails. Partiality of basic recognizers can be implemented using Scala's monadic `Option[T]`:

```
type Result[T] = Option[T]
def success[T](t: T): Result[T] = Some(t)
def failure[T](): Result[T] = None
```

`Result[T]` is defined as a type alias to `Option[T]`, and two functions, parameterized with type parameter `T`, are provided to compute success and failure. This way, success with the next position `i` is represented by the value constructor `Some`, which takes a value of type `T` and creates a value of type `Option[T]`, e.g., `Some(i)`, and failure by the value constructor `None`.

Basic recognizers can be composed with four combinators (higher-order functions): `terminal`, `epsilon`, `seq` and `rule` (Figure 2). The first two combinators construct basic recognizers for terminals and ε, respectively. For example, `terminal` returns a closure (defined using the `=>` notation) that takes an input position `i` and reports success with the next input position `i + t.length` if terminal `t` matches a substring of the input string starting from `i`, otherwise reports failure. For the sake of presentation, we assume that `input` is globally visible instead of being passed as an argument.

The combinator `seq` is used to encode sequential composition of multiple recognizers. The resulting recognizer chains the given recognizers as long as each of them produces a result, and if any of them fails, the entire composition also fails. The asterisk ($*$) next to the `Recognizer` type permits a variable number of arguments, and therefore, `rs` refers to a sequence of recognizers. `seq` invokes the method `reduceLeft` on the sequence `rs` to reduce this sequence to a

[2] The code snippets used in this paper are available at: `https://github.com/meerkat-parser/cps-parsers`

```
type Recognizer = Int => Result[Int]

def terminal(t: String): Recognizer
  = i => if(input.startsWith(t, i)) success(i + t.length)
         else failure

def epsilon: Recognizer = i => success(i)

def seq(rs: Recognizer*): Recognizer
  = rs.reduceLeft((cur, r) => (i => cur(i).flatMap(r)))

def rule(nt: String, alts: Recognizer*): Recognizer
  = alts.reduce((cur, alt) => (i => cur(i).orElse(alt(i))))
```

Figure 2. Combinator-style recognizers.

new recognizer. At each step, `reduceLeft` applies a binary operator, passed to `reduceLeft` as a closure, to the recognizers in the sequence. For example, `reduceLeft` called on a sequence of three elements a_1, a_2, a_3 with some binary operator f is semantically equivalent to $f(f(a_1,a_2),a_3)$. The binary operator in the definition of `seq` takes two recognizers, `cur` and `r`, and returns a new recognizer. This recognizer takes an input position `i` and first calls `cur` at `i`. Then, if `cur` succeeds at `i`, e.g., by returning a value `Some(j)` with the next input position `j`, the recognizer calls `r` at `j` returning the result, otherwise the recognizer fails returning `None`. The `flatMap` method defined in type `Option[T]` enables such composition of basic recognizers, systematically accounting for partiality.

The combinator `rule` is used to define a nonterminal with head `nt` and alternatives `alts`. To recognize a nonterminal at an input position using basic recognizers, the alternatives of the nonterminal are tried at the input position until one of them succeeds. `rule` invokes the method `reduce` on a sequence of recognizers, `alts`, to reduce alternatives to a new recognizer by applying an associative binary operator at each step. The binary operator takes two recognizers, `cur` and `alt`, and returns a recognizer that given an input position `i`, first calls `cur` at `i`, and if it succeeds, returns its result, otherwise, calls `alt` at `i`. This semantics of handling failure is provided by the `orElse` method defined in type `Option[T]`. The `orElse` method uses the call-by-name evaluation strategy in its argument so that if `cur` succeeds at `i`, the expression `alt(i)` is not evaluated.

Given the combinators of Figure 2, a recognizer for $A ::= a$ can be directly defined as follows:

```
val A = rule("A", terminal("a")) // A ::= a
```

where `A` is a variable to which the resulting recognizer is assigned. This formulation, however, can be problematic when recursive definitions are required. For example, consider the grammar $S ::= aSb \mid aS \mid s$ defined as follows:

```
val S = rule("S",
      seq(terminal("a"), S, terminal("b")), // S ::= a S b
      seq(terminal("a"), S),                //     | a S
      terminal("s"))                        //     | s
```

In a programming language with strict evaluation, this recursive definition is not well-defined: when the defining expression on the right-hand side of the assignment, which recursively uses variable `S`, is evaluated, `S` is unbound. One way to solve this problem is to use a fix-point combinator `fix`, defined as:

```
def fix[A,B](f: (A=>B)=>(A=>B)): A=>B =
               { lazy val p: A=>B = f(t => p(t)); p }
```

Here we use the definition of `fix` that can be used in languages with strict evaluation. This is also reflected in the type signature. Using `fix`, the recognizer for S can be defined as:

```
type Result[T] <: MonadPlus[T, Result]
def success[T](t: T): Result[T]
def failure[T]: Result[T]

trait MonadPlus[T, M[_] <: MonadPlus[_, M]] {
    def map[U](f: T => U): M[U]
    def flatMap[U](f: T => M[U]): M[U]
    def orElse(r: => M[T]): M[T]
}
```

Figure 3. Monadic `Result[T]`.

```
val p = fix(S => rule("S",
      seq(terminal("a"), S, terminal("b")), // S ::= a S b
      seq(terminal("a"), S),                //     | a S
      terminal("s")))                       //     | s
```

The resulting recognizer is the fix point of the function passed to `fix` as a closure. This way, the recursive structure of S is encoded as an anonymous recursive function that is assigned to variable `p`. The use of `S` in the body of the closure replaces the recursive uses of the recognizer in the previous definition.

As illustrated, recognizers are directly constructed using combinators, resembling the grammar. In the next section, we generalize this framework to allow results other than `Option[Int]`. In particular, we reuse the definitions of Figure 2 *as is* to obtain continuation-passing style recognizers, and later parsers.

2.2 Full Backtracking Using Continuation-Passing Style

The first problem with basic recursive-descent recognizers is that backtracking is local to a nonterminal. As a result, the order of alternatives may influence the recognition of a sentence. For example, a recognizer for the grammar $A ::= a \mid ab$ reports failure when recognizing the input string `"ab"`, although it is apparent that the second alternative matches `"ab"`. The reason for failure is that the first alternative reports success, and the second alternative is never tried, while there is an unmatched `b` left. The second problem is that basic recognizers only return a single derivation of the input string, which depends on the order of alternatives. The grammar $S ::= aSb \mid aS \mid s$, for example, can derive `"aasb"` in two different ways, corresponding to the following leftmost derivations:

1. $S \Rightarrow aSb \Rightarrow aaSb \Rightarrow aasb$
2. $S \Rightarrow aS \Rightarrow aaSb \Rightarrow aasb$

However, basic recognizers can only deliver one. To support the full class of context-free grammars, basic recursive-descent recognizers require exhaustive search, and therefore, need full backtracking. In such a setting, a recognizer can potentially succeed multiple times at the same input position. To introduce full backtracking into basic recursive-descent recognizers, the recognition functions have to be adapted to produce multiple values.

One way to approach this is to use Continuation-Passing Style (CPS). In fact, to transform basic recognizers into CPS recognizers, it is sufficient to only redefine `Result[T]` and accompanying functions `success` and `failure`. Indeed, the combinators of Figure 2 can be used with any `Result[T]` that defines how to compose two functions via `flatMap` and how to combine two results via `orElse`, so that the details specific to `Result[T]`, such as partiality of basic recognizers, are systematically managed by `flatMap` and `orElse`.

The `MonadPlus` trait in Figure 3 specifies a monadic interface: `map`, `flatMap` and `orElse`. The type constraint, expressed using `<:`, requires `Result[T]` to define the methods of the interface. Note that `=>`, used before the parameter type in `orElse`, indicates call-by-name evaluation in its argument. This can also be achieved by explicitly constructing a closure. In addition, `success` defines how to

```
type K[T] = T => Unit // Continuation type

trait Result[T] extends (K[T] => Unit)
                with MonadPlus[T, Result] {

    def map[U](f: T => U): Result[U]
               = result(k => this(t => k(f(t))))

    def flatMap[U](f: T => Result[U]): Result[U]
               = result(k => this(t => f(t)(k)))

    def orElse(r: => Result[T]): Result[T]
               = { lazy val v = r;
                   return result(k => { this(k); v(k) }) }
}

def success[T](t: T): Result[T] = result(k => k(t))
def failure[T](): Result[T] = result(k => {/*do nothing*/})

def result[T](f: K[T] => Unit): Result[T]
           = new Result[T]{ def apply(k: K[T]) = f(k) }
```

Figure 4. `Result[T]` for CPS recognizers.

```
1  def memo[T](f: Int => Result[T]): Int => Result[T] = {
2    val table: Map[Int, Result[T]] = HashMap.empty
3    return i => table.getOrElseUpdate(i, memo_result(f(i)))
4  }

6  def memo_result[T](res: => Result[T]): Result[T] = {
7    val Rs: MutableList[T]    = MutableList.empty
8    val Ks: MutableList[K[T]] = MutableList.empty
9    return result(k =>
10     if (Ks.isEmpty) { // Called for the first time
11       Ks += k
12       val k_i: K[T] = t => if (!Rs.contains(t)) {
13                              Rs += t
14                              for (kt <- Ks) kt(t)
15                            }
16       res(k_i)
17     } else {          // Has been called before
18       Ks += k
19       for (t <- Rs) k(t)
20     })
21 }
```

Figure 5. Memo functions for CPS recognizers.

lift a value to the one of type `Result[T]` (basic computation) while `failure` defines *zero* (computation with no value).

A CPS recognizer is a function of the same type as in Figure 2, but with `Result[T]` in Figure 4, i.e., defined as the continuation monad [27]. In Figure 4, `K[T]` represents a continuation type defined as a type alias to `T => Unit`[3]. Now, `Result[T]` is a function type, a subtype of `K[T] => Unit`, which also defines the methods of the `MonadPlus` interface (we discuss them later).

The helper method `result` (on the bottom of Figure 4) is used to define values of type `Result[T]` using ordinary functions of type `K[T] => Unit`. In Scala, a type can extend a function type, and a function is an object that has an `apply` method, e.g., if `f` is of type `Int => Int`, `f(0)` is equivalent to `f.apply(0)`. Given a function `f` of type `K[T] => Unit`, the `result` method creates an instance of `Result[T]`, say `g`, such that the result of `g(k)` is equal to the result of `f(k)`.

A CPS recognizer takes an input position and returns a function of type `Result[Int]`. The returned function takes a continuation of type `K[Int]` and returns `Unit`. A continuation is a function, now additionally passed to recognizers, that represents the "rest" of the recognition process. Instead of directly returning a value, a recognizer "returns" success by calling its continuation with the next input position, as in `success`, and fails by not calling its continuation, as in `failure`. For example, given a CPS recognizer for a terminal `val f: Recognizer = terminal("a")` and an initial continuation `val k0: K[Int] = i => println("success: " + i)`, the result of evaluating `f(0)(k0)` is either `"success: 1"`, printed to the console if the input string starts with `"a"`, where `1` is the next input position, or nothing otherwise.

Similar to basic recognizers, CPS recognizers can be composed using the combinators of Figure 2, but now defined in terms of `flatMap` and `orElse` of Figure 4. In the definition of `seq`, given two CPS recognizers, `cur` and `r`, the result of their composition using `flatMap` is a CPS recognizer that given an input position `i`, first calls `cur` at `i`, as before, but now returns a function of type `Result[Int]`. This function, given a continuation `k`, first creates a new continuation `t => r(t)(k)` and then passes this continuation to the result of `cur(i)`, so that the second recognizer, `r`, is called via this continuation when the recognizer `cur` succeeds at `i` with a new input position. In other words, `seq` now constructs a continuation-passing chain of function calls, one for each given recognizer.

In the definition of `rule`, given two CPS recognizers, `cur` and `alt`, a new CPS recognizer, returned by the binary operator, sequentially calls `cur` and `alt` at an input position `i` and combines the results of `cur(i)` and `alt(i)` using `orElse`, so that when the new CPS recognizer is called at `i` with a continuation, say `k`, `k` is passed to both results, i.e., `cur(i)(k)` and `alt(i)(k)`. This way, `rule` tries all the alternatives. Note that in the definition of `orElse`, the use of variable `v` with keyword **`lazy`** ensures that the argument to `orElse` is (lazily) evaluated once.

CPS recognizers support full backtracking as the `rule` combinator always tries all of its arguments. The runtime behavior of such recognizers is exponential in the worst case. Furthermore, these recognizers will fail to terminate in face of left-recursive rules. Since Norvig's work on memoization in top-down parsing [18], it is known that memoizing recognizers brings down the exponential runtime performance to polynomial. However, this type of memoization does not solve the problem of left recursion. In the next section we introduce Johnson's approach on memoized CPS recognizers [13] that solves the problem of left recursion.

2.3 Support for Left Recursion

Neither the basic recognizers of Section 2.1, nor the CPS recognizers of Section 2.2 support left-recursive rules. Consider the following left-recursive recognizer:

```
val A = fix(A => rule("A",
   seq(A, terminal("a")), terminal("a"))) // A ::= A a | a
```

The call to `A` at input position `0` leads to unbounded number of recursive calls at the same input position, as the recursive calls do not change the function's state and never reach a base case.

The memo functions of Figure 5 turn an arbitrary CPS recognizer into a memoized CPS recognizer. The `memo` function, when applied to a recognizer, returns a new recognizer that consults the memo table each time it is called at an input position `i`. If the memoized recognizer has not been yet called at `i`, the result of calling the original, unmemoized recognizer, `f`, at `i` is memoized, `memo_result(f(i))`, and returned after updating the memo table. Due to the call-by-name nature of `memo_result` (`=>` before the parameter type), `f(i)` is not evaluated at this moment. This ensures that `f` is called at `i` at most once. If the memoized recognizer has been called at `i` before, its

[3] `Unit` is equivalent to type `void` in other programming languages, e.g., Java.

```
def rule(nt: String, alts: Recognizer*): Recognizer
  = memo(alts.reduce((cur, alt) =>
                       i => cur(i).orElse(alt(i))))
```

Figure 6. Memoizing CPS recognizers.

result is taken from the table. Note that the `getOrElseUpdate` method uses the call-by-name evaluation strategy in its second argument, so that it is not evaluated if the key is found.

A memoized result, returned when `memo_result` is applied to the result of calling an unmemoized recognizer `f` at an input position `i`, has access to two lists: `Rs` and `Ks` (lines 7–8). The result list `Rs` stores all input positions produced by the unmemoized recognizer when it succeeds at `i`, and the continuation list `Ks` stores all continuations passed to the memoized recognizer when it is called at `i`. If the memoized result is called for the first time (`Ks.isEmpty` in line 10), the current continuation `k` is added to `Ks`, and the original, unmemoized result, `res`, is called with a new continuation `k_i` (defined in lines 12–15). Note that `f(i)` is evaluated at this moment (line 16), and therefore, `f` can be called at `i` at most once. Also, `k_i` is created only upon the first call to the memoized recognizer at `i`. Each time the unmemoized recognizer succeeds at `i` with an input position, `k_i` checks whether this input position has been seen before and if not (`!Rs.contains(t)`), first records it in `Rs`, and then, runs all the continuations recorded so far in `Ks` at this input position (for-loop, line 14). On the other hand, if the memoized result has been called before (else-branch, lines 17–20), the current continuation `k` is added to `Ks` and is called for each input position recorded in `Rs`.

To add memoization to CPS recognizers, the `rule` combinator needs to be re-defined as in Figure 6. Now, when a memoized left-recursive CPS recognizer is called at an input position `i`, its termination is guaranteed as the respective unmemoized recognizer (`f` in the body of `memo`) will never be called at `i` more than once. At the same time, the part of the execution path, which led to a left-recursive call and can produce new input positions for the left-recursive recognizer at `i`, is effectively recorded as continuations. A continuation is recorded for the left-recursive call, and, in case of indirect left recursion, for each call to a memoized recognizer at `i` that indirectly led to the left-recursive call. Each continuation captures the next step in the alternative after the current call returns, and a continuation defined in lines 12–15 is called at the end of each alternative. These continuations will be run (re-trying the terminated path) for any input position produced by the left-recursive recognizer at `i`, and as long as new input positions are produced.

Intuitively, memoizing CPS recognizers does not reduce the number of execution paths as all the continuations passed to a memoized CPS recognizer at an input position will be recorded in the continuation list, and each of the recorded continuations will be run for each result produced by the recognizer at this input position. This follows from the definition of `memo_result`. In the if-branch, all recorded continuations are invoked on a newly produced result. In the else-branch, all existing results are input to a new continuation.

2.4 Memoization on Continuations

In Appendix A.3 we show that the execution of memoized CPS recognizers of Figure 2 and 6 can require $O(n^{m+1})$ operations, where m is the length of the longest rule in the grammar. The reason for such unbounded polynomial behaviour is that the same continuation can be called multiple times at the same input position. As illustrated in Appendix A.3, this happens when the same continuation is added to different continuation lists that are associated with calls made to the same recognizer but at different input positions. If the recognizer produces the same input position starting from different input positions, duplicate calls are made. The duplicate calls further result in adding the same continuations to the same continuation lists multiple times.

```
def memo_k[T](k: T => Unit): T => Unit = {
  val s: Set[T] = HashSet.empty
  return t => if(!s.contains(t)) { s += t; k(t) }
}
```

Figure 7. Memoization on continuations.

```
trait Result[T] extends (K[T] => Unit)
                with MonadPlus[T, Result] {

    def map[U](f: T => U)
               = result(k => this(memo_k(t => k(f(t)))))

    def flatMap[U](f: T => Result[U])
               = result(k => this(memo_k(t => f(t)(k))))

    def orElse(r: => Result[T])
               = { lazy val v = r
                   return result(k => { this(k); v(k) }) }
}
```

Figure 8. `Result[T]` extended with memoization on continuations.

To reduce the worst-case complexity to cubic, the duplicate calls to continuations need to be eliminated. To achieve this, we extend the memoization strategy by adding memoization on continuations, as in Figure 7, and by re-defining `Result[T]` as in Figure 8. A memoized continuation consults the set of already passed arguments (Figure 7), input positions in case of recognizers, and runs the unmemoized continuation, `k`, only when it has not been called before with the current input position. The memoization on continuations prevents the same execution path to be explored more than once, where an execution path is identified by a grammar position, the input position of the parent nonterminal and the current input position. Note that continuations `k_i` in Figure 5, lines 12–15, have been already defined with the memoization semantics.

In the next section we explain how CPS recognizers can be extended to parsers that construct binarized SPPFs. In Appendix B we show that memoization on continuations is also sufficient to keep the cubic bound for such CPS parsers.

2.5 Trampoline

In our parser combinators, when one continuation calls another continuation, the number of calls in the call stack may exceed the default size of the JVM stack. To avoid stack overflow, we can turn the calls into a trampoline-style loop. Using a trampoline, calls are handled in a loop over a custom stack data structure, ensuring that the JVM stack does not grow too large. To implement a trampoline, we create and pass around an object of type `Trampoline`. This object maintains a custom stack of values that represent calls and runs a loop over this stack. In the definition of `orElse` and `memo_result`, instead of calling a continuation or a function of type `Result[T]`, a value representing the call is pushed on top of the stack in the trampoline object. When the main loop runs, a value is popped from the stack, and the actual call represented by this value is made. Parsing terminates when there are no elements left in the stack.

3. SPPF Construction

3.1 Binarized SPPF

General context-free parsing algorithms explore all derivations of a sentence. To deal with potentially exponential number of parse trees,

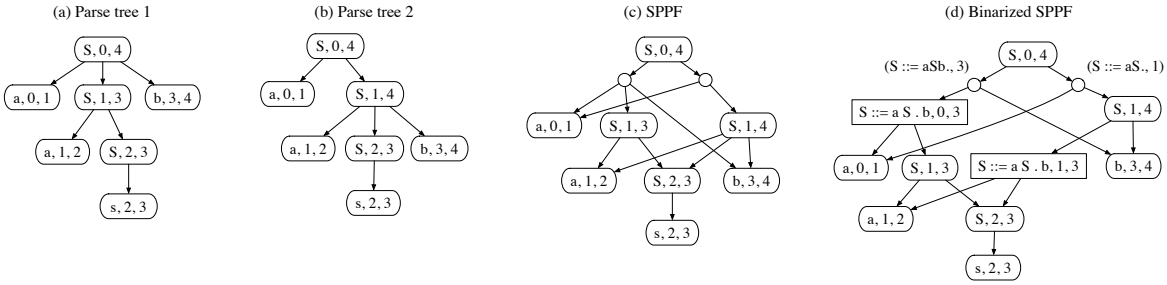

Figure 9. Two parse trees (a) and (b), their corresponding SPPF (c), and the binarized SPPF version (d).

common subtrees are shared to form a Shared Packed Parse Forest (SPPF), introduced by Tomita [24]. For example, the two parse trees, resulting from parsing `"aasb"` using the grammar $S ::= aSb \mid aS \mid s$, and their corresponding SPPF are shown in Figure 9.

In an SPPF, there are two types of nodes: symbol nodes and packed nodes. Symbol nodes are of the form (x, i, j) where x is the name of a nonterminal, terminal or epsilon, and i and j are the left and right extents, indicating the start and end positions in the input string recognized by x. Packed nodes, shown by small circles in Figure 9, represent a derivation. When there is ambiguity, e.g., under the root node in Figure 9 (c), multiple packed nodes are present, each identifying a derivation.

Johnson [12] showed that any parsing algorithm that produces a Tomita-style SPPF has $O(n^{m+1})$ worst-case runtime, where m is the length of the longest rule in the grammar. Therefore, in order to have general parsing in $O(n^3)$ time, which is common in general recognizers, rules need to be of length at most two. Although a grammar can be transformed to a grammar with rules of length at most two, this transformation leads to a large grammar, with many extra nonterminals. This affects the maintainability of the grammar and the parsing performance [22].

To enable general context-free parsing in $O(n^3)$ without transforming the grammar, Scott and Johnstone [22] introduced binarized SPPFs, which have additional *intermediate* nodes. Intermediate nodes are of the form (L, i, j), where L is a grammar position, and i and j are the left and right extents. Grammar positions for intermediate nodes are of the form $A ::= \alpha \cdot \beta$ where $|\alpha| \geq 2$. The binarized version of the SPPF is shown in Figure 9 (d). Intermediate nodes, similar to nonterminal nodes, can be ambiguous. In this case, they have more than one packed node as children.

Packed nodes in a binarized SPPF are of the form (L, k), where L is a grammar position, and k, *pivot*, is an input position. For packed nodes under a nonterminal A, L is of the form $A ::= \alpha \cdot$, where α is an alternative of A. For example, in the binarized SPPF of Figure 9 (d), the left and right packed nodes under the root node have labels $(S ::= aSb \cdot, 3)$ and $(S ::= aS \cdot, 1)$, respectively. For packed nodes under an intermediate nodes, L is the same as the grammar position of the intermediate node. Moreover, k is equal to the left extent of the packed node's right child. Packed nodes can have at most two children, which are non-packed nodes.

3.2 SPPF Construction for Cubic CPS Parsers

To extend CPS recognizers to parsers that produce binarized SPPFs, we redefine `terminal`, `epsilon`, `seq`, and `rule` as in Figure 10. Now, these combinators build parsers of type `Parser`, which is a function that takes an input position and returns a non-packed (symbol or intermediate) node as `Result[NonPackedNode]`.

The SPPF creation is delegated to an instance of `SPPFLookup` (`sppf`), which we assume is globally visible. The `SPPFLookup` interface provides methods that ensure sharing SPPF nodes. Each method of `SPPFLookup` first searches for an existing SPPF node with the given arguments and either returns an existing node, or creates a new one. The `getNonterminalNode` and `getIntermediateNode` methods take two non-packed nodes as arguments, which will be attached to the returned node via a packed node.

Each combinator returns a parser that is responsible for creation of a specific type of an SPPF node. A terminal node is created via `getTerminalNode` which takes the name of the terminal, the input position at which the function is called, and the next input position corresponding to the end of the matched terminal. Epsilon (`epsilon`) nodes have the same left and right extents, both being equal to the input position at which the parser is called.

The `seq2` combinator constructs a parser that creates an intermediate node based on the results of its two operands, `p1` and `p2`. An intermediate node is created via the `getIntermediateNode` method, which takes a label and two non-packed nodes, `t1` and `t2`, returned by the parsers `p1` and `p2`, respectively. Note that `seq2` uses its resulting parser, `q`, as the label of the intermediate node, and as this definition is recursive, the fix point combinator is used.

Finally, the `rule` combinator first iterates over a sequence of parsers (by calling `map` on `alts`) to create parsers (by applying `rule1` to each parser of the sequence) that are responsible for creating nonterminal nodes. Then, `rule` reduces the resulting sequence of parsers. At the end of an alternative, a nonterminal node labeled `nt` is created. This is done by calling `getNonterminalNode` with the name of the nonterminal, the label of its packed node, and the non-packed node produced by `p` at `i`. Note that `rule1` uses its resulting parser, `q`, as the label of the packed node, and since this definition is recursive, the fix point combinator is used.

Sharing non-packed nodes relies on identifying non-packed nodes by their label and left and right extents. The label of a nonterminal or terminal node is a string, while the label of an intermediate node corresponds to a grammar position. In a parser generator setting, the labels of intermediate nodes can be determined by processing the grammar. In a parser combinator setting, on the other hand, no such preprocessing step exists, and labels corresponding to grammar positions should be dynamically determined. We use the identity of the parser objects resulting from `seq2` as labels of intermediate nodes (variable `q`) to effectively encode grammar positions. For example, for $S ::= aSb \mid aS \mid s$, which is defined as

```
val S = fix(S =>
            rule("S", seq(terminal("a"), S, terminal("b")),
            seq(terminal("a"), S),
            terminal("s")))
```

the parsers resulting from applying `seq2` to the first two symbols, `terminal("a")` and `S`, in the first and second alternatives represent the unique grammar positions $S ::= aS \cdot b$ and $S ::= aS \cdot$, respectively.

3.3 Semantic Actions and Generation of ASTs

Binarized SPPFs are part of the internal machinery of a general parser, and are not intended for the end user. To provide a user-

```
type Parser = Int => Result[NonPackedNode]

def terminal(t: String): Parser = i => if(input.startsWith(t, i)) success(sppf.getTerminalNode(t, i, i + t.length))
                                       else failure

def epsilon: Parser = i => success(sppf.getEpsilonNode(i))

def seq(ps: Parser*): Parser = ps.reduceLeft(seq2)

def rule(nt: String, alts: Parser*): Parser = memo(alts.map(rule1(nt, _)).reduce((cur, p) => (i => cur(i).orElse(p(i)))))

private def seq2(p1: Parser, p2: Parser): Parser
  = fix(q => (i => p1(i).flatMap(t1 => p2(t1.rExtent).map(t2 => sppf.getIntermediateNode(q, t1, t2)))))

private def rule1(nt: String, p: Parser): Parser = fix(q => (i => p(i).map(t => sppf.getNonterminalNode(nt, q, t))))
```

Figure 10. Extension of CPS recognizers to parsers that construct binarized SPPF.

friendly format for processing parsing results, the Meerkat library supports conversion of binarized SPPFs to terms that reflect the underlying grammar. This approach is popular in tools that allow algebraic specification and term rewriting, for example in ASF+SDF [26]. Figure 11 shows a visualization of the terms corresponding to the binarized SPPF in Figure 9 (d).

The traversal of binarized SPPFs and generation of terms is straightforward. As shown in Section 3.1, packed nodes under nonterminal nodes have labels which corresponds to grammar positions of the form $A ::= \alpha\cdot$. The type of a term is computed as the result of sequencing parsers (`seq` and `rule1`) and is stored in the packed nodes of a nonterminal. We traverse the binarized SPPF bottom-up, and for each SPPF node type, perform a specific action.

For terminal nodes, a terminal term is created that stores the name of the terminal and its associated matched string. For nonterminal or intermediate nodes that are ambiguous, i.e., have more than one packed node, an ambiguity term is created. An ambiguity term gets a list of terms as its children. For nonterminal nodes, a nonterminal term is created. We bypass intermediate nodes, as they do not correspond to any constructs in the grammar and should not appear in final terms. This effectively flattens the intermediate nodes.

Besides creation of terms from a binarized SPPF, we support execution of semantic actions. Due to the inherent nondeterminism in general parsing, many parsing paths will eventually die. Therefore, it is desirable to postpone the execution of semantic actions until parsing is done. In the Meerkat library, semantic actions are stored in the packed nodes of an SPPF and executed post-parse by traversing the resulting binarized SPPF. An example of using semantic actions in the Meerkat library is shown in Figure 1.

The traversal mechanism for semantic actions is basically the same as for building terms, with the difference that we throw an exception when an ambiguous node is encountered. In most cases, an ambiguity is a sign of error in the grammar, and the user should first resolve the ambiguity, e.g., by investigating the terms corresponding to the ambiguous parse, and then run the semantic actions.

4. Evaluation

In this section we evaluate the performance of CPS parsers, as implemented in the Meerkat library. We use the highly ambiguous grammar Γ_3, $S ::= SSS \mid SS \mid b$, and the grammar of Java. The results show that Meerkat parsers are cubic on the highly ambiguous grammar, and behave nearly linearly on the Java grammar. The experiments were carried out on a machine running Mac OS X 10.9.4 on a quad-core Intel Core i7 2.6 GHz CPU with 16 GB of memory. The Meerkat library was compiled with Scala 2.11.2 and ran on a 64-Bit Oracle HotSpot^{TM} JVM version 1.7.0_55. The

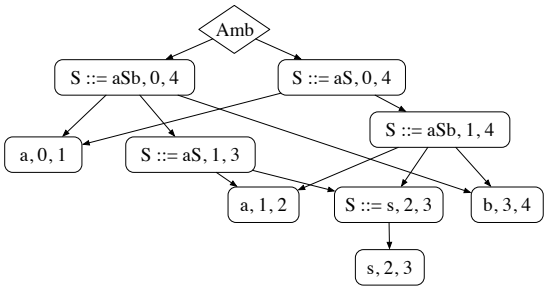

Figure 11. Terms for the binarized SPPF of Figure 9 (d).

reported time is the mean running time (CPU user time) of 10 runs for each parse. To allow for JIT optimizations, the first three runs for each file were skipped.

4.1 Parsing Γ_3

To evaluate the runtime performance of CPS parsers in the worst case, we ran the Meerkat parser for Γ_3 on sequences of b's, varying from 10 to 500. Γ_3 triggers the worst-case behavior for CPS parsers. Standard GLR parsers, except for BRNGLR that produces binarized SPPFs, are $O(n^4)$ on Γ_3. The results are shown in Figure 12. As can be seen, the resulting curve is cubic with high confidence, as indicated by the R^2 value of 0.9998.

4.2 Parsing Java

To evaluate the performance of CPS parsers on grammars of real programming languages, we have chosen the grammar of Java 7 from the main part of the Java Language Specification [8]. This grammar has a left-recursive unambiguous expression grammar that encodes operator precedence by introducing new nonterminals. We ran the parser for the character-level Java grammar[4] for 7449 Java files in the source release of JDK 1.7.0_60-b19. All files were parsed successfully and without ambiguity. Figure 13 shows the running time corresponding to the execution of the parser for the character-level Java grammar for increasing input sizes. We use a log-log (base 10) plot. The goodness of the fit is indicated by the R^2 value of 0.9828. The regression line equation (log-log scale) is written in the plot. As the regression is calculated after a log transformation of the original data, and the coefficient is close to one (1.138201), we can conclude that the running time for Java is near linear ($y \approx x^{1.138201}$).

[4] `https://github.com/meerkat-parser/grammars`

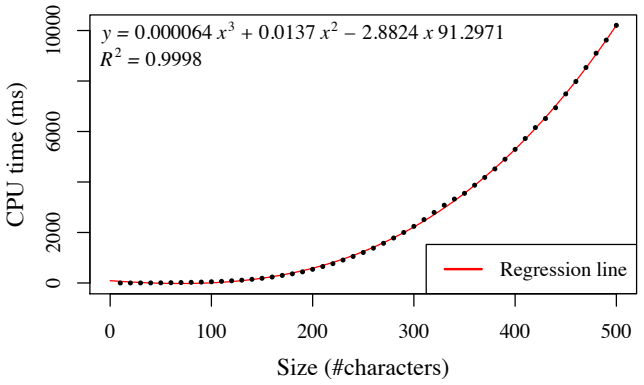

Figure 12. Runtime of parsing string of b's using Γ_3.

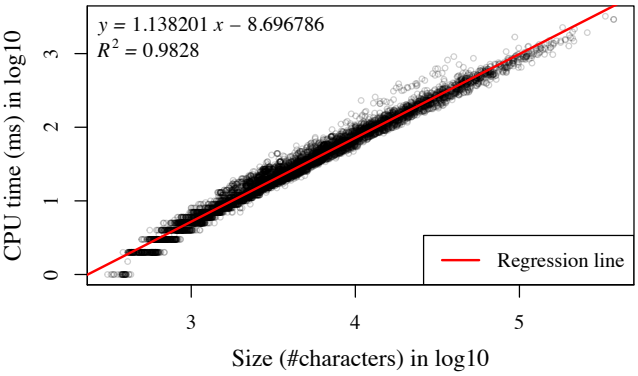

Figure 13. Runtime of parsing Java files.

5. Related Work

Conventional parser combinators are recursive-descent like. Therefore, a natural choice for generalizing parser combinators to support all context-free grammars is to generalize recursive-descent parsing. The main challenge is support for left recursion. In this section we discuss a number of parsing techniques that generalize recursive-descent parsing, with a focus on how support for left recursion is provided. We discuss each work based on the following aspects:

1. The mechanism used to support direct left recursion.
2. Support for indirect/hidden left recursion: not supported, requires extra mechanism, or *uniformly* works with the mechanism for direct left recursion.
3. The worst-case runtime complexity of the recognizer and parser.
4. The output of a parser (single parse tree or a parse forest).

Table 1 gives an overview of related work based on these aspects. Note that there are other parser combinator tools, e.g., Parsec [14] and the Scala parser combinator library [16], which are essentially basic recursive-descent recognizers (see Section 2), and we do not discuss them in this section.

5.1 Left-Recursion Curtailment Using the Input Length

Frost *et al.* [7] present an approach for supporting direct left recursion based on the length of the input. In this approach, the number of calls to recognizers at each input position is maintained. For non-left-recursive recognizers, the count will be at most one. For left-recursive ones, the count increases by each recursive call at the same input position. When the count exceeds the number of remaining tokens in the input string plus one, the call is curtailed, as no successful parse is possible at this point.

In this approach, each left-recursive recognizer can be called at each input position at most n times, where n is the length of the input. This brings the worst-case complexity of this approach to $O(n^4)$, compared to the expected $O(n^3)$ complexity. In addition, this approach requires extra machinery to accommodate indirect left recursion. Since the parser version of Frost *et al.* 's approach creates a Tomita-style SPPF, it is of unbounded polynomial complexity.

One essential difference between Frost *et al.* 's approach and CPS parsers is the moment when a chain of left-recursive calls is terminated. In CPS parsers, this happens when the second call to a recognizer at the same input position is made. Then, the results for left-recursive recognizers are effectively computed in a loop: as long as a new result is produced, the terminated parsing paths, recorded as continuations, are restarted at the new input position. As a result, handling left-recursive rules is more efficient in CPS parsers. Finally, it should be noted that Frost *et al.* 's approach cannot be used in cases where the length of the input is not known, for example, when reading from a network socket.

5.2 Left Recursion in PEGs

Parsing Expression Grammars (PEGs) [6] are an alternative to context-free grammars, where the unordered alternation operator is replaced with a *prioritized choice* operator. PEGs are commonly implemented as recursive-descent parsers with local backtracking: the alternatives of a nonterminal are tried in order, and the next alternative is tried only if the current one fails. As a result, PEGs produce at most one parse tree and cannot be ambiguous. PEGs suffer from the quirk that if an alternative is a prefix of another one, the second alternative is never tried. This, for example, leads to parse error on `"ab"` for the grammar $A ::= a|ab$, although it is apparent that the second alternative can correctly parse this input.

Packrat parsing [5] uses memoization to implement PEGs in linear time. However, Packrat parsers, like other recursive-descent parsers, do not support left recursion. Warth *et al.* [28] propose a mechanism to support left recursion by modifying the memoization in Packrat parsing. In this approach, a special fail value is put into the memo table before calling a parser at an input position for the first time, so that left-recursive calls fail. This ensures termination of the parser for a left-recursive nonterminal. Then, if any of the non-left-recursive alternatives can produce a result, the parser is restarted to re-try the left-recursive ones. As long as a new result is produced, the new result replaces the previous one in the memo table, and the parser is called again at this input position, reusing the last result from the memo table for the left-recursive calls. This process is called *growing the seed*.

For indirect left recursion, this approach uses an extra data structure, called *rule invocation stack*, to maintain the recursive calls between mutually left-recursive nonterminals. Warth *et al.* 's approach breaks the linear runtime guarantee of Packrat parsing, as for some left-recursive grammars, the runtime complexity of this approach is $O(n^2)$ [28]. Tratt [25] identifies a problem with Warth *et al.* 's approach for rules that are both left and right recursive, e.g., $E ::= E + E$. For such rules, this approach is biased towards producing a right-associative derivation, which does not conform to the semantics of PEGs.

5.3 Cancellation Parsing

Cancellation parsing [17] is a technique to support left recursion for Definite Clause Grammars (DCGs) in Prolog. The basic idea behind this technique is that each call to a nonterminal takes a set of already

Table 1. Overview of related work that extend recursive-descent parsing towards a general parsing solution.

Approach	Mechanism	Indirect/Hidden	Worst-case Complexity		Output
			Recognizer	Parser	
Left-recursion curtailment	Input-length heuristics	extra	$O(n^4)$	unbounded polynomial	Tomita-style SPPF
Left recursion in PEGs	Memoization and growing the seed	extra	$O(n^2)$	$O(n^2)$	Single parse tree
Cancellation parsing	Passing cancellation sets	extra	exponential	exponential	Single parse tree
ANTLR 4	Left-recursion elimination by rewriting	no	$O(n^4)$	$O(n^4)$	Single parse tree
GLL	Cycles in the GSS	uniform	$O(n^3)$	$O(n^3)$	Binarized SPPF
CPS parsers	Memoization in CPS	uniform	$O(n^3)$	$O(n^3)$	Binarized SPPF

called nonterminals, the cancellation set. If the nonterminal is already in the set, the parser backtracks and tries the next alternative, otherwise the nonterminal is added to the cancellation set. This guarantees termination in presence of left recursion. To construct the parse trees corresponding to the terminated paths, for each left-recursive nonterminal A, a special token $\overline{A}$ is put before the rest of the token stream, using `untoken`, and a rule $A ::= \overline{A}$ is added to the grammar. After inserting $\overline{A}$, nonterminal A is called again with the current cancellation set.

```
A(c) ::= [A ∉ c] A(A ∪ c) 'a' untoken(Ā)  A(c)
       | 'a' untoken(Ā) A(c)
       | Ā        // Added rule for the inserted tokens
```

This approach works for direct and indirect left recursion, but does not work for hidden left recursion, i.e., when a left-recursive call is hidden behind a nullable nonterminal, e.g., $A ::= BAa$ and $B \overset{*}{\Rightarrow} \varepsilon$. To support hidden left recursion, this approach requires grammar analysis to identify nullable nonterminals, and pass a boolean flag to each call. The need for grammar analysis for hidden left recursion and customization of left-recursive definitions (`untoken` and rules for $\overline{A}$) makes this approach fundamentally different from ours. Finally, this parsing technique is designed to have no side effects [17], and does not memoize previous results, thus is of worst-case exponential complexity.

5.4 ANTLR 4

ANTLR 4 [19] supports direct left-recursive rules by rewriting them to non-left-recursive ones. During this rewriting process, ANTLR inserts semantic predicates to resolve ambiguities based on the order of alternatives. The resulting parsers mimic the operator precedence technique by Clarke [3]: the rules that come earlier have higher precedence, and all rules are left-associative by default, unless explicitly marked as right-associative. ANTLR 4 does not support indirect left recursion because rewriting grammars to eliminate indirect left-recursion results in large grammars that have no obvious relationship with the original ones.

ANTLR 4 uses the Adaptive LL(*), ALL(*), strategy, in which a sub-parse is invoked for each alternative of a nonterminal and intermediate results are cached. ALL(*) effectively uses global backtracking to avoid the problems with PEG-style backtracking of previous versions of ANTLR. ALL(*) parsers have worst-case complexity of $O(n^4)$ and produce at most one derivation, since ambiguities are resolved during parsing. The ambiguity resolution mechanism is based on the order of alternatives, in which the sub-parse with the lowest alternative number (appearing earlier) is preferred. Because of complex grammar transformations performed by ANTLR 4 before parsing, and its lack of direct support for left recursion, this parsing strategy is not suitable for parser combinators.

5.5 GLL Parsing

Generalized LL (GLL) [21] is a fully general, worst-case cubic parsing algorithm. GLL uses a Graph Structured Stack (GSS) that handles multiple function call stacks, and produces binarized SPPFs. The problem of left recursion (direct/indirect/hidden) is uniformly solved by allowing cycles in the GSS. GLL parsers are recursive-descent like, and have a close relationship with the grammar.

Among all the related work we discussed so far, GLL parsing is the closest to our work, especially if we consider a version of GLL [2] which uses a more efficient GSS [1]. In fact, CPS parsers have the same performance characteristics as GLL parsers with the new GSS [1]. Although CPS parsers and GLL use very different terms to describe their inner workings, presumably because of different communities they have been developed in, there are many similarities in these two approaches. Most notably, left recursion is handled in both approaches essentially in the same way.

In a GLL parser (with new GSS [1]) when the parser is before a nonterminal, a GSS node corresponding to the nonterminal and the current input position is searched. If the GSS node exists, the GSS edge is added, recording the current grammar position, and the previous parsing results associated with this GSS node are reused. When a new result is added to the results associated with a GSS node, the parser will explore the paths recorded on outgoing GSS edges with the new result. Similarly, in CPS parsers, if a memoized parser has been already called at the current input position, a continuation is added, recording the current position in the sequence, and the parsing results associated with the parser are reused. When a new result is added to the results of a parser, the recorded continuations will be called with the new result.

The main difference between a GLL parser and CPS parser is how the control flow is designed. GLL uses a GSS, which is a global data structure that encodes all parsing paths, while in a CPS parser, the control flow is encoded in continuation-passing style. It is in principle possible to realize a direct embedding of context-free grammars based on GLL parsing, although such an implementation may not be trivial. The GLL parsing algorithm [21] was designed for a code generation setting, in which a grammar processing phase generates the required labels for grammar positions. In parser combinators based on GLL, these labels should be dynamically represented. For example, Spiewak [23] shows how to build parser combinators based on GLL by encoding GSS nodes and edges as closures. This encoding resembles a form of continuation-passing style. Moreover, parser combinators based on GLL may benefit from a different GSS structure [1], as it is more similar to function memoization. We believe that our general parser combinators, compared to a functional formulation of GLL, are a more natural and elegant choice for realizing general parser combinators, as they offer a more straightforward generalization of traditional parser combinators.

6. Conclusions

In this paper we presented a foundation for general parser combinators based on an extension of Johnson's CPS recognizers. Jonhson stated that for constructing a parse forest from his approach *"a*

straightforward implementation attempt would probably be very complicated" [13]. To the best of our knowledge no parser version of Johnson's CPS recognizers existed before our work. One of our core contributions is the extension of CPS recognizers to parsers that construct binarized SPPFs [21, 22]. We showed that binarized SPPFs are a perfect fit for CPS recognizers. In particular, intermediate nodes provide a natural way to build a node from a binary sequence combinator. The results of parsing Java show that CPS parsers are practical for large, real-world grammars, even in a dynamic parser combinator setting where static grammar analysis is not an option. As future work, we plan to experiment with more programming languages and explore optimization opportunities in the Meerkat library.

Acknowledgments

We thank Jurgen Vinju, Paul Klint, and the anonymous reviewers for their constructive feedback on earlier versions of this paper.

References

[1] A. Afroozeh and A. Izmaylova. Faster, Practical GLL Parsing. In *Compiler Construction*, CC'15, pages 89–108. Springer, 2015.

[2] A. Afroozeh and A. Izmaylova. One Parser to Rule Them All. In *Onward! 15*, pages 151–170. ACM, 2015.

[3] K. Clarke. The Top-down Parsing of Expressions. Technical report, Dept. of Computer Science and Statistics, Queen Mary College, 1986.

[4] J. Earley. An Efficient Context-free Parsing Algorithm. *Commun. ACM*, 13(2):94–102, Feb. 1970. ISSN 0001-0782.

[5] B. Ford. Packrat Parsing: Simple, Powerful, Lazy, Linear Time, functional pearl. In *International conference on Functional programming*, ICFP '02, pages 36–47, 2002.

[6] B. Ford. Parsing Expression Grammars: A Recognition-based Syntactic Foundation. In *Symposium on Principles of Programming Languages*, POPL '04, pages 111–122, 2004.

[7] R. A. Frost, R. Hafiz, and P. Callaghan. Parser Combinators For Ambiguous Left-recursive Grammars. In *Practical Aspects of Declarative Languages*, PADL'08, 2008.

[8] J. Gosling, B. Joy, G. Steele, G. Bracha, and A. Buckley. The Java Language Specification Java SE 7 Edition, February 2013.

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

[10] G. Hutton and E. Meijer. Monadic Parsing in Haskell. *J. Funct. Program.*, 8(4):437–444, July 1998.

[11] T. Jim, Y. Mandelbaum, and D. Walker. Semantics and Algorithms for Data-dependent Grammars. In *Principles of Programming Languages*, POPL '10, pages 417–430, 2010.

[12] M. Johnson. The Computational Complexity of GLR Parsing. In *Generalized LR Parsing*, pages 35–42. Springer US, 1991.

[13] M. Johnson. Memoization in Top-down Parsing. *Comput. Linguist.*, 21 (3):405–417, Sept. 1995. ISSN 0891-2017.

[14] D. Leijen. Parsec, A Fast Combinator Parser. Technical Report 35, Department of Computer Science, University of Utrecht (RUU), 2001.

[15] S. McPeak and G. C. Necula. Elkhound: A Fast, Practical GLR Parser Generator. In *Compiler Construction*, CC'15, pages 73–88, 2004.

[16] A. Moors, F. Piessens, and M. Odersky. Parser Combinators in Scala. Technical report, Katholieke Universiteit Leuven, 2008.

[17] M.-J. Nederhof. A New Top-down Parsing Algorithm for Left-recursive DCGs. In *Progamming Language Implementation and Logic Programming*, pages 108–122. Springer, 1993.

[18] P. Norvig. Techniques for Automatic Memoization with Applications to Context-free Parsing. *Computational Linguistics*, 17(1):91–98, 1991.

[19] T. Parr, S. Harwell, and K. Fisher. Adaptive LL(*) Parsing: The Power of Dynamic Analysis. In *OOPSLA '14*, pages 579–598, 2014.

[20] T. Ridge. Simple, Efficient, Sound and Complete Combinator Parsing for All Context-Free Grammars, Using an Oracle. In *Software Language Engineering*, pages 261–281. Springer, 2014. .

[21] E. Scott and A. Johnstone. GLL Parse-tree Generation. *Science of Computer Programming*, 78(10):1828–1844, 2013.

[22] E. Scott, A. Johnstone, and R. Economopoulos. BRNGLR: A Cubic Tomita-style GLR Parsing Algorithm. *Acta informatica*, 44(6):427–461, 2007.

[23] D. Spiewak. Generalized Parser Combinators. `http://www.cs.uwm.edu/~dspiewak/papers/generalized-parser-combinators.pdf`, March 2010.

[24] M. Tomita. *Efficient Parsing for Natural Language: A Fast Algorithm for Practical Systems*. Kluwer Academic Publishers, USA, 1985.

[25] L. Tratt. Direct Left-recursive Parsing Expression Grammars. Technical Report EIS-10-01, Middlesex University, Oct. 2010.

[26] M. van den Brand, J. Heering, P. Klint, and P. Olivier. Compiling Rewrite Systems: The ASF+SDF Compiler. *ACM Transactions on Programming Languages and Systems*, 24(4):334–368, 2002.

[27] P. Wadler. Comprehending Monads. In *LISP and Functional Programming*, LFP '90, pages 61–78. ACM, 1990.

[28] A. Warth, J. R. Douglass, and T. Millstein. Packrat Parsers Can Support Left Recursion. In *Partial Evaluation and Semantics-based Program Manipulation*, PEPM '08, pages 103–110, 2008.

A. Complexity of CPS Recognizers

A.1 Notation

We start by introducing notation to support reasoning about the execution and complexity of memoized CPS recognizers. We denote the resulting recognizers of the terminal, epsilon, seq and rule combinators of Figure 2 and 6 as follows:

- $f_\varepsilon = \text{epsilon}()$ and $f_a = \text{terminal}(\text{"}a\text{"})$ denote recognizers for ε and for terminal a, respectively.
- $f_A = \text{rule}(f_{\alpha_1}, f_{\alpha_2}, \ldots, f_{\alpha_k})$ denotes a recognizer for nonterminal $A ::= \alpha_1 \mid \alpha_2 \mid \ldots \mid \alpha_k$, where $\alpha_i = x_1 x_2 \ldots x_m$ is an alternative consisting of a sequence of symbols.
- f_{α_i} is either a recognizer for a symbol, $f_{\alpha_i} = f_{x_1}$, $m = 1$, or a sequence of symbols, $f_{\alpha_i} = \text{seq}(f_{x_1}, f_{x_2}, \ldots, f_{x_m})$, $m \geq 2$.
- $f_{\alpha_1 \mid \alpha_2 \mid \ldots \mid \alpha_k}$ denotes the result of applying reduce in the body of the rule combinator, i.e., an unmemoized recognizer for A.

We also use the following notation:

(f_A, i, κ) denotes a call to f_A at an input position i with a continuation κ passed to the result of calling f_A at i.

$(R^{(A,i)}, K^{(A,i)})$ denotes the memo entry created upon the first call to f_A at the input position i, where $R^{(A,i)}$ is the result list, and $K^{(A,i)}$ is the continuation list.

For any call (f_A, i, κ), such that f_A is called at the input position i for the first time, continuations of the following forms are created:

κ_A^i denotes the continuation (Figure 5, lines 12–15) that maintains the results produced by f_A at i: it records new input positions in $R^{(A,i)}$ and runs all the continuations pending for this call in $K^{(A,i)}$.

$\kappa^i_{A ::= \alpha \cdot x \beta}$ denotes the continuation that corresponds to a grammar position $A ::= \alpha \cdot x\beta$ in an alternative of A, $|\alpha| \geq 1$, and is uniquely identified by $A ::= \alpha \cdot x\beta$ and i. When f_A is called at the input position i for the first time, i.e., when the unmemoized recognizer for A is called at i with κ_A^i (Figure 5, line 16), continuations of this form are recursively created as follows, where function application binds stronger than $\rightarrow$:
$\kappa^i_{A ::= \alpha \cdot x \beta} = j \rightarrow f_x(j)(\kappa^i_{A ::= \alpha x \cdot \beta})$, where f_x is the recognizer for the next symbol x in the alternative, and $\kappa^i_{A ::= \alpha x \cdot \beta}$ corresponds

to the next grammar position $A ::= \alpha x \cdot \beta$. This follows from the definition of seq, namely, the left reduce semantics and composition via `flatMap`. Note that for recognizers, $\kappa^i_{A::=\alpha x\beta\cdot}$ refers to the same continuation as κ^i_A.

A.2 Execution of Memoized CPS Recognizers

In Figure 5 we use data structures that are sufficient to explain the semantics but require amortized constant or linear time for their operations. The complexity analysis of the next sections, however, assumes that certain operations execute in constant time during the execution of memoized CPS recognizers. Therefore, we need to discuss how to provide such constant-time operations.

We assume that the following operations execute in constant time: copying arguments into the stack when the function call is executed, assigning the value of a variable to another variable, and creating closures. Scala is a JVM-based language that handles primitive types by value and reference types by reference value. Reference values are fixed-size values representing addresses in memory. Therefore, passing arguments of reference types to a function, or assigning one variable to another, results in copying the reference values, which executes in constant time. To implement closures, Scala uses closure conversion, such that captured variables are turned into fields of anonymous classes, representing closures, and these fields are initialized by passing extra arguments to the constructors. Therefore, we assume that operations such as `success`, `failure`, `map`, `flatMap` and `orElse` in Figure 4 execute in constant time.

Now, we consider the execution of memoized CPS recognizers that performs one of the following calls at each step:

$(f_\varepsilon, i, \kappa)$, a call to the recognizer for ε. The execution continues with the call (κ, i), and the recognizer call returns after the continuation call returns.

(f_a, i, κ), a call to the recognizer for a terminal a. If the terminal matches a substring in the input string starting from i (a constant-time operation), the execution continues with the call (κ, j), where j is the next input position after the match, otherwise no continuation is called, and the recognizer call returns.

(f_A, i, κ), a call to the recognizer for a nonterminal A. This call requires memo-table lookup of the result of calling f_A at i. If the result is not found, i.e., the first call to f_A at position i, a new function is created with two variables in its scope: the result list $R^{(A,i)}$ and continuation list $K^{(A,i)}$. As the memo table can be implemented as an array of length $n+1$, n is the length of the input, the lookup operation can execute in constant time. In addition, ast the continuation list can be implemented as a linked list and the result list as an array of size $n+1$, addition of a new element into the lists (`+=`) and element lookup into the result list (`contains`) can execute in constant time. The execution of (f_A, i, κ) continues with addition of κ to $K^{(A,i)}$ (if- and else-branch of `memo_result`). Depending on the check whether $K^{(A,i)}$ is empty (a constant-time operation), the execution continues with either the call to the unmemoized recognizer for A, $(f_{\alpha_1|\alpha_2|\ldots|\alpha_k}, i, \kappa^i_A)$, or iteration over $R^{(A,i)}$ (a linear operation) calling κ for each recorded input position. The call $(f_{\alpha_1|\alpha_2|\ldots|\alpha_k}, i, \kappa^i_A)$ will perform a constant number of steps to combine the results of calling f_{α_i} at i (via `orElse`), and will eventually lead to a constant number of calls $(f_{\alpha_1}, i, \kappa^i_A), \ldots, (f_{\alpha_k}, i, \kappa^i_A)$, corresponding to the alternatives of A.

$(f_\alpha, i, \kappa^i_A)$, a call to the recognizer for an alternative of A. If $|\alpha| > 1$, this call results in a constant number of composition steps (via `flatMap`). Then, the continuations corresponding to the grammar positions in α are created, and the call $(f_x, i, \kappa^i_{A::=x\cdot\gamma})$, assuming $\alpha = x\gamma$, to the recognizer for the first symbol in α is made.

$(\kappa^i_{A::=\alpha\cdot x\beta}, j)$ or (κ^i_A, j), a call to a continuation. The former call directly results in $(f_x, j, \kappa^i_{A::=\alpha x\cdot\beta})$, a call to the recognizer for x with the next continuation $\kappa^i_{A::=\alpha x\cdot\beta}$. The latter call leads to the check (constant-time) whether j exists in the result list, and if not, j is added (constant-time, as discussed above) to the result list. Finally, iteration over the continuation list $K^{(A,i)}$ (a linear operation) runs each continuation with the new input position j.

Consider calls of the forms (f_x, i, κ), where x is any symbol, and (κ^i_A, j). The execution of memoized CPS recognizers continues linearly until either a call (f_A, i, κ), to the recognizer for a nonterminal, or (κ^i_A, j) is executed: both calls may result in iteration over a list, the size of which depends on n, calling a continuation in each iteration step. When a call (f_A, i, κ) is the first call to f_A at the input position i, it does not lead to iteration but requires an $O(n)$ operation to create an array of size $n+1$ for the result list.

A.3 Complexity of Memoized CPS Recognizers

In this section we show that the execution of memoized CPS recognizers can require $O(n^{m+1})$ operations, where m is the length of the longest rule in the grammar. This unbounded polynomial behavior can be observed by the family of highly ambiguous grammars [12]: $S ::= S^m | SS | b | \varepsilon$, $m \geq 3$, where S^m denotes a sequence of S's of length m. Because of the last three alternatives, any, possibly empty, sequence of b's can be recognized by S.

We consider parsing string b^n. The memoization technique of Section 2.3 ensures that for any input position i, there will be at most one call (f_{S^m}, i, κ^i_S), corresponding to the first alternative of S, made upon the first call to f_S at i. Now, we show that the total number of calls that will be made to the continuations κ^i_S, $0 \leq i \leq n$, is $O(n^{m+1})$.

After creating the continuations corresponding to the positions in the sequence S^m and the input position i, the call $(f_S, i, \kappa^i_{S::=S\cdot SS^{m-2}})$ to the recognizer for the first symbol in the alternative S^m will be made. First, we consider the call $(f_S, 0, \kappa^0_{S::=S\cdot SS^{m-2}})$ corresponding to input position 0.

This call results in addition of the continuation $\kappa^0_{S::=S\cdot SS^{m-2}}$ to $K^{(S,0)}$, and therefore, for each i_1 in $R^{(S,0)}$ a call $(\kappa^0_{S::=S\cdot SS^{m-2}}, i_1)$ will be executed. Each of these calls in turn results in a call to the recognizer for the second S in S^m, $(f_S, i_1, \kappa^0_{S::=SS\cdot S^{m-2}})$, which adds the continuation $\kappa^0_{S::=SS\cdot S^{m-2}}$ to $K^{(S,i_1)}$. Again, for each i_2 in $R^{(S,i_1)}$, $i_1 \leq i_2$, a call $(\kappa^0_{S::=SS\cdot S^{m-2}}, i_2)$ will be executed.

Since $0 \leq i_1 \leq i_2 \leq n$, the total number of calls made to $\kappa^0_{S::=SS\cdot S^{m-2}}$ is $\binom{n+2}{2}$. Some of these calls, however, are duplicate: for each i_2, there will be multiple i_1, $0 \leq i_1 \leq i_2$, such that $\kappa^0_{S::=SS\cdot S^{m-2}} \in K^{(S,i_1)}$ and $i_2 \in R^{(S,i_1)}$. Each duplicate call $(\kappa^0_{S::=SS\cdot S^{m-2}}, i_2)$ leads to addition of the continuation $\kappa^0_{S::=SSS\cdot S^{m-2}}$ to $K^{(S,i_2)}$, resulting in the same continuation being added to the same continuation list multiple times. As a result, the total number of calls made to κ^0_S, corresponding to the end of the alternative S^m, is equal to the number of all the combinations with repetition for the respective indices $0 \leq i_1 \leq i_2 \leq \ldots \leq i_{m-1} \leq i_m \leq n$, which is $\binom{n+m}{m}$.

Finally, if we consider all the calls $(f_S, i, \kappa^i_{S::=S\cdot SS^{m-2}})$, $0 \leq i \leq n$, the total number of calls made to the continuations κ^i_S, resulting from the first alternative, is $\binom{n+m+1}{m+1}$, which is a polynomial in n of order $m+1$.

B. Complexity of CPS Parsers

In this section we show that the complexity of CPS parsers is $O(n^3)$, where n is the length of the input. We start by adapting the notation of Section A.1 to the parser version. First, f_a, f_ε, f_A, etc. now denote

the respective parsers instead of recognizers. Second, result list $R^{(A,i)}$ stores nonterminal nodes (A,i,j), instead of input positions. Finally, for any call (f_A,i,κ), such that the parser f_A is called at the input position i for the first time, continuations of the following forms can be created:

κ^i_A denotes the continuation that maintains the results produced by f_A at i, now, recording new nonterminal nodes (A,i,j) in $R^{(A,i)}$. In addition, $\kappa^i_{A::=\alpha\cdot}$ now denotes the continuation that corresponds to a grammar position $A ::= \alpha\cdot$ and is created as:
$\kappa^i_{A::=\alpha\cdot} = t \rightarrow \kappa^i_A(h^i_{A::=\alpha}(t))$, where $h^i_{A::=\alpha}$ is a function (passed to `map` in `rule1` of Figure 10) which is uniquely identified by i and $A ::= \alpha$. $h^i_{A::=\alpha}$ takes a non-packed node t and creates a nonterminal node (A,i,j). If $|\alpha| > 1$, t is an intermediate node of the form $(A ::= \alpha\cdot, i, j)$, otherwise a symbol node of the form (x,i,j) corresponding to x, the only symbol in α.

$\kappa^i_{A::=\alpha\cdot x\beta}$ denotes the continuation that corresponds to a grammar position $A ::= \alpha\cdot x\beta$ in an alternative of A, $|\alpha| \geq 1$, and is uniquely identified by $A ::= \alpha\cdot x\beta$ and i. Upon the first call to f_A at i, continuations of this form are recursively created as:
$\kappa^i_{A::=\alpha\cdot x\beta} = t \rightarrow g^i_{A::=\alpha\cdot x\beta}(t)(\kappa^i_{A::=\alpha x\cdot\beta})$, where t is either an intermediate node of the form $(A ::= \alpha\cdot x\beta, i, j)$, $|\alpha| > 1$ or a symbol node of the form (y,i,j), $|\alpha| = 1$.
$g^i_{A::=\alpha\cdot x\beta}$ is a function (passed to `flatMap` in `seq2` of Figure 10), which is uniquely identified by i and $A ::= \alpha\cdot x\beta$. This function calls the parser for the next symbol f_x at the right extent of t, say j. Then, it creates the following continuation and passes this continuation to the result of $f_x(j)$:
$\kappa^{i,j}_{A::=\alpha x\cdot\beta} = t \rightarrow \kappa^i_{A::=\alpha x\cdot\beta}(h^{i,j}_{A::=\alpha x\cdot\beta}(t))$, where t is a symbol node of the form (x,j,k).
$h^{i,j}_{A::=\alpha x\cdot\beta}$ is the function (passed to `map` in `seq2` of Figure 10) created by $g^i_{A::=\alpha\cdot x\beta}$ to construct an intermediate node $(A ::= \alpha x\cdot\beta, i, k)$ given (x,j,k). Finally, $\kappa^{i,j}_{A::=\alpha x\cdot\beta}$ calls $\kappa^i_{A::=\alpha x\cdot\beta}$ with the resulting intermediate node.

LEMMA 1. For any parser f_B and any input position j, $0 \leq j \leq n$ the number of continuations in the continuation list $K^{(B,j)}$ is $O(n)$.

PROOF 1. For any nonterminal A such that $A ::= \alpha B\beta$ is an alternative of A, we have only the following calls that add a continuation to $K^{(B,j)}$:

1. $(f_B, j, \kappa^i_{A::=B\cdot\beta})$ when $|\alpha| = 0$ and $i = j$. This call can only result from $(f_{A::=B\beta}, j, \kappa^i_{A::=B\beta\cdot})$.
2. $(f_B, j, \kappa^{i,j}_{A::=\alpha B\cdot\beta})$ when $|\alpha| \geq 1$ and $0 \leq i \leq j$. This call can only result from $(\kappa^i_{A::=\alpha\cdot B\beta}, t)$, where t is an intermediate node $(A ::= \alpha\cdot B\beta, i, j)$, $|\alpha| > 1$, or a symbol node (y,i,j), $|\alpha| = 1$.

In the first case, given that CPS parsers for nonterminals are memoized, there will be at most one call to the parser for an alternative at each input position, such as $(f_{A::=B\beta}, j, \kappa^i_{A::=B\beta\cdot})$, and therefore, at most one call to the parser for the first symbol in the alternative, such as $(f_B, j, \kappa^i_{A::=B\cdot\beta})$. Thus the continuation $\kappa^i_{A::=B\cdot\beta}$ can be added to $K^{(B,j)}$ at most once. In the second case, given that continuations of the form $\kappa^i_{A::=\alpha\cdot B\beta}$ are memoized, for any i, $0 \leq i \leq j$, there will be at most one call $(\kappa^i_{A::=\alpha\cdot B\beta}, t)$ resulting in creation of $\kappa^{i,j}_{A::=\alpha B\cdot\beta}$ and the call $(f_B, j, \kappa^{i,j}_{A::=\alpha B\cdot\beta})$. Thus the continuation $\kappa^{i,j}_{A::=\alpha B\cdot\beta}$ is uniquely identified by i, j and $A ::= \alpha B\cdot\beta$ and can be added to $K^{(B,j)}$ at most once. Finally, given that $0 \leq i \leq n$, the total number of continuations in $K^{(B,j)}$ is at most $O(n)$. □

LEMMA 2. For any parser f_A and any input position i, $0 \leq i \leq n$, the number of elements in $R^{(A,i)}$ is $O(n)$.

PROOF 2. Given that continuations of the form $\kappa^i_{A:=\alpha\cdot}$ and κ^i_A are memoized, for any non-packed node t with left extent i and right extent j, $i \leq j \leq n$, there will be at most one call $(\kappa^i_{A:=\alpha\cdot}, t)$ resulting in $(\kappa^i_A, (A,i,j))$, and there will be at most one call $(\kappa^i_A, (A,i,j))$ adding (A,i,j) to $R^{(A,i)}$. Thus the number of elements in $R^{(A,i)}$ is at most $O(n)$. □

THEOREM 1. The complexity of CPS parsers that construct a binarized SPPF is $O(n^3)$.

PROOF 3. We consider calls of the forms:

1. $(f_x, j, \kappa^i_{A::=x\cdot\beta})$ and $(f_x, j, \kappa^{i,j}_{A::=\alpha x\cdot\beta})$, $|\alpha| \geq 1$
2. $(\kappa^i_{A::=\alpha\cdot x\beta}, t)$
3. $(\kappa^i_A, (A,i,j))$

where $0 \leq i \leq j \leq n$, and t is a non-packed node with left extent i and right extent j. The execution of CPS parsers continues linearly until either a call to the parser for a nonterminal, of the form $(f_B, j, \kappa^i_{A::=B\cdot\beta})$ and $(f_B, j, \kappa^{i,j}_{A::=\alpha B\cdot\beta})$, or a call of form 2 or 3 is executed. In the parser version, a call of form 2 may require an $O(n)$ operation to create a continuation $\kappa^{i,j}_{A::=\alpha x\cdot\beta}$[5]. We show that there will be at most $O(n^2)$ calls of form 1. Also, there will be at most $O(n^2)$ calls of form 2, each of which creates a continuation $\kappa^{i,j}_{A::=\alpha x\cdot\beta}$, and at most $O(n^2)$ calls of form 3, each of which results in iteration over a continuation list.

Similar to the proof of Lemma 1, there are only two forms of calls that may lead to a call of form 1. Given that CPS parsers for nonterminals are memoized, there will be at most $O(n)$ calls of the form $(f_{A::=x\beta}, j, \kappa^i_{A::=x\beta\cdot})$ resulting in $(f_x, j, \kappa^i_{A::=x\cdot\beta})$, where $0 \leq i = j \leq n$. Also, given that continuations of the form $\kappa^i_{A::=\alpha\cdot x\beta}$ are memoized, there will be at most $O(n^2)$ calls of form 2 creating $\kappa^{i,j}_{A::=\alpha x\cdot\beta}$ and resulting in $(f_x, j, \kappa^{i,j}_{A::=\alpha x\cdot\beta})$, where $0 \leq i \leq j \leq n$. Thus there will be at most $O(n^2)$ calls of form 1. Given that continuations of the form κ^i_A are memoized, there will be also at most $O(n^2)$ calls of form 3, $0 \leq i \leq j \leq n$, resulting in iteration.

Each call of the form $(f_B, j, \kappa^i_{A::=B\cdot\beta})$ or $(f_B, j, \kappa^{i,j}_{A::=\alpha B\cdot\beta})$, when it is the first call to f_B at the input position j, or each call of form 2 may result in a constant number of $O(n)$ operations to create arrays of size $n+1$ (to initialize a result list and/or to create memoized continuations) followed by a constant number of other calls of form 1 (already subsumed by the $O(n^2)$ calls above). Each call of the form $(f_B, j, \kappa^i_{A::=B\cdot\beta})$ or $(f_B, j, \kappa^{i,j}_{A::=\alpha B\cdot\beta})$, when it is not the first call to f_B at the input position j, or each call of form 3 may result in iteration over a list leading to at most $O(n)$ other continuation calls (by Lemma 1 and 2). Each of the $O(n)$ other continuation calls is either a duplicate call eliminated by the memoization, or a call to a continuation of the form $\kappa^i_{A::=\alpha\cdot x\beta}$ (already subsumed by the $O(n^2)$ continuation calls above), or of the forms $\kappa^{i,j}_{A::=\alpha x\cdot\beta}$ and $\kappa^i_{A=\alpha\cdot}$ directly resulting in a call already subsumed by the $O(n^2)$ calls above. Thus the complexity of CPS parsers that construct binarized SPPFs is at most $O(n^3)$. □

[5] Although in the complexity analysis we assume that all continuations are memoized, it can be shown that memoizing continuations of this form is not needed. We use this as an optimization in the Meerkat library.