Generalized Parser Combinators

Daniel Spiewak

March 28, 2010

Abstract

Parser combinators are a well-known technique for constructing recursive-descent parsers through composition of other, less-complex parsers. In fact, parser combinators are often held up as an example of the elegance and composability inherent to functional programming. However, like any application of recursive-descent, parser combinators fail to support grammars with left-recursion or most kinds of ambiguity. They also suffer from exponential runtimes in the worst case. We present a framework which allows the construction of parsers for *any* context-free grammar (including those with left-recusion or ambiguity), $O(n^3)$ performance in the worst-case and $O(n)$ for most grammars, all while still preserving the functional composability and natural control flow offered by traditional parser combinators. This framework also serves as the first implementation of the GLL algorithm in a functional style.

Parser combinators were first proposed by Philip Wadler [12] in the late '80s as a means of defining composable recursive-descent parsers in a functional style. The concept is based upon the observation that recursive-descent parsers can actually be seen in terms of monads. At a fundamental level, any recursive-descent parser may be viewed as a composition of the following rules:

$a\|b$	Either a or b
$a\ b$	First a, then b
`x`	The terminal '`x`'
ϵ	The empty string

Table 1: Atomic parser operations

Each of these rules may be defined in terms of the standard monadic operations. The alternation rule $a|b$ would be $a\ orelse\ b$; the sequential rule $a\ b$ would be $a \ggg= (\lambda x\ .\ b)$; a terminal `x` would be $lift$(`x`) (`return x` in Haskell syntax); and ϵ could be defined as either a special value, *success*, or the empty terminal (`''`). By encoding parsers in terms of these rules, Wadler was able to develop a framework for representing any LL(*) grammar through the use of an embedded DSL. This framework eventually became the basis for more sophisticated implementations such as Haskell's Parsec [2] and Scala's modular parser combinators [3].

While parser combinators are an extremely elegant way of encoding recursive-descent parsers, they also share all of the disadvantages of that parsing technique. LL(k) (for $k > 1$) alternates are handled correctly, but only through the use of infinite backtracking. This means that certain grammars can lead to exponential runtimes on invalid input. Additionally, parser combinators share the Achilles heel of top-down parsers in that they cannot handle left-recursive grammars (with the exception of some very specific edge-cases).

Further, parser combinators — and recursive-descent in general — cannot handle most forms of ambiguity in their input grammar. The "most forms" qualifier refers to the fact that certain local ambiguities can be avoided through careful ordering of alternates in the parse process. As an example, the classic "dangling `else`" ambiguity found in many languages in the Pascal family can be resolved in a recursive-descent parser by declaring the `if/else` clause before the shortened `if` in the rule for matching expressions. It is sometimes possible to resolve operator precedence ambiguities using the same technique. However, in general, languages with ambiguous grammars cannot be parsed using parser combinators.

This shortcoming dramatically reduces the usefulness of parser combinators. Natural language processing unequivically depends upon parsers which can handle ambiguity. Certain biochemical analyses have the same dependence. Even languages which are not inherently ambiguous will often benefit from controlled local ambiguities in their grammar. Such ambiguities often serve to make the grammar cleaner and the parser more robust.

Despite all of these limitations, parser combinators have been widely adopted in functional languages as the de facto method for constructing parsers. This is primarily due to their inherent composability and highly-intuitive control flow. It is usually fairly easy to debug a parser composed using parser combinators. The parsers themselves are also quite flexible, easily satisfying use-cases such as interactive shells (where language fragments must be parsed in isolation) and asynchronous parsing, areas to which more powerful techniques such as LALR are generally ill-suited. In short, parser combinators are favored despite their shortcomings purely because of their ease of use.

We present a framework of combinators for the construction of parsers which can handle *any* context-free grammar in polynomial time. The resulting parsers are composed of smaller atomic parsers, allowing for all of the flexibility of traditional parser combinators. Perhaps more importantly, the control flow of the parser follows the structure of the grammar, just as in recursive-descent. This means that the "intuitive aspect" of traditional parser combinators remains intact. In fact, we make the claim that the combinators defined by our framework retain all of the advantages of traditional parser combinators while adding support for useful features such as left-recursion and ambiguity.

This is accomplished through an adaptation of GLL [7], a new parsing algorithm proposed by Elizabeth Scott and Adrian Johnstone. We give an overview of the algorithm in its unmodified form in Section 2. Section 1 shows the development of a traditional parser combinator framework, and Section 3 explores our modifications to GLL, both those necessary for combinatorial parser construction as well as the modifications which would be required by any functional implementation of the algorithm. In Section 4, we examine the performance implications of GLL in more detail, offering proof sketches of its asymptotic complexity along with some emperical results on a highly-ambiguous grammar.

All of our examples (and the GLL combinators framework itself) are written in Scala [5, 4]. This language was chosen as some of its unique features (including a hybrid object-oriented/functional design) were extremely helpful in the implementation of the framework. Generally speaking, the modifications to GLL we describe could be applied to any functional programming language (including pure-functional languages like Haskell).

1 Traditional Parser Combinators

In this section we implement a cut-down parser combinator framework using the conventional approach. Input is received as an instance of `Stream[Char]`, or a lazy-list of character values (similar to Haskell's `String`). There is no support for regular expressions, line counting or error recovery — though, these features *could* be added without any significant alterations. The idea is to present the fundamentals of parser combinators in a concrete fashion.

The first step is to codify the overall design of our framework. Every parser is an instance of type `Parser[A]`, where `A` is the type of value returned by the parser upon success. Every parser is also a function from `Stream[Char]` to `Result[A]`, which is an algebraic data-type with two constructors: `Success[A]` and `Failure`. Instances of `Success[A]` represent a successful parse and contain the resulting value as well as the tail of the stream, the tokens which have yet to be consumed. Instances of `Failure` represent an unsuccessful parse and contain only an error message detailing the failure.

There are three different implementations of `Parser[A]`: `DisjunctiveParser[A]`, `SequentialParser[A, B]` and `LiteralParser`. Each of these represents an atomic recursive-descent parser operations given in Table 1. The ϵ parser will be encoded as a `LiteralParser` for the empty string. The code for these classes is given in Listing 1.

It is insufficient to just define the parser implementations, we must also provide some mechanism by which one parser can be combined with another, producing either a disjunctive or sequential parser in return. We

```
case class ~[+A, +B](a: A, b: B)

sealed trait Result[+A]

case class Success[+A](value: A, tail: Stream[Char]) extends Result[A]
case class Failure(msg: String) extends Result[Nothing]

trait Parser[+A] extends (Stream[Char] => Result[A]) {
  // more to come...
}

class DisjunctiveParser[+A](left: Parser[A], right: Parser[A]) extends Parser[A] {
  def apply(in: Stream[Char]) = left(in) match {
    case s: Success[A] => s
    case _: Failure => right(in)
  }
}

class SequentialParser[+A, +B](left: Parser[A], right: =>Parser[B]) extends Parser[~[A, B]] {
  def apply(in: Stream[Char]) = left(in) match {
    case Success(res1, tail) => right(tail) match {
      case Success(res2, tail) => Success(new ~(res1, res2), tail)
      case f: Failure => f
    }

    case f: Failure => f
  }
}

class LiteralParser(value: String) extends Parser[String] {
  def apply(in: Stream[Char]) = {
    if (in.lengthCompare(value.length) < 0) {
      Failure("Unexpected end of stream")
    } else {
      val trunc = in take value.length

      if (trunc.zipWithIndex forall { case (c, i) => value(i) == c })
        Success(value, in drop value.length)
      else
        Failure("Expected '" + value + "' got '" + trunc.mkString + "'")
    }
  }
}
```

Listing 1: Fundamental parser combinators

could just force users to encode their grammars by explicitly creating new instances of the three core parsers, but that would be ugly. Instead, we will add two operators to `Parser[A]` and an implicit conversion on type `String` (to easily create instances of `LiteralParser`). Scala's built-in parser combinators use the '`~`' operator to signify sequential composition and the '`|`' operator for disjunctions. For the sake of clarity, we will maintain that convention. The modified `Parser` trait and its companion object are given below:

```
trait Parser[+A] extends (Stream[Char] => Result[A]) {
  def |[B >: A](that: Parser[B]) = new DisjunctiveParser(this, that)

  def ~[B](that: =>Parser[B]) = new SequentialParser(this, that)
}

object Parser {
  implicit def literal(str: String) = new LiteralParser(str)
}
```

Even with this incredibly simple framework, it is already possible to build basic recognizers. For example, the following code defines a recognizer for a 0-1 arithmetic language with right-associativity:

```
import Parser._

def expr: Parser[Any] = (
    num ~ "+" ~ expr
  | num ~ "-" ~ expr
  | num
)

def num = "0" | "1"
```

Notice that this grammar is not LL(1) due to the fact that `num` is the first parser of each of the three rules in the `expr` disjunction. We could very easily factor this grammar so that each production would have a distinct PREDICT set, but that is unnecessary. Even this nearly-trivial framework is able to handle this grammar by blindly stepping into each production in turn. If the first production it tries returns `Failure`, it moves on to the next one. This repeated backtracking continues until every branch of every possible disjunction has been exhausted. If each parser in the disjunction fails, it will return `Failure`.

The real utility of parser combinators comes to the fore when they are being used to produce values of greater sophistication than simply `Success` or `Failure`. The `expr` grammar given above does a fine job of determining whether or not a given input matches the grammar, but it doesn't provide anything else of use. This deficiency is easily remedied by adding a `^^` operator to the `Parser` trait. This operator will take a function as a parameter and return a new `Parser[B]` corresponding to the result of mapping the given function over the current parser's value:

```
trait Parser[+A] extends (Stream[Char] => Result[A]) { outer =>
  ...

  def ^^[B](f: A => B) = new Parser[B] {
    def apply(in: Stream[Char]) = outer(in) match {
      case Success(res, tail) => Success(f(res), tail)
      case f: Failure => f
    }
  }
}
```

We can now modify our expression grammar from earlier to compute the actual numeric result of the expression being parsed:

```
import Parser._

def expr: Parser[Int] = (
    num ~ "+" ~ expr  ^^ { case x ~ _ ~ y => x + y }
  | num ~ "-" ~ expr  ^^ { case x ~ _ ~ y => x - y }
  | num
)

def num = ("0" | "1") ^^ { _.toInt }
```

The syntax to invoke this parser against the input string "`0+1-1+1+1`" is as follows:

```
expr("0+1-1+1+1" toStream) match {
  case Success(result, _) => println(result)
  case Failure(msg) => println(msg)
}
```

As expected, this snippet prints "2" to the standard output stream when executed. It is worth noting that the semantic actions (the anonymous functions passed to the `^^` operator) are completely type-safe and statically checked. This feature alone begins to provide some insight into why parser combinators are often considered to be much easier to use than external parser generators.

It is worth noting that this implementation of parser combinators does not define either of the monadic functions $\gg=$ (`flatMap` in Scala) or *orelse*. This omission is merely for the sake of brevity; it would be quite simple to add these methods to the `Parser` trait, allowing parser instances to be composed using Scala's `for`-comprehensions rather than merely through the chaining of the `~` and `|` operators.

2 Overview of GLL

Originally, GLL started out as a form of RIGLR [6]. In the original paper, Scott and Johnstone make the observation that by applying a process they refer to as "terminalization" to all but the topmost instance of each non-terminal, the resulting RIGLR parser would have a control flow which would closely mirror that of a recursive-descent parser. This idea eventually grew into GLL.

2.1 Recursive-Descent

Despite its origins in the world of bottom-up parsers, GLL is probably easiest to understand by starting with recursive-descent, rather than the less-intuitive RIGLR. In this section, we develop a pseudo-code recursive-descent recognizer for a moderately ambiguous grammar. However, recursive-descent can only handle grammars which are at worst LL(*). We resolve this inadequacy by implementing two critical modifications, converting from vanilla recursive-descent to a full GLL recognizer. Consider the following:

$$S \quad ::= \quad a\, S \mid a \mid \epsilon$$

To create a recursive-descent parser for this grammar, we must define a function for each non-terminal in the grammar. Since our grammar only has one non-terminal (S), this task is fairly simple. Within this function, we define a conditional branch to determine the relevant production based on which values are seen in the stream and whether they match the FIRST set for a certain production. Note that this form of recursive-descent is actually only as powerful as LL(1). Its runtime is linear, but the class of grammars it can handle is much smaller than a parser with full backtracking (or infinite lookahead). In psuedo-code, our parser looks like this:

$I[i]$ is the input character stream at some index i
$fail(i)$ is a function which aborts the parse and reports an error for some index i

```
$parseS(i)$ :
  if $(I[i] = a)$ {
    $i \leftarrow parseNT(i, a)$
    $i \leftarrow parseS(i)$
    return $i$
  } else if $(I[i] = \$)$ {
    return $i$
  } else {
    $fail(i)$
  }
$parseNT(i, nt)$ :
  if $(I[i] = nt)$ {
    return $i + 1$
  } else {
    $fail(i)$
  }
```

This parser will successfully recognize any input which is valid according to the grammar we specified. However, it only accomplishes this by effectively ignoring the second production of the S non-terminal ("a"). In order to correctly handle this and other ambiguous grammars, we need to devise a way to follow all possible derivations for a given input.

2.2 Generalizing the Parser

This can be accomplished through the addition of a dispatch stack (which we call $\mathcal{R}$). At any point in the parse where ambiguities arise, we will use the stack to keep track of all possible "choices" arising from that ambiguity. We leverage this stack by using it to determine which "parse function" we should call and with what input. These modifications require some fairly serious restructuring of the recursive-descent algorithm.

The first thing we need to do is switch from parse functions to parse labels. We must also further subdivide our former parse functions by giving each alternate its own label. Thus, our grammar will have parse labels L_S, L_{S_0}, L_{S_1} and L_{S_2}. There is also an additional parse label $L_{S_0'}$ representing the "continuation" of the L_{S_0} production following the dispatch to the S non-terminal. The dispatch stack will store a list of descriptors of the form (L, s, i) where L is a parse label, s is a stack and i is an index into the input stream. The stack s is to keep track of the parse continuation. Whenever we reach the end of a production, we will pop this stack and push its head onto $\mathcal{R}$. In conventional recursive-descent, we can simply rely on the call stack for this task, but since we are using labels instead of functions, a separate stack must be maintained.

Of course, if we are pushing things onto the dispatch stack, we must logically have some place where we pop them back off and perform the appropriate dispatch. We denote this section by the label L_0. In essence, its task is to pop the head of $\mathcal{R}$ and then jump to the label given by the descriptor. If $\mathcal{R}$ is empty and our current index (denoted c_i) is equal to the length of the input stream I, then we report SUCCESS. This situation corresponds to when all possible parse derivations have been exhausted and we have successfully consumed the entire input stream. The other alternative is that $\mathcal{R}$ is empty but $c_i < |I|$. In this case, we have run out of possible derivations, but we have not consumed all of the available input. Thus, we must report FAILURE. If we apply these modifications to our earlier example, we arive at the following parser:

```
$I[i]$ is the input character stream at some index $i$
$c_i$ is the current index into input stream $I$
$\mathcal{R}$ is a stack of descriptors $(L, s, i)$

$parse()$ :
  $\mathcal{R}.push(L_S, [], 0)$
  $L_0$ :
    if $(\mathcal{R} \neq \emptyset)$ {
        $(L, s, c_i) \leftarrow \mathcal{R}.pop()$
        **goto** $L$
    } **else if** $(c_i = |I|)$ {
        **return** SUCCESS
    } **else** {
        **return** FAILURE
    }
$L_S$ :
    **if** $(I[c_i] = a)$ {
        $\mathcal{R}.push(L_{S_0}, s, c_i)$
        $\mathcal{R}.push(L_{S_1}, s, c_i)$
    } **else if** $(I[c_i] = \$)$ {
        $\mathcal{R}.push(L_{S_2}, s, c_i)$
    }
    **goto** $L_0$
$L_{S_0}$ :
    **if** $(I[c_i] = a)$ {
        $s \leftarrow [L_{S_{0'}}, s]$
        **goto** $L_S$
    }
    **goto** $L_0$
$L_{S_{0'}}$ :
    $[L, s] \leftarrow s$
    **goto** $L$

$L_{S_1}$ :
    **if** $(I[c_i] = a)$ {
        $[L, s] \leftarrow s$
        $\mathcal{R}.push(L, s, c_i + 1)$
    }
    **goto** $L_0$
$L_{S_2}$ :
    **if** $(I[c_i] = \$)$ {
        $[L, s] \leftarrow s$
        $\mathcal{R}.push(L, s, c_i)$
    }
    **goto** $L_0$
```

Intuitively, any time we arive at a disjunction in the grammar, we push all possible alternates onto the dispatch stack. The real beauty of this algorithm is the fact that for grammars which are less ambiguous, most disjunctions will only push a single descriptor. When we push but one descriptor, the algorithm degrades directly to predictive top-down parsing, which is $O(n)$ in the length of the input.

Even though we have eliminated our parse functions, this algorithm is still a variant of recursive-descent. The stack carried along with each descriptor takes the place of the call stack in the original implementation. Further, within each alternate (L_{S_0}, L_{S_1}, etc) the parse follows exactly the same steps as the original implementation. The dispatch stack ensures that the traversal remains depth-first, and so even the control flow can be traced as following the same route.

2.3 Trimming Exponentiality

Unfortunately, for grammars which are recursive and even slightly ambiguous, the algorithm as it stands will be much worse than linear time. In fact, the runtime would actually be *exponential* as an explosive number of descriptors would be pushed. It is possible that a single token might be consumed an incredible number of times in an attempt to exercise every parse path. Clearly, something must be done to reduce the number of descriptors being pushed onto the stack.

The underlying problem is that we are repeating a great deal of work. Grammars which exhibit recursive ambiguity often derive on a path which diverges at points of ambiguity, but which eventually recombines at a later point in the parse (indicating a local ambiguity in the grammar). With our current scheme, the control flow will never "recombine". Instead, duplicate descriptors will continue being pushed onto the dispatch stack until the end of the input stream is reached. We need some way of detecting the points where one derivation rejoins another and a technique for "pinching off" the duplicate path.

As it turns out, the best way of accomplishing this is a 20 year-old data structure invented by Masaru Tomita: the graph-structured stack [9], or "GSS". The GSS is a way of literally modeling ambiguity in such a way that common prefixes and suffixes of divergent paths are shared. We can leverage and extend this data structure slightly to avoid pushing duplicate descriptors.

The first thing we add is a set, U_i, which contains all labels which have already been pushed for index i. We must also add another set $\mathcal{P}$ which contains all pairs of GSS node and index (u, i) which have *already* been popped. This is to allow common suffixes to actually merge, rather than simply eliminating shared paths. When we implement GLL as a full parser (rather than simply a recognizer), we will also need to store derived "reduce values" in a related set.

The GSS nodes themselves are of a standard design. Each node u contains a descriptor (L, i) and pointers to its parent(s). A *pop*() action on such a node will return the set of all parents of node u. Thus, when we pop back to a continuation descriptor in the algorithm, we may actually push more than one descriptor onto $\mathcal{R}$. Conceptually, the GSS may be seen as a directed acyclic graph where only the leaf nodes may be seen at any point. A *pop*() action is as simple as removing the relevant leaf node, marking its parent(s) as leaves and new "heads" of the stack.

Very few changes are required to implement these conceptual modifications in our running example. The primary difference is that we will always check U_i for the prior existence of some descriptor before we add it to $\mathcal{R}$. This action will be implemented by the *add*(...) function. We must add popped descriptors to $\mathcal{P}$, so we define another function, *pop*(...), to handle this task. Finally, we define a third function, *create*(...), which generates a new GSS node u along with the appropriate edges.

```
$I[i]$ is the input character stream at some index $i$
$c_i$ is the current index into input stream $I$
$\mathcal{R}$ is a stack of descriptors $(L, u, i)$
$U_i$ is the set of labels $L$ for index $i$ which have been added to $\mathcal{R}$
$\mathcal{P}$ is the set of popped descriptors $(u, i)$
$\mathcal{G}$ is the set of all GSS nodes $(L, i)$
$E_u$ is the set of GSS nodes adjacent to node $u$
$c_u$ is the current GSS node

$create(L, u, i)$ :
  **if** $((L, i) \notin \mathcal{G})$ {
    let $v$ be a new GSS node $(L, i)$
    add $v$ to $\mathcal{G}$
  } **else** {
    let $v$ be the GSS node $(L, i)$
  }
  **if** $(u \notin E_v)$ {
    add $u$ to $E_v$
    **for each** $((v, j) \in \mathcal{P})$ {
        $add(L, u, j)$
      }
    }
    **return** $v$
$add(L, u, i)$ :
    **if** $((L, u) \notin U_i)$ {
      add $(L, u, i)$ to $\mathcal{R}$
      add $(L, u)$ to $U_i$
    }
$pop(u, i)$ :
    $(L, \_) \leftarrow u$
    add $(u, i)$ to $\mathcal{P}$
    **for each** $(v \in E_u)$ {
      $add(L, v, i)$
    }
$parse()$ :
    $add(L_S, [], 0)$
    $L_0$ :
      **if** $(\mathcal{R} \neq \emptyset)$ {
        $(L, c_u, c_i) \leftarrow \mathcal{R}.pop()$
        **goto** $L$
      } **else if** $(c_i = |I|)$ {
        **return** SUCCESS
      } **else** {
        **return** FAILURE
      }
    $L_S$ :
      **if** $(I[c_i] = a)$ {
        $add(L_{S_0}, c_u, c_i)$
        $add(L_{S_1}, c_u, c_i)$
      } **else if** $(I[c_i] = \$)$ {
        $add(L_{S_1}, c_u, c_i)$
      }
      **goto** $L_0$
    $L_{S_0}$ :
      **if** $(I[c_i] = a)$ {
        $c_i \leftarrow c_i + 1$
        $c_u \leftarrow create(L_{S_{0'}}, c_u, c_i)$
        **goto** $L_S$
      }
      **goto** $L_0$
    $L_{S_{0'}}$ :
      $pop(c_u, c_i)$
      **goto** $L_0$
    $L_{S_1}$ :
      **if** $(I[c_i] = a)$ {
        $c_i \leftarrow c_i + 1$
        $pop(c_u, c_i)$
      }
      **goto** $L_0$
```

```
$L_{S_2}$ :
  **if** $(I[c_i] = \$)$ {
    $pop(c_u, c_i)$
  }
  **goto** $L_0$
```

With this fairly simple set of modifications, we have gone from a vanilla recursive-descent parser all the way to a fully-general, GLL parser. Surprisingly, this technique also provides other benefits. Consider the following grammar:

$$S ::= S\ a \mid a$$

This is a fairly natural left-recursive grammar matching one or more a tokens. If we were using conventional recursive-descent, we would be unable to create a parser from this grammar. This is because recursive-descent works by calling the corresponding parse function for any non-terminals as they are encountered. When we step through the first production for the S non-terminal, the very first rule we encounter is again for the non-terminal S. This means that the *very first* thing a recursive-descent parser would do when parsing this grammar would be to dispatch to the (hypothetical) $parseS()$ function, which would in turn dispatch back to $parseS()$, and so on. Even on modern hardware, this process would very quickly run out of stack space.

However, a GLL parser treats disjunctions somewhat differently. Rather than persuing each alternate in sequence, a GLL parser pushes every potential alternate onto the dispatch stack. In the case of the S non-terminal, this would be both the first *and* second alternates as a is the first token in both cases. Assuming that the input stream represents the string "a, a, a", the descriptor representing the second alternate would parse the first a from the stream and then report back to L_0 with an empty dispatch stack. However, as there are two more tokens to consume, this derivation will fail and we will be left with the left recursive case for $c_i = 0$. This descriptor will be popped and its relevant actions executed. Of course, the very first action this descriptor will perform is push a new descriptor for the S non-terminal and $c_i = 0$.

This is the point at which recursive-descent breaks down. However, GLL maintains a set of labels which have already been pushed for a given index. At this point in the parse, that set $U_0 = \{L_{S_0}, L_{S_1}\}$. In other words, the algorithm will prune the left-recursion before it explodes, preventing the infinitely-recursive descriptor from being pushed onto $\mathcal{R}$.

If you actually trace the control flow of the GLL parser for our left-recursive grammar, you begin to see that something very interesting is happening in the GSS. For one thing, the post-left-recursive attempt to parse according to the second alternate is going to unify with the pre-left-recursive attempt (which has already succeeded). This represents a point in the GSS where two branches come together. The really interesting thing is what happens in the GSS with the post-left-recursive attempt to parse the first alternate. The set U_0 will prevent the redundant descriptor from entering the dispatch stack, but not before a new edge is created between an "earlier" node in the GSS and the current head. In other words, the GSS is no longer a directed, acyclic graph; we have introduced a cycle directly representing the left-recursion in the grammar. Thus, when we pop back through the GSS (on reduction), we consider not only the parent node representing the root of the left-recursive trace, but also the left-recursive sub-graph itself. We literally model left-recursive reduction as an ambiguity which can result in an arbitrary number of repetitions.

This same technique even applies to hidden left recursion, something which is beyond the capabilities of GLR [10], to say nothing of LALR [1] or LR(k). Consider the following modification of our earlier "as" grammar, incorporating hidden left-recursion:

$$\begin{aligned} S &::= A\ S\ a \mid a \\ A &::= \epsilon \end{aligned}$$

It is easy to see that the S non-terminal is left-recursive in its first production. Unfortunately, for a standard LR automaton, this sort of grammar is impossible to handle. There are modifications to the GLR algorithm (such as RIGLR [6]) which do allow for hidden left-recursion, but the conventional way of addressing the problem is to simply factor it out of the grammar.

With GLL, the fact that A is nullable poses no problems whatsoever. If you recall, the key element of the algorithm that allows left-recursion is the set U_i. Whenever a GLL parser encounters left-recursion, there will be some attempt to push onto $\mathcal{R}$ a descriptor for a label and index that has already been handled — and thus, included in U_i. Hidden left-recursion leads to exactly the same situation. The parser will recurse into the A non-terminal, consume exactly zero tokens and then pop back to the first production of S. At this point, it will attempt to push the hidden left-recursive descriptor for S onto the dispatch stack. Since we didn't consume any tokens in A, we're still at the same value of c_i. Thus, the hidden left-recursion will be caught by U_i just as readily as the non-hidden form.

In short, GLL can handle *any* context-free grammar. Hidden left-recursion, cyclic ambiguity, none of these elements pose any serious problems. Because GLL is just a variant of RIGLR, there exist a set of formal proofs which clearly establish the correctness of the algorithm and the full breadth of grammars it can manage.

3 GLL Combinators

In Section 1, we saw how to construct recursive-descent parsers in a functional and highly-compositional fashion. In Section 2, we saw how we could modify hand-written recursive-descent parsers into a more-powerful, fully-generalized equivalent. Of course, this begs the obvious question: can we combine these two techniques and construct GLL parsers in a functional and highly-compositional fashion?

The answer is "yes", but it does require some re-imagining of the GLL algorithm. The core problem is that GLL was designed for an imperative language with an *unrestricted* `goto`. The pseudo-code given in Section 2 makes it fairly clear that the algorithm depends upon several shared mutable data structures. The unrestricted `goto` is absolutely essential for the L_0 dispatch area. When a descriptor is popped off of the dispatch stack ($\mathcal{R}$), our pseudo-code uses `goto` to jump to the location given by that descriptor's label. However, this means that `goto` must be able to jump to locations which are not statically predetermined, a feature which is fairly rare in languages.

The original implementation of GLL was in C, a language which supports shared mutable data structures, but not the unrestricted `goto`. To get around this limitation, Scott and Johnstone used a widely-branching `switch`/`case` statement where each `case` corresponded to a label. With this modification, descriptors could use integers to describe jump locations. The `switch`/`case` would serve as the prerequisite "unrestricted" `goto`, since its jump destination is unknown until runtime. Scott and Johnstone have proposed that this same technique could be used to implement GLL in languages like Java which do not support any form of `goto`, unrestricted or otherwise.

While it is true that we could adapt the "giant `switch`/`case`" approach to implement a GLL parser in Scala, the result could hardly be called either functional or compositional. Optimally, we should be able to define parsers as atomic instances which are then put together one at a time to create more complex parsers. This design is obviously incompatible with a single, giant `switch`/`case`, so we need to find some other way of looking at the algorithm at its most fundamental level.

3.1 Trampolines, L_0 and Tail-Recursion

The critical observation which allows us to render GLL in a purely-functional setting is to see that GLL is really much like a tail-recursive form of recursive-descent. In functional programming languages, algorithms are often implemented first using recursion in whatever fashion seems natural. Later, these same algorithms are often adapted to be tail-recursive for efficiency. Unfortunately, many algorithms (including recursive descent) perform their recursive calls "in the middle" of their straight-line execution, meaning that a tail-recursive rendering of the algorithm must leverage continuations in order to procede with the remainder of the tasks which must be performed by that function call. As the terminology was intended to suggest, GLL continuation labels (e.g. $L_{S_0'}$) fit this role precisely.

Tail-recursion is more efficient in many functional languages because the runtime is able to perform the recursion in constant stack space, discarding each frame as soon as the recursive dispatch takes place. This is very easy to implement when the function in question exhibits *self*-tail-recursion[1], since the tail-recursive call can be translated directly into a `jump`. However, many algorithms (including recursive-descent) are *mutually*-recursive, meaning that any tail-recursive rendering thereof would have to be mutually-tail-recursive. This poses a problem that is very familiar to those who have implemented a functional language on platforms like the JVM or CLR. Scala itself can only optimize self-tail-recursion; optimization of mutual-tail-recursion would require extra features from the JVM.

The generally-accepted solution to this problem of mutual-tail-recursion is called *trampolined dispatch*. In loose terms, all mutually-tail-recursive dispatch is handled by a single loop. Any trampoline-enabled function must return a thunk wrapping the required dispatch. In this way, the call stack always contains zero or one frames, regardless of how many tail-recursive calls are made.

Now, if GLL is a tail-recursive variant of recursive-descent, then the L_0 label would be the trampoline. Every label in the parser eventually jumps back into L_0, effectively turning it into a `while`-loop, running as long as the dispatch stack contains more descriptors. The only difference between L_0 and a standard trampoline is that in GLL, the trampoline loop is dealing with more than one queued dispatch. A standard trampoline will only have to handle and dispatch one thunk at a time. The L_0 loop in GLL has an entire stack ($\mathcal{R}$) of dispatches to be processed, effectively representing a depth-first search through all potential parse paths.

It is worth noting at this point that GLL does not actually depend upon $\mathcal{R}$ being a stack. We could also implement $\mathcal{R}$ using a queue. This would convert the parser from depth- to breadth-first search. In practice however, this minor difference imposes some fairly high performance penalties — sometimes as high as 50%. For this reason, we have and will continue to confine our discussion to stack-based GLL parsing.

3.2 Functional Descriptors and Combinators

Having established the L_0 dispatch loop as a functional trampoline, it's not hard to see that the label descriptors we push onto $\mathcal{R}$ could actually be implemented using function values. Thus, instead of jump labels or `switch`/`case` branches for each production alternate (e.g. L_{S_0}, $L_{S_{0'}}$, L_{S_1}, etc), we could actually move the "alternate routines" each into their own function. Whenever we push a descriptor onto the $\mathcal{R}$, that descriptor would contain a pointer to the relevant function, rather than a jump label.

This conversion not only eliminates the need for an unrestricted `goto`, it completely removes the need for any `goto` whatsoever! We still rely upon shared mutable data structures ($\mathcal{R}$, $\mathcal{P}$, etc), but we have made some fairly significant progress toward a functional design. We will revisit this issue of shared mutable data structures later on. For now, the only remaining problem is one of composability...

With parser combinators (Section 1), we were able to take a set of mutually-recursive functions representing a recursive-descent parser and split them appart into discrete, composable units. We did this by seeing recursive-descent in terms of three primitive operations: sequence, disjunction and literal. These three operations were used to define three different types of `Parser`(s), and instances of these `Parser`(s) were composed together into larger, more complex parsers (either sequential or disjunctive). Since GLL is merely a trampolined, tail-recursive variant of recursive-descent, it only makes sense that we should be able to play the same trick and achieve a combinatorial implementation.

The one wrinkle in this plan is that we can no longer rely upon parsers returning either `Success` or `Failure` to determine the next course of action. As with most mutually-tail-recursive implementations, we will need to use continuations in the form of anonymous methods[2] to "continue the work" when one parser dispatches to another. These continuations will take as a parameter the result of the previous parse — what used to be a return value. Since the parsers no longer return any meaningful result, relying instead on trampolined dispatch to the continuation of that parse, the main "parsing" function of each `Parser` implementation will actually return `Unit`. This may not seem like a very "functional" design since it immediately

[1]That is, when the tail-recursive call is directly back to the function itself.

[2]Scala's name for function literals

```
trait Parser[+A] extends (Stream[Char] => Result[A]) {
  def apply(in: Stream[Char]): List[Result[A]]

  /**
   * Abstract def for the main parse function of each
   * Parser implementation.
   */
  protected def chain(t: Trampoline, in: Stream[Char])(f: Result[A] => Unit)
}

class Trampoline {
  // L_0
  def run() { ... }

  def add(p: Parser[A], in: Stream[Char])(f: Result[A] => Unit) { ... }

  def remove(): (Parser[Any], Stream[Char]) = { ... }
}
```

Listing 2: Proto-Implementations of `Parser` and `Trampoline`

implies the use of side-effects, but as we shall see later, this perversion is only for convenience and efficiency. We can eliminate both the side-effecting parse functions *and* the shared mutable data structures with a single, reasonable straightforward modification in design.

The other concerning aspect of GLL is the actual trampoline and its interaction with those troublesome mutable data structures. The simplest way to approach this is to wrap up the trampoline loop, all of the shared mutable state *and* the two all-important utility functions (*add*(...) and *pop*(...)) in a single class: `Trampoline`. Each parse process (starting at the head of the input stream and consuming completely to the end) will have exactly one instance of this class, created at the beginning. To avoid unnecessary shared state, this instance will be passed to each parse function along with the current input stream, which will be represented as an instance of `Stream[Char]` (as it was in Section 1).

Having made these design decisions, we can begin to outline the `Parser` and `Trampoline` classes which will be so foundational in our final implementation. This basic outline is given in Listing 2. Unlike traditional parser combinators (given in Listing 1) which handle anything interesting within their `apply` method, GLL combinators will perform most of their work in the `chain` method. This method implements the "main parse function" idea as described earlier. In order to preserve the API, every `Parser[A]` will also be a function from `Stream[Char]` to `Result[A]`, which means that every `Parser` implementation must define an `apply` method which serves as the entry point for the parser. We will provide a more complete definition of this function presently.

3.3 Terminals, Non-Terminals and the `apply` Method

In the interest of both efficiency and organization, we will group `Parser` implementations into two categories: terminals and non-terminals. At a high level, a terminal parser is defined as any `Parser` which does *not* contain a disjunction. Thus, any composition of sequential and literal parsers would qualify as a terminal. Conversely, a non-terminal parser is defined as any `Parser` which does contain a disjunction. This disjunction may actually be nested within several sequential parsers; the "top-level" sequential parser in this nesting would still be classified as a non-terminal. From these definitions, we can infer that disjunctions are *always* non-terminals, literals are *always* terminals, and sequences may fall into either category, depending on their nested composites.

It is worth noting that these definitions of terminal and non-terminal differ slightly from those conventionally used to describe context-free grammars. This is necessary as a grammar defined using parser combinators does not have any labeled non-terminals in the conventional sense. Every grammar is essentially defined in Chomsky Normal Form, making the conventional definition of "non-terminal" useless.

These definitions are significant as they immediately convey something about the grammar represented by the underlying `Parser`: whether it can be parsed using conventional recursive-descent and thus, conventional parser combinators; or if it *might* require the full power of GLL. By definition, terminal parsers do not require any "decisions" to be made. There is no reason to impose the complexity of GLL on what is essentially a recursive-descent parse. On the other hand, non-terminal parsers *might* be compatible with traditional recursive-descent, but this is not a certainty. Thus, any `Parser` which claims to be a non-terminal will require the creation of a new `Trampoline` and the added complexity of the continuation-based implementation.

Since we know literal parsers to be terminal by definition, it is fairly easy to define the relevant `Parser` implementation. In fact, it is almost identical to the implementation required for a traditional framework of parser combinators. Listing 3 gives the definition of `LiteralParser` along with `TerminalParser` and `NonTerminalParser`, the only direct sub-classes of the `Parser` trait.

For a `TerminalParser`, the `apply` method is where all of the action is. There is no need to carry around a trampoline or go out of our way to use the continuation-passing style. This simplicity allows us to implement the `chain` method as a mere delegate to `apply`. However, in a `NonTerminalParser`, the `apply` method cannot perform any sort of parse action. Instead, it must create a new instance of `Trampoline` and push its own descriptor onto the dispatch stack (using the `chain` method):

```
trait NonTerminalParser[+A] extends Parser[A] {
  ...

  final def apply(in: Stream[Char]) = {
    val t = new Trampoline

    chain(t, in) {
      case s: Success => ...
      case f: Failure => ...
    }
    t.run()

    ...
  }
}
```

In terms of the API, the `apply` method for a given `Parser` instance will only be called *once*: by the API consumer to initiate the parse process. After that, any internal framework dispatch will be made on the `chain` method. The only exceptions to this rule are terminal parsers, but as these parsers do not perform any GLL-specific operations, their use of the `apply` method is not a concern. The point is that `apply` may be relied upon to perform initialization operations for a specific parse. Since GLL initialization is the same for any non-terminal parser, these operations can be implemented safely in the superclass.

We have said before that the `chain` method is the main parse function for each non-terminal, but this doesn't completely encompass its functionality. In general, `chain` takes care of pushing a single parser's descriptors onto the dispatch stack. Thus, if we need to queue up an entire parser (whatever that entails), we can simply dispatch to its `chain` method, passing the parse continuation as a parameter (`f`). This continuation will be invoked once for each outcome produced by the parser: either `Success` or `Failure`. As the algorithm supports ambiguity, there may be more than one `Success` produced by a single parser.

We now have enough information to actually define `SequentialParser`, the `NonTerminalParser` implementation which will handle sequences. Note that it will also be necessary to implement `SequentialParser` as a subclass of `TerminalParser`. Rather than attempting to merge these orthogonal cases, we will implement two different sequential parsers. The non-terminal case will simply queue up its first parser, passing a continuation which queues up its second parser on `Success`:

```
case class ~[+A, +B](_1: A, _2: B)

class SequentialParser[+A, +B](left: Parser[A], right: Parser[B]) extends NonTerminalParser[A ~ B] {
  def chain(t: Trampoline, in: Stream[Char])(f: Result[A ~ B] => Unit) {
    left.chain(t, in) {
      case Success(res1, tail) => {
        right.chain(t, tail) {
          case Success(res2, tail) => f(Success(new ~(res1, res2), tail))

          case res: Failure => f(res)
        }
      }

      case res: Failure => f(res)
    }
  }
}
```

The thing to notice here is that we *still* haven't actually `push`ed anything onto the `Trampoline`, we're just delegating down to `chain` methods. This is because the dispatch stack is a way to manage alternative parse paths. Since there are no alternatives in a sequence, there is no need to exploit this feature in any way.

An instance of `SequentialParser` will be returned from the `~` method for any sequence where either the left- or the right-operand are instances of `NonTerminalParser`. Thus, we can immediately provide an implementation for the `~` method in `NonTerminalParser`:

```
trait NonTerminalParser[+A] extends Parser[A] {
  def ~[B](that: Parser[B]) = new SequentialParser(this, that)
}
```

This is valid because the left-operand for the `~` operator is already known to be a `NonTerminalParser`. We don't need to check the right-operand as our result will be a `NonTerminalParser` regardless. The implementation of `~` for `TerminalParser` is not quite so straight-forward:

```
trait TerminalParser[+A] extends Parser[A] { self =>
  def ~[B](other: Parser[B]) = other match {
    case that: TerminalParser[B] => {
      new TerminalParser[A ~ B] {
        def apply(in: Stream[Char]) = self(in) match {
          case Success(res1, tail) => that(tail) match {
            case Success(res2, tail) => Success(new ~(res1, res2), tail)
            case f: Failure => f
          }

          case f: Failure => f
        }
      }
    }

    case that => new SequentialParser(this, that)
  }
}
```

And here we have our sequential parser for the terminal case. This is quite analogous to the implementation for the non-terminal case save that we use `apply` directly rather than working through `chain` and using continuations.

3.4 Dealing with Alternates

The one parser we have yet to address from our traditional parser combinators is the `DisjunctiveParser`. This is the parser which must manage more than one alternate sub-parser and make a decision as to which branch is correct based on the input stream. In a recursive-descent parser, disjunctions are handled by predicting the correct production based on FIRST and FOLLOW sets, computed ahead of time. In our parser combinator implementation from earlier, disjunctions are handled in an almost "brute force" manner: each possible production is handled in turn; if any alternate fails, the next production is tried until one succedes or there are no more possibilities.

As demonstrated in Section 2, GLL disjunctions fall somewhere between these two techniques. A GLL parser does predict the correct production(s) for a given input, but it doesn't restrict this prediction to just a single production. Whenever the PREDICT set is ambiguous, all possibilities are pushed onto the dispatch stack and each is handled in turn. This immediately suggests the implementation of `DisjunctiveParser` for our GLL combinators framework. We must gather all possible alternates at a given point and use the `add` method of our `Trampoline` to simultaneously queue them up for eventual execution. A more sophisticated implementation could perform some basic predictive queueing (as in a hand-written GLL parser), but we will omit this optimization for the sake of simplicity.

The only problem we have is the fact that disjunctions in a parser combinator framework are composed of exactly two alternates. Productions which logically contain more than two alternates are represented by nesting disjunctions in a left-associative fashion. For example, we would encode the rule $a|b|c|d$ as $((a|b)|c)|d$. This poses an annoying issue, since we must queue *every* alternate possible at a given point in the parse. Thus, we must somehow flatten our nested tree of disjunctions to produce a list of all possible alternates reachable from the current disjunction. In Scala, this can be done using a very simple recursive visitation of the parser tree. In Haskell or some other functional language, a more complicated scheme would be required:

```
class DisjunctiveParser[A](l: =>Parser[A], r: =>Parser[A]) extends NonTerminalParser[A] {
  lazy val gather = gatherImpl(Set()).toList

  private def gatherImpl(seen: Set[DisjunctiveParser[A]]): Buffer[Parser[A]] = {
    val newSeen = seen + this

    def process(p: Parser[A]) = p match {
      case d: DisjunctiveParser[A] => {
        if (!seen.contains(d))
          d.gatherImpl(newSeen)
        else
          new ListBuffer[Parser[A]]
      }

      case p => p +: new ListBuffer[Parser[A]]
    }

    process(left) ++ process(right)
  }
}
```

Once we have our flat list of alternates, we need only iterate each possibility, pushing it onto the dispatch stack:

```
def chain(t: Trampoline, in: LineStream)(f: Result[A] => Unit) {
  // eliminate duplicate results
  val results = mutable.Set[Result[A]]()
```

```
    for (p <- gather) {
      t.add(p, in) { res =>
        if (!results.contains(res)) {
          f(res)
          results += res
        }
      }
    }
  }
```

3.5 The Trampoline

Now that we have seen how the basic parsers can be implemented in terms of GLL, we can finally return to the all-important `Trampoline`, outlined in Listing 2. This class has two main functions:

- It encapsulates the dispatch stack and provides a `run()` method which loops while the stack is non-empty, popping off descriptors and dispatching as appropriate.
- It maintains the GSS in the form of several associated data structures.

The `Trampoline` class is the only traditional, imperative-style class in the entire implementation. All of its operations side-effect. This is done mostly for the sake of efficiency and convenience in the implementation; it is not a fundamental restriction of the algorithm. If we were implementing GLL in Haskell, we would likely return a modified `Trampoline` from the `chain` function for each combinator rather than modifying its data structures in-place.

As seen earlier, every `chain` function takes the trampoline as a parameter. This is a simple requirement to satisfy as all of the dispatch to the `chain` methods is done either from the `chain` method in another parser (as in the case of `SequentialParser` or from the trampoline itself). With this in mind, we can sketch an initial outline of the `run()` method in the `Trampoline`:

```
class Trampoline {
  private val stack = new mutable.Stack[(Parser[Any], Stream[Char])]

  def run() {
    while (!stack.isEmpty) {
      val (p, s) = remove()

      p.chain(this, s) { res =>
        ...
      }
    }
  }

  private def remove() = stack.pop()
}
```

Every descriptor is a 2-tuple consisting of a parser and a pointer into the input stream. This allows us to kick off the next parser in line, but it isn't quite sufficient yet. We also have to deal with parser continuations. This is handled in the conveniently-elided function value above.

As mentioned earlier, the `Trampoline#add` method takes a parser and a stream, as well as a function value which represents the continuation of that parse process. These continuations literally represent the back-edges for the GSS. At any point in the stream, there will be a set of mappings from parser instances to sets of continuations. Thus, GSS nodes are actually descriptors represented by the 2-tuple of a parser and a pointer into the input stream:

```
type FSet[A] = mutable.Set[Result[A] => Unit]

private val backlinks = mutable.Map[Stream[Char], HOMap[Parser, FSet]]()
```

As an aside, we will make some use of the `HOMap` data structure[3] in our implementation. Please be aware that this is *not* the same as a `HMap`. `HOMap` is simply a map where the key and value types are of kind $* \Rightarrow *$ and their type parameters are instantiated with the same (unrestricted) type. In simpler terms: both the key and value types are functions of the same type. This allows the parameter type of the values extracted from the map to vary based on the parameter type of the key used to extract that value. This is required for type-safe continuation dispatch, as we will see in a moment.

With the `backlinks` map defined, we can now move ahead with our implementation of the `add` method:

```
def add[A](p: Parser[A], s: Stream[Char])(f: Result[A] => Unit) {
  if (!backlinks.contains(s))
    backlinks += (s -> HOMap[Parser, FSet]())

  if (!backlinks(s).contains(p))
    backlinks(s) += (p -> new mutable.HashSet[Result[Any] => Unit])

  backlinks(s)(p) += f

  val tuple = (p, s)
  stack.push(tuple)
}
```

Now that we have continuations saved in the GSS, we can properly implement the elided `chain` continuation in our `run()` method from earlier:

```
p.chain(this, s) { res =>
  backlinks(s)(p) foreach { _(res) }     // this is where we need HOMap
}
```

This takes care of the dispatch stack and the GSS back-edges, but we also need to maintain a number of other data structures. Primarily, we need a way of tracking parsers which have already been pushed onto the dispatch stack for a particular index into the input stream (we call this set U_j in Section 2).

This set can be represented in a relatively straightforward manner, mapping from the input stream to a set of `Parser`(s):

```
private val done = mutable.Map[Stream[Char], mutable.Set[Parser[Any]]]()
```

It is important to note that we are assuming here (as well as for the `backlinks` set) that `Parser` instances have some sort of intrinsic identity. In other words, if we have a `Parser` which represents a particular non-terminal, we should be able to test that parser for equality and receive `true` if the compared parser represents the same non-terminal.

This is very different from how traditional parser combinators operate. Consider the following grammar constructed using the framework described in Section 1:

```
def s: Parser[Any] = (
    "a" ~ s
  | "a"
)
```

[3]Credit to Jorge Ortiz for this very useful type-safe wrapper

The important thing to remember here is even though the syntax may make `s` *appear* as a non-terminal in a grammar, it is still technically a method. By definition, repeated invocations of the `s` "non-terminal" actually produce new instances of `Parser`. Thus, if we attempt to parse the string "`aaa`", three separate instances of `Parser` will be created, each corresponding to the `s` "non-terminal".

This technique works well for traditional parser combinators, but it breaks down as soon as we need to associate any sort of semantics with specific `Parser` instances (such as ensuring that we don't repeatedly invoke the same `Parser` on a particular index of the input stream). We could attempt to define equality for `Parser` in such a way that structurally equivalent parsers are recognized as being the same, but this approach very quickly runs into problems with recursive parsers. Actually, it's not surprising that we are unable to define a structural `equals` method for `Parser` given the fact that determining language equality for context-free grammars is undecidable.

The solution to this problem is to ensure that repeated invocations of `s` (or any other name corresponding to a particular non-terminal) return the same `Parser` instance. If we can guarantee this, then equality can be handled at the instance-level, neatly side-stepping the tangled morass of structural equality. Fortunately, Scala allows lazy values to have recursive definitions, giving us a convenient way to guarantee that we always have a consistent `Parser` instance for each non-terminal in the grammar. Thus, the grammar from above could be handled almost as-is. The only change required is to swap `def` for `lazy val`[4]:

```
lazy val s: Parser[Any] = (
    "a" ~ s
  | "a"
)
```

It is interesting to note that the parser combinator framework included in Scala 2.8 ran into a similar difficulty implementing Packrat parsing and arrived at exactly the same solution. The Kiama framework [8] imposes a similar restriction on its parser combinators.

Now that non-terminals are guaranteed to have a consistent `Parser` identity, we can complete the modifications to our `add` method with respect to the `done` set. Rather than simply pushing the descriptor onto the dispatch stack, we first check to see if the descriptor has been previously pushed by looking in the `done` set for the current index into the input stream:

```
...

val tuple = (p, s)

if (!done.contains(s))
  done += (s -> new mutable.HashSet[Parser[Any]])

if (!done(s).contains(p)) {
  stack.push(tuple)
  done(s) += p
}
```

This is sufficient to ensure that we don't repeatedly consider a given parser at a particular index into the input stream, pruning off redundant suffixes. However, this is not sufficient to ensure correctness in every case. When a local ambiguity merges with a parse trail which has yet to be followed (still waiting on the dispatch stack), the additional GSS back-edge will allow reduction to follow through the ambiguous parse trail. However, if a local ambiguity merges with a parse trail which has *already* been followed and reduced, we must take extra steps to immediately reduce along the ambiguous path using the results obtained by the ealier parse. Thus, we must maintain not one, but *two* sets of completed descriptors: one which represents

[4]We also explored a slightly more devious solution which involved reflectively determining the auto-generated container class for the left and right thunks contained by `DisjunctiveParser`. This worked reasonably well, but it was far less straight-forward than simply requiring `lazy val` and had some rather murky performance implications.

descriptors which have been pushed onto the dispatch stack, and another which maps *popped* descriptors to a set of results. This second set is the $\mathcal{P}$ set from the psuedo-code description of the GLL algorithm in Section 2:

```
type SSet[A] = mutable.Set[Success[A]]

private val popped = mutable.Map[LineStream, HOMap[Parser, SSet]]()
```

Here once again we're using `HOMap` to associate `Parser` instances with a set of `Success`(es). We need this to be a set (rather than a single `Success`) because a single node in the GSS may be traversed several times, resulting in more than one possible result during reduction.

Whenever we add a descriptor, we must now check it against the `popped` set *before* we check against the `done` set. If we find the relevant descriptor in `popped`, we immediately reduce on that parser, once for each `Success` in the set:

```
...

val tuple = (p, s)

if (popped.contains(s) && popped(s).contains(p)) {
  for (res <- popped(s)(p)) {
    f(res)
  }
} else {
  if (!done.contains(s))
    done += (s -> new mutable.HashSet[Parser[Any]])

  if (!done(s).contains(p)) {
    stack.push(tuple)
    done(s) += p
  }
}
```

Of course, this code isn't very useful if we never add anything to the `popped` set. As indicated by the psuedo-code in Section 2, we take care of this bit in the trampoline reduce continuation immediately prior to traversing the GSS back-edges:

```
p.chain(this, s) { res =>
  if (!popped.contains(s))
    popped += (s -> HOMap[Parser, SSet]())

  if (!popped(s).contains(p))
    popped(s) += (p -> new mutable.HashSet[Success[Any]])

  res match {
    case succ: Success[Any] => popped(s)(p) += succ
    case _: Failure => ()
  }

  backlinks(s)(p) foreach { _(res) }
}
```

At this point, we have fully implemented the pseudo-code given in Section 2. However, this alone is not quite sufficient to allow for fully-general GLL parsers. The critical distinction is the pseudo-code given was for a GLL *recognizer* while our GLL combinators framework serves as a tool for building GLL *parsers.*

As we note in Section 4, this distinction does have some interesting performance implications. Unfortunately, it also has an important effect on the correctness of our algorithm. Consider the following grammar:

```
lazy val s: Parser[Any] = (
    s ~ "a"
  | "a"
)
```

This is a simple left-recursive encoding for the language of one or more `a`(s). As we showed in Section 2, GLL can handle left-recursive grammars by treating the left-recursion as an ambiguity and by creating a loop in the GSS. Our implementation represents such loops with indirect self references in the `backlinks` set. More specifically, we add a reduce function to `backlinks` which indirectly reduces back to the same state we are currently in. This allows left-recursive productions to loop as many times as necessary to consume the input in question. Unfortunately, it can also lead to non-terminating execution.

Upon reflection, the problem becomes quite apparent. We are pushing a function which indirectly calls back to ourselves into the `backlinks` set. We eventually traverse the `backlinks` set and call every function therein. One of these functions will lead back to where we started, at which point we once again traverse the backlinks set, calling the same function we dealt with earlier.

Clearly we need some way of preventing a backedge from being traversed indefinitely for a particular index of the input stream. This is remarkably similar to the problem we faced with our `done` and `popped` sets, except instead of repeated invocation of a particular `Parser`, we need to prevent repeated invocation of a particular function. Function equality is just as undecidable as parser equality, but once again we can exploit the fact that we have real a real object which represents the entity in question. In Scala, functions are objects, and so we can simply rely on instance identity to catch repeated invocation of the same function:

```
private val saved = HOMap[Result, FSet]()
```

Recall that `Result` encapsulates not only the `Success` or `Failure` of a reduction but also the current location of the input stream. Thus, it is sufficient to map a particular `Result` to the set of all reduce functions which have been called with that value. This prevents a single result from looping around and around *ad infinitum* when our GSS contains a loop:

```
...

if (!saved.contains(res))
  saved += (res -> new mutable.HashSet[Result[Any] => Unit])

for (f <- backlinks(s)(p)) {
  if (!saved(res).contains(f)) {
    saved(res) += f
    f(res)
  }
}
```

Instead of just looping over the set of relevant GSS back-edges, we now check each function against the `saved` set for the current `Result`. If we haven't yet invoked this particular back-edge, store the function in the `saved` set and go ahead with the call. If the back-edge in question represents a loop, we will eventually come back to this exact point with the same `Result`. However, instead of recursively continuing forever, the infinite loop will be nipped in the bud when the `saved` set reports that the function in question has already be invoked for that particular `Result`.

3.6 Putting it All Together

Now that we have a complete implementation of the `Trampoline` class, we can finally implement the `apply` method in `NonTerminalParser`:

```
final def apply(in: Stream[Char]) = {
  val t = new Trampoline

  var successes = Set[Success[A]]()
  var failures = Set[Failure]()

  chain(t, in) {
    case Success(res, Stream()) => {
      successes += Success(res, Stream())
    }

    case Success(_, tail) => {
      val msg = "Unexpected trailing characters: '" + tail.mkString + "'"
      failures += Failure(msg, tail)
    }

    case f: Failure => failures += f
  }
  t.run()

  if (successes.isEmpty)
    failures.toList
  else
    successes.toList
}
```

This is really nothing more than a starting point for the trampoline process. We create a new `Trampoline` and use it to invoke our own `chain` method. The most interesting part of this function is way in which a failed parse is identified. By its very nature, the GLL algorithm will report `Failure` for a large number of parse trails even for parse which will eventually succeed. This is because *every* `Failure` is propagated all the way back the through the GSS to the root (which is precisely the `apply` function in `NonTerminalParser`), even when the `Failure` in question is for a local ambiguity which was resolved along another parse trail.

In order to provide sane semantics, we define a parse to be a failure iff *all* results are `Failure`(s). If any `Success` is received, it means that one possible parse trail has run to completion and successfully reduced. We make the (rather optimistic) assumption that this `Success` is the desired result and drop any `Failure`(s) we may have received. If we don't receive any `Success`(es), we assume the parse is a failure and return *all* `Failure`(s) received. The caller can then impose whatever hueristics are desirable to determine which is the "definitive" `Failure`. For example, we could implement a longest-match error reporting strategy in the following way:

```
lazy val p: Parser[Any] = ...

p(input) match {
  case successes @ (_: Success[Any]) :: _ => ...     // handle ambiguous successes

  case failures @ (_: Failure) :: _ => {
    val sorted = failures sort { _.tail.length < _.tail.length }
    val longest = sorted takeWhile { _.tail.length == sorted.head.tail.length }
    // traverse `longest` and report all errors
  }
}
```

It is also worth noting that we do not allow `Success`(es) which have failed to consume the entire input stream. We will actually receive a `Success` *every* time a `Parser` reduces successfully. While this is technically correct (reflecting the ambiguity between greedy and non-greedy matching), it is almost never the desired semantics.

In order to stem this torrent of spurious `Success`(es), we make the assumption that greedy matching should always be the default. This assumption is natural considering that traditional parser combinators always match greedily. Arguably, it would be better to allow extension of this method to permit alternative strategies (in fact, our full practical implementation does exactly this). However, we will conveniently ignore this practical consideration for the sake of simplicity.

4 Performance

Of course, with any generalized parsing algorithm, performance is an important concern. Generalized parsing can be naïvely handled in $O(k^n)$ time, clearly we should aim to be better than this. Conversely, it has been shown that context-free language recognition is equivalent to matrix multiplication, a problem for which the fastest-known algorithm is $O(n^{\log_2 7}) \approx O(n^{2.807})$ and the proven lower-bound is $O(n^2)$, so it is highly unlikely that our GLL implementation could do any better than this. In Section 2.3, we made the claim that GLL is an inherently $O(n^3)$ algorithm which gracefully degrades to $O(n)$ when the grammar is LL(1). In this section, we give a rough proof sketch for both of these claims as well as emperical evidence that our implementation maintains the performance properties of the theoretical algorithm.

Note that we will only consider GLL recognizers in our proofs. Generalized context-free parsing remains an inherantly exponential problem because all possible results must be produced. A recognizer can prune redundant parse trails when they are detected to converge to the same point; but a parser must consider not only the point of convergence but also the possible values reachable from that point. Put another way, it is very possible to construct a generalized parser which produces an exponential number of results. At the very least, these results must be enumerated by the parser if only for the sake of output. Thus, it is possible for the values produced by semantic actions to utterly defeat the optimizations employed by generalized parsing algorithms.

4.1 Linear Degradation

GLL is $O(n)$ in the case where each disjunction in the grammar is unambiguous. This is to say that each disjunction has at most one valid production for any given input. Recall that left-recursion is treated as an ambiguity by GLL. Thus, the class of grammars which can be handled by GLL in $O(n)$ time is precisely equivalent to LL(1).

Linear performance is achieved due to the fact that the parser will never follow more than one parse trail. Whenever the parse reaches a disjunction, at most one production will be valid (by assumption) and thus at most one descriptor will be pushed onto the dispatch stack. This descriptor will be immediately popped off by the L_0 trampoline. Thus, GLL will behave exactly like a predictive, tail-recursive form of recursive-descent, an algorithm which is known to be $O(n)$.

4.2 Generally Cubic

Our proof for $O(n^3)$ performance in the worst case essentially relies on the fact that GLL is based around the lazy construction of the same graph-structured stack data structure used by GLR, RIGLR and other generalized bottom-up parsers. Algorithms based around this structure are known to be $O(n^3)$, and thus GLL trivially inherits this property.

The only case where GLL differs from a standard GSS-based parse algorithm is when dealing with left-recursive rules. Specifically, GLL treats left-recursion as a local ambiguity which creates a special loop edge in the GSS to be used during reduction. Thus, left-recursive rules will parse with the same performance as a right-recursive rule. The main difference comes during reduction: GLL will reduce to the origin of the left-recursive production as well as back into the left-recursive production itself. However, the loop reduction will complete in constant-time, either resulting in an additional parse trail in the GSS (if the reduction is ambiguous) or failing immediately. Thus, left-recursive reduction has the same performance characteristics as conventional reduction, indicating that GLL is indeed $O(n^3)$ in the worst-case.

4.3 Emperical Evidence

In an effort to reinforce the theoretical conclusions of the proofs, we have done some emperical testing using an implementation of the GLL combinators framework. Note that this implementation is significantly more advanced than the basic implementation given in Section 3. However, the differences are exclusively constant-time optimizations and API refinements designed to make the framework more practically applicable, they do not affect the asymptotic performance of the algorithm.

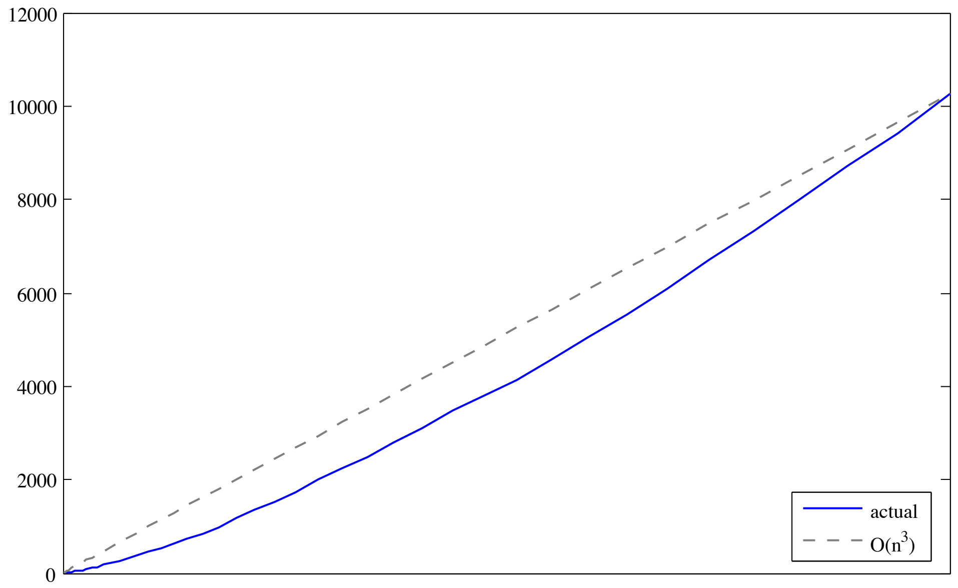

Figure 1: Performance on Language Γ_2

Figure 1 shows the results of our performance testing on the following highly-ambiguous grammar (which Scott and Johnstone refer to as "Γ_2"):

$$\begin{array}{rcl} S & ::= & S\ S\ S \\ & \mid & S\ S \\ & \mid & \texttt{a} \end{array}$$

In order to obtain an approximation of the asymptotic performance, we tested the framework using input lengths varying from 10 up to 100 at increments of 2. Each test run was performed fifteen times with the first three, highest and lowest results dropped. The remaining ten results were averaged and used as the runtime (in milliseconds) for that particular input length. In order to ensure JVM warmup, we the parser fifty times against a moderate input length before starting any testing. Listing 4 shows our complete test harness.

It is important to note that the parser used for testing defines all of its semantic actions to concatenate their string arguments together. As such, there is no ambiguity in the result values, effectively turning our parser into a simple recognizer.

The results in Figure 1 are plotted on a cubic scale. The curve described by the data on this scale is very close to linear, indicating that the framework is indeed providing $O(n^3)$ performance modulo some constant factor. The data curve does depart *slightly* from a pure cubic result (in fact, it is slightly better), but this abberation is well within the margin of experimental error, especially when considering the many advanced (and sometimes unpredictable) optimizations performed by the JVM at runtime.

All tests were performed on MacOS X with a 2.4 Ghz processor and 4 GB of memory running under Apple's JDK version 1.6.0_17 in 64bit mode (note that Apple's JDK runs in `-server` mode by default).

5 Conclusion

In this paper, we have presented a practical implementation of the GLL algorithm in a primarily-functional framework. While our implementation does make use of some mutable state, it is confined to very specific areas. As noted, it would be reasonably straightforward to replace the usage of mutable state with purely-functional alternatives. The primary motivation for the mutable state was convenience and not necessity.

In the interest of brevity, we have omitted many constant-time optimizations from the implementation presented in this paper. For example, we do not attempt to perform any sort of predictive disjunction handling. This optimization (along with many others) are fully explored in our practical implementation of the framework, available at `http://github.com/djspiewak/gll-combinators`. This implementation also provides several useful constructs not mentioned in this paper, such as support for regular expressions, parser negation (similar to SGLR's [11] reject reductions) and line tracking (for error reporting). The performance tests in Section 4 were done using this real-world framework implementation rather than the simplified form described here.

Given its obvious power and simplicity implementation, Scott and Johnstone — creators of the GLL algorithm — predicted that GLL could become the generalized parsing algorithm of choice, overtaking more traditional approaches like Earley and GLR. As this paper has shown, GLL is also quite amenable to remarkably simple and compositional frameworks such as parser combinators. We can only hope that future language implementors will consider using GLL in light of its vastly superior flexibility (when compared to common techniques such as LALR and LL(k)) and simplicity of implementation.

References

[1] Frank DeRemer, *Practical translators for LR (k) languages*, 1969.

[2] Daan Leijen and Erik Meijer, *Parsec: Direct style monadic parser combinators for the real world*, Tech. Report UU-CS-2001-27, Department of Computer Science, Universiteit Utrecht, 2001.

[3] Adrian Moors, Frank Piessens, and Martin Odersky, *Parser combinators in Scala*, Tech. Report CW491, Department of Computer Science, K.U. Leuven, 2007.

[4] M. Odersky, P. Altherr, V. Cremet, B. Emir, S. Micheloud, N. Mihaylov, M. Schinz, E. Stenman, and M. Zenger, *The Scala language specification*, `http://www.scala-lang.org/docu/files/ScalaReference.pdf`, 2007.

[5] Martin Odersky and al., *An overview of the Scala programming language*, Tech. Report IC/2004/64, EPFL Lausanne, Switzerland, 2004.

[6] Elizabeth Scott and Adrian Johnstone, *Generalized bottom up parsers with reduced stack activity*, 2005, pp. 565–587.

[7] ———, *GLL parsing*, Proc. of LDTA, 2009, pp. 113–126.

[8] A.M. Sloane, *Experiences with domain-specific language embedding in Scala*, Proceedings of the 2nd International Workshop on Domain-Specific Program Development, 2008.

[9] Masaru Tomita, *Graph-Structured Stack and natural language parsing*, Proceedings of the 26th annual meeting on Association for Computational Linguistics, Association for Computational Linguistics Morristown, NJ, USA, 1988, pp. 249–257.

[10] ———, *Generalized LR parsing*, Kluwer Academic Pub, 1991.

[11] Eelco Visser, *Scannerless generalized-LR parsing*, Tech. report, Citeseer, 1997.

[12] Philip Wadler, *How to replace failure by a list of successes*, Proc. of a conference on Functional programming languages and computer architecture (New York, NY, USA), Springer-Verlag New York, Inc., 1985, pp. 113–128.

```
sealed trait Parser[+A] extends (Stream[Char] => Result[A]) {
  ...

  def ~[B](that: Parser[B]): Parser[A ~ B]
}

trait TerminalParser[+A] extends Parser[A] {
  final def chain(t: Trampoline, in: Stream[Char])(f: Result[A] => Unit) {
    f(apply(in))
  }

  def ~[B](that: Parser[B]) = ...
}

trait NonTerminalParser[+A] extends Parser[A] {
  final def apply(in: Stream[Char]): List[Result[A]] = { ... }

  def ~[B](that: Parser[B]) = ...
}

class LiteralParser(str: String) extends TerminalParser[String] {
  def apply(in: Stream[Char]) = {
    val trunc = in take str.length
    lazy val errorMessage = "Expected '%s' got '%s'".format(str,
        canonicalize(trunc.mkString))

    if (trunc.lengthCompare(str.length) != 0) {
      Failure("Unexpected end of stream (expected '%s')".format(str), in)
    } else {
      val succ = trunc.zipWithIndex forall {
        case (c, i) => c == str.charAt(i)
      }

      if (succ)
        Success(str, in drop str.length)
      else
        Failure(errorMessage, in)
    } :: Nil
  }
}
```

Listing 3: `LiteralParser` and Friends

```
import edu.uwm.cs.gll._

object Main {
  import Parsers._

  def time(body: =>Unit): TimingInfo = {
    for (_ <- 1 to 3) {   // warm-up
      body
    }

    val results = (1 to 12) map { _ =>
      val start = System.currentTimeMillis
      body
      System.currentTimeMillis - start
    } toList

    val trimmed = results sort { _ < _ } drop 1 take 10
    TimingInfo((0L /: trimmed) { _ + _ } / 10)
  }

  def main(args: Array[String]) {
    lazy val s: Parser[String] = (
        s ~ s ~ s ^^ { _ + _ + _ }
      | s ~ s     ^^ { _ + _ }
      | "a"
    )

    for (_ <- 1 to 50) {    // warmup
      s("aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa")
    }

    for (num <- 10 to 100 by 2) {
      print("Parsing length " + num + "...")
      System.out.flush()

      val input = LineStream((1 to num).foldLeft("") { (str, i) => str + 'a' })
      val TimingInfo(avg) = time { s(input) }

      println(avg + "ms")
    }
  }

  case class TimingInfo(avg: Long)
}
```

Listing 4: Performance Test Harness