Simple, functional, sound and complete parsing for all context-free grammars

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

Abstract

Parsing text to identify grammatical structure is a common task, especially in relation to programming languages and associated tools such as compilers. Parsers for context-free grammars can be implemented directly and naturally in a functional style known as "combinator parsing", using recursion following the structure of the grammar rules. However, naive implementations fail to terminate on left-recursive grammars, and despite extensive research, the only complete parsers for general context-free grammars are constructed using other techniques such as Earley parsing.

Our main contribution is to show how to construct simple, sound and complete parser implementations directly from grammar specifications for all context-free grammars based on combinator parsing. We first develop a solution to handle a restricted class of grammars with left recursion based on the idea of *commitments*. We formalize the concepts involved in order to treat the general case, and use König's infinite path lemma on trees to characterize non-terminating parse attempts. Unfortunately this characterization is not effective. An effective over-approximation exists, but this eliminates some finite parses that a priori are valid. Fortunately these parses are redundant, and eliminating them preserves completeness: any input for which a parse tree can be constructed will be parsed with our approach. We then define a parser generator based on our ideas and prove it correct.

The focus of this work is on correctness, in particular completeness. In terms of efficiency, following [Norvig 1991] memoized combinator parsing is polynomial time for non-left-recursive grammars. Our approach can handle arbitrary context-free grammars, but because we insist on completeness, some highly-ambiguous left-recursive grammars cause our approach to return an exponential number of non-redundant parse trees in exponential time.

Categories and Subject Descriptors D.1.1 [*Applicative (Functional) Programming*]; F.4.2 [*Grammars and Other Rewriting Systems*]: Parsing; I.2.7 [*Natural Language Processing*]: Parsing

General Terms Parsing, Functional programming, Combinator parsing, Context-free grammar

Keywords functional, terminating, sound, complete, parsing, context-free

1. Introduction

Parsing is central to many areas of computer science, including databases (database query languages), programming languages (syntax), network protocols (packet formats), the internet (transfer protocols and markup languages), and natural language processing. Context-free grammars are typically specified using Backus-Naur Form (BNF), an example of which is[1]

```
E -> "(" E "+" E ")" | "1"
```

Informally, the symbol `E` can be replaced by the sequence consisting of the terminal `"("`, an `E`, the terminal `"+"`, another `E` and the terminal `")"`; alternatively the symbol `E` can be replaced by the terminal `"1"`. If we view `->` as a relation between lists of symbols we can construct the transitive closure of the `->` relation which we write as `->+`. For example

```
E ->+ "(" "1" "+" "1" ")"
```

If we concatenate the resulting terminals we get a string `(1+1)` which is accepted by the grammar. A parser is sound if it only recognizes strings accepted by the grammar. It is complete if it recognizes all such strings.

In combinator parsing sequencing and alternation are implemented using the infix combinators `**>` and `|||` (higher-order functions that take parsers as input and produce parsers as output). For example[2]

```
let rec E = fun i ->
  ((a "(") **> E **> (a "+") **> E **> (a ")")
  ||| (a "1")) i
```

The implementation works by first consuming a `"("` character from the input, then calling itself recursively to parse an `E`, then consuming a `"+"` character, and so on. Termination is obvious, because any recursive call to `E` is given strictly less input to parse.

Combinator parsing cannot be used directly if the grammar contains rules such as `E -> E "+" E` that are left recursive. A naive implementation of this rule would attempt to parse an `E` by first expanding to `E "+" E`, and then recursively attempting to parse an `E` on the same input, leading to non-termination.

> Although it is highly modular, general top-down parsing is often ignored as it has been traditionally categorized as expensive, and non-terminating while processing left-recursive grammars. [Hafiz and Frost 2010]

In practice alternative approaches are used. One approach is to change the implementation. Earley parsing [Earley 1970] and other approaches based on dynamic programming can be used to implement any context-free grammar. Implementations are usually complex, hard to understand, hard to modify, and hard to maintain. For example, sixteen years after Earley parsing was introduced [Tomita 1986] noted that the technique was incorrect for rules containing the epsilon or empty terminal `""`. This problem was eventually fixed in [Aycock and Horspool 2002] but the cost in terms of further implementation complexity is substantial.

[1] Real BNF requires nonterminals to be written eg `<E>`.

[2] The examples in this section are essentially the same as the OCaml examples described later, but are formally pseudocode because eg names of OCaml functions must start with a lowercase letter.

Another approach often combined with an alternative implementation is to limit the grammar, for example by ruling out left-recursive grammars, or ambiguous grammars. Much more draconian restrictions are typically enforced in practice, and the resulting grammar classes go by such names as LL, LR, LR(i), SLR, LALR. Unfortunately the specifications of these restricted classes are often complicated and rely on details of the underlying implementations.

The current situation is that if your grammar fits into some known restricted class and there is a reasonable parser generator for that class (and for your desired implementation language) then implementing a parser may be straightforward. Otherwise you must typically modify your initial grammar specification so that it fits one of these classes. Such a process is error-prone and introduces unwanted complexity into the grammar specification. If the parser computes some function of the parse tree, rather than simply recognising input, grammar modification may be inappropriate [Frost and Hafiz 2006].

In practice, parser generators reject grammars with incomprehensible messages detailing some failed implementation step eg "a shift-shift conflict has been detected". The generated parsers reject input with similarly mystic pronouncements. Even if the user has been trained to accept these complications as necessary, the process of producing a parser is often a frustrating one.

In light of these issues, the benefits of combinator parsing seem even more appealing: the process of implementation is straightforward; the resulting implementation is closely related to the grammar specification, which helps ensure soundness; and the implementation is clean, concise, and comprehensible, which eases modification and maintenance. Therefore a longstanding goal is to adapt combinator parsing to handle all context-free grammars. This has lead to much related work, see Sect. 12. The work most closely related to ours is that of Frost et al. [Frost and Launchbury 1989; Frost 1992, 1994; Frost and Hafiz 2006; Frost et al. 2007, 2008; Hafiz and Frost 2010]. As recently as [Frost and Hafiz 2006], the approach was known to be incomplete.

> ... Frost and Hafiz (2006) ... does not accommodate indirect left recursion ... [Frost et al. 2007]

More recent work, starting from [Frost et al. 2007], tries to limit the recursive nesting of a parser for a nonterminal X to $|s|+1$ when applied to an input s. If there are m parsers corresponding to m nonterminals in the grammar, the maximum depth of recursion is then $m(|s|+1)$, and termination is straightforward. However, correctness of the approach is left for future work.

> Future work includes proof of correctness ... [Frost et al. 2007]
>
> We are constructing formal correctness proofs ... [Hafiz and Frost 2010]

This suggests that the authors do not view correctness, in particular completeness, as an immediate consequence of their approach. In fact [Frost et al. 2007] seems to state explicitly that the approach is incomplete, at least for "circular grammars".

> At this point no parse is possible (other than spurious parses which could occur with circular grammars – which we want to reject) ... the grammar is circular (`N` is being rewritten to `N`) ... [Frost et al. 2007]

However their approach actually *is* complete for arbitrary context-free grammars. Given a substring s, the worst that can happen with *our* approach is that a parser for a nonterminal X is given a substring of length $|s|$ (the initial input) and recursive calls are then given an input of length $|s|-1, |s|-2, \ldots, 0$. Clearly the maximum nesting for a parser for nonterminal X is $|s|+1$, so for a grammar with m nonterminals recursion is bounded by $m(|s|+1)$. Since our approach is complete, a simple corollary is that *any* approach that returns all parse trees where the maximum nesting depth of a parser for a nonterminal X is $|s|+1$ will be complete *for arbitrary context-free grammars* (and this includes the recent work of Frost et al.). This is a minor contribution of our work.

The main contribution of our work is to show how to implement simple, terminating, sound and complete parsers for arbitrary context-free grammars using combinator parsing. Based on this we define a parser generator for arbitrary context-free grammars and prove it correct.

Commitment-based parsing Commitment-based parsing is simpler than the general case discussed in the rest of the paper, and has some merit as an independent implementation technique. The main reason we present it here is that the underlying intuition is the basis for the solution in the general case.

Consider the following BNF and its implementation.

```
(* E -> E "+" E | "1"; faulty implementation *)
let rec E = fun i ->
  ((E **> (a "+") **> E) ||| (a "1")) i
```

The recursive call to `E` may be given exactly the same input as the parent, which leads to an infinite chain of recursive calls, or stack overflow. However, when we attempt to parse an `E` using the rule `E -> E "+" E` clearly *something* is decreasing: we *commit* to parsing a `+` from the input *after* the first recursive call to `E`. In particular, we know that the first `E` should not consume the entire input. An obvious way to ensure this is to give the first `E` only a part of the input. We introduce a parser transformer (a function from parsers to parsers) `ignr_last` that ignores the last character of the input when invoking the underlying parser, see Sect. 5 for details.

```
(* E -> E "+" E | "1" *)
let rec E = fun i ->
  (((ignr_last E) **> (a "+") **> E) ||| (a "1")) i
```

This parser is terminating, sound and complete for the left-recursive grammar `E -> E "+" E | "1"`.

Analysis Rules are of the form X `->` $Y \ldots Z$, where X is a nonterminal and $Y \ldots Z$ are either nonterminals or terminals. We let Greek letters $\alpha, \beta, \ldots$ range over sequences of terminals or nonterminals. For the rule X `->` $\alpha\ X\ \beta$ there are several cases:

- Not α `->+ "" ... ""`. By the time the recursive call to parse X is made, part of the input will have been consumed whilst parsing α, so the recursive call will receive strictly less input than the parent, thereby ensuring termination.
- α `->+ "" ... ""`, but not β `->+ "" ... ""`. Then we can use commitment-based parsing, and wrap the parser for $\alpha\ X$ in `ignr_last` as described above. The input is reduced when attempting to parse αX, in particular, the input is reduced by the time the recursive call to parse X is made.
- α `->+ "" ... ""`, and β `->+ "" ... ""`, ie X `->+ "" ... ""` X `"" ... ""`, which is the problematic general case.

Our example rule involves direct recursion (the rule for X involves X on the right-hand side) but the problems are similar for indirect recursion eg X `->` $\alpha\ A\ \beta$ and A `->+` $\alpha'\ X\ \beta'$.

Structure of the paper In Sect. 2 we discuss notation. In Sect. 3 we give a brief introduction to substrings, which are used extensively in the rest of the paper. In Sect. 4 we give formal definitions for concepts such as terminal, nonterminal, parse rule, grammar and parse tree. In Sect. 5 we give the full implementation of commitment-based parsing, which usefully introduces much of the machinery used in later sections. Commitment-based parsing is an independent technique that is simpler than the general solution, so

represents a minor contribution of this paper. In Sect. 6 we examine the case of general context-free grammars. We characterize infinite parse trees, define an effective approximation based on "bad" nodes, and show that removing parse trees with bad nodes preserves completeness. In Sect. 7 we briefly discuss an implementation strategy based on checking for bad nodes and in Sect. 8 we give an implementation based on these ideas. This requires extending the input to include a context, and wrapping potentially looping parsers in a function `check_and_upd_ctxt`. For example, the following is the implementation of a terminating, sound and complete parser for the highly ambiguous grammar `E -> E E | "1" | ""`.

```
let rec E = fun i ->
  check_and_upd_ctxt "E"
    ((E **> E) ||| (a "1") ||| (a ""))
    i
```

Our approach retains the simplicity and elegance of traditional combinator parsing whilst extending it to treat all context-free grammars. In Sect. 9 we implement a parser generator, capable of taking any context-free grammar and producing a terminating, sound and complete parser for it. This allows us to state a general correctness theorem concerning our approach. In Sect. 10 we discuss memoization. In Sect. 11 we discuss mechanization of the implementations and proofs in a theorem prover. There is a great amount of related work which we discuss in Sect. 12. Finally we draw some conclusions in Sect. 13.

Our implementation language is OCaml [Leroy et al.]. The complete OCaml code can be downloaded from the author's homepage[3].

2. Notation

Throughout we mix mathematical definitions and notation with real OCaml code excerpts formatted using typewriter font. We also use typewriter font for concrete grammars, nonterminals and terminals. For example, a concrete nonterminal may be written `X`, whilst a variable ranging over nonterminals may be written X. Concrete strings are eg `hello`. The length of a string s is $|s|$. Terminals are eg `"1"`. Nonterminals are eg `E`.

We write function application as $f(i)$ or $f\ i$ or f_i.

OCaml syntax should be comprehensible to anyone with a knowledge of a similar functional programming language such as SML or Haskell. Boolean conjunction is `&&`. Lists are written `[x;y;z]` or `x::y::z::[]`. Anonymous functions $\lambda x \ldots$ are written `fun x -> ...`. Non-recursive bindings are eg `let x = ...`. General recursive function definitions are written eg `let rec f = fun x -> ...`. Standard operations are `fst,snd` to return the first or second component of a pair; `List.map` to map a function over a list; `List.concat` to form a list from a list of lists by concatenating them together; `List.append` to append two lists; `List.exists p xs` which returns `true` iff the list `xs` contains an element satisfying property `p`; `List.filter p xs` which returns the elements in `xs` satisfying the property `p`; `String.length` to return the length of a string; `String.sub s l n` to return the substring of `s` from position `l` with length `n`; `s ^ t` to return the string formed by appending the strings `s` and `t`. In OCaml `(* this is a comment *)`.

3. Preliminaries

We need an infix function composition operator.

```
let ($) f g x = f(g x)
```

[3] `http://www.cs.le.ac.uk/~tr61/parsing`

Note that in OCaml, infix operators are defined using prefix form, where the operator is surrounded by brackets. We make extensive use of substrings ie a triple (s, l, h) of a string s, with low index l and high index h, satisfying the invariant $l \leq h \leq |s|$.

```
type substring = string * int * int

let string (s,l,h) = s

let [low;high;len] = [
  (fun (s,l,h) -> l);
  (fun (s,l,h) -> h);
  (fun (s,l,h) -> h-l)]

let full s = (s,0,String.length s)

let inc_low n (s,l,h) = (s,l+n,h)
let dec_high n (s,l,h) = (s,l,h-n)
let inc_high n (s,l,h) = (s,l,h+n)

let content s =
  String.sub (string s) (low s) (len s)
```

Definition 1 (contiguous, concatenate, span)**.** *A pair of two substrings (s, l_1, h_1) and (s, l_2, h_2) is contiguous iff $l_2 = h_1$, in which case the substrings can be concatenated to form the substring (s, l_1, h_2). A sequence of substrings $(s, l_0, h_0), \ldots, (s, l_n, h_n)$ is contiguous iff each pair of adjacent substrings is contiguous, in which case it can be concatenated to form the substring (s, l_0, h_n). A sequence $(s, l_0, h_0), \ldots, (s, l_n, h_n)$ of substrings spans a substring t iff they can be concatenated to form t.*

Typical parser implementations will attempt to parse the longest prefix of the input.

Definition 2 (contains, prefix, suffix)**.** *A substring (s, l_1, h_1) contains a substring (s, l_2, h_2) iff $l_1 \leq l_2 \leq h_2 \leq h_1$. If in addition $l_2 = l_1$ we say that (s, l_2, h_2) is a prefix of (s, l_1, h_1). Similarly if $h_2 = h_1$ we say that (s, l_2, h_2) is a suffix of (s, l_1, h_1).*

4. Basic definitions

Definition 3 (symbol, terminal, nonterminal)**.** *The set of symbols is the disjoint union of the set of terminals, the set of nonterminals and the set of singletons. A singleton is a set containing a single substring. For every terminal X there is a set of substrings $\mathcal{S}_X$.*

Terminals and nonterminals are just elements of some uninterpreted sets. A terminal is implemented by some terminating OCaml function, the details of which are mathematically irrelevant; what matters is the set of substrings $\mathcal{S}_X$ that the implementation of terminal X accepts. A singleton $\{(s, l, h)\}$ is used to indicate exactly how a terminal matched a given portion of the input s.

For an implementation, we must pick some underlying types for terminals and nonterminals.

```
type term = string
type nonterm = string
type singleton = substring

type symbol = TM of term
  | NT of nonterm | SN of singleton
```

An example of a parse rule is `E -> E "+" E`. The right-hand side may list several alternatives eg `E -> E "+" E | "1"` which is equivalent to the two rules `E -> E "+" E` and `E -> "1"`. Informally when we talk of a rule `X ->` $X_0 \ldots X_i$, we mean a rule with X on the left-hand side, and $X_0 \ldots X_i$ as one of the alternatives.

Definition 4 (alternative, parse rule)**.** *An alternative is a list $[X_0, \ldots, X_i]$ of symbols such that each X_j is either a terminal or a nonterminal. A nonterminal parse rule is a pair (X, Z) of a nonterminal X and a list of alternatives Z. A terminal parse rule is a pair $(X, [[\{x\}]])$, where X is a terminal and $\{x\}$ is a singleton, such that $x \in \mathcal{S}_X$.*

In implementations we only need nonterminal parse rules: terminal parse rules are a mathematical abstraction used to model the behaviour of opaque terminal parsers. Note that the following type is imprecise as it allows singletons on the right-hand side.

```
type parse_rule = nonterm * ((symbol list) list)
```

Definition 5 (grammar)**.** *A grammar G is a set of parse rules formed from a finite set of nonterminal parse rules by adding all terminal parse rules.*

At the implementation level there are no terminal parse rules, so we identify a grammar with a finite set of parse rules represented as a list.

```
type grammar = parse_rule list
```

Trees are used to describe successful parses. We also use trees to describe parse attempts that result in looping behaviour by a hypothetical implementation. For this reason, our definition of tree includes infinite trees.

Definition 6 (sequence, tree)**.** *A sequence is a finite list of natural numbers. A tree T is a set of sequences such that any prefix of a sequence in T is also in T.*

Although this definition is very concrete, it has the benefit of being extremely precise. We reassure the reader that none of what follows depends essentially on the details of this definition. In the following, we work only with trees that are finitely branching. We refer to sequences $x \in T$ as nodes of T. The root node of T is the empty sequence $[]$.

Definition 7 (parent, child, descendant, ascendant)**.** *If x is a sequence $x_0, \ldots, x_i$ in T, and $y = x_0, \ldots, x_i, y_{i+1} \in T$, then x is a parent of y, and y is a child of x. The transitive closure of the child relation is called descendant. The transitive closure of the parent relation is called ascendant.*

Definition 8 (leaf)**.** *A sequence x in T is a leaf iff x is not a parent.*

Definition 9 (leaf order)**.** *The leaf order is simply the lexicographic order on the underlying sequences (restricted to those sequences that are leaves). Sequences of differing lengths are compared in the standard way: the longer sequence is first truncated to the length of the shorter.*

For two distinct leaves x and y, either $x < y$ or $y < x$ in the leaf order.

Definition 10 (decorated tree)**.** *A decorated tree is a pair (T, f) of a tree T and a function f with domain T.*

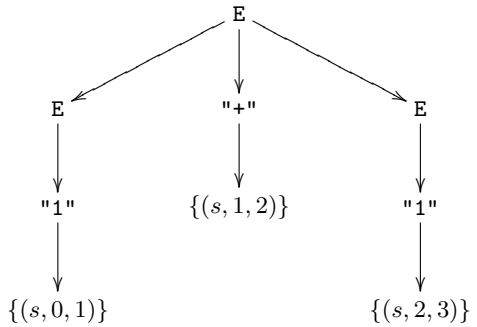
Definition 11 (subtree under x)**.** *Given a tree T, and a node $x \in T$, to form the subtree T^x under $x = x_0, \ldots, x_i$, first form the set U of sequences consisting of x and all descendants of x, then remove the common prefix $x_0, \ldots, x_i$ from each sequence to give the tree T^x. To form a decorated subtree (T^x, f^x) of a decorated tree (T, f) let $f^x = \lambda(y_0, \ldots, y_j).f(x_0, \ldots, x_i, y_0, \ldots, y_j)$*

Informally, given a grammar G, a parse tree for some initial input is a tree constructed according to the rules of G.

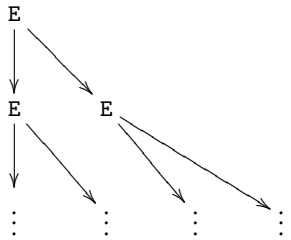
Definition 12 (parse tree)**.** *Given a grammar G and a substring (s, l, h), a parse tree is a decorated tree (T, X) consisting of a tree T and a function X from nodes x to symbols X_x. For each parent $p \in T$ with children $i \ldots j$, there must be a rule $r \in G$ of the form X_p `->` $X_i \ldots X_j$. For each leaf x, X_x must be a singleton $\{(s, l_x, h_x)\}$.*

In this definition, the tree T may be infinite. Singletons only appear at leaves, and X_x is a singleton for all leaves x. If x is a leaf with $X_x = \{(s, l_x, h_x)\}$, and y is the parent of x, then by the nature of the rules in G, X_y must be a terminal with $(s, l_x, h_x) \in \mathcal{S}_{X_y}$. Furthermore, since there are only a finite number of nonterminal parse rules in G, there are only a finite number of nonterminals X_x, that is, the range of X, restricted to nonterminals, is finite.

Let s be the string `1+1`. An example of a finite parse tree for $(s, 0, 3)$ is



In the previous example, nodes x are annotated with symbols X_x. An example of an infinite parse tree for the grammar `E -> E E | "1" | ""` is



Some parse trees represent completed parses.

Definition 13 (final parse tree)**.** *Given a substring s, a parse tree (T, X) is final iff*

- *T is finite*
- *if the singletons in leaf order are $\{s_0\}, \ldots, \{s_i\}$ then the substring sequence $s_0, \ldots, s_i$ spans s.*

The example finite parse tree is final. Note that a final parse tree gives precise information about how the input was parsed. For a final parse tree (T, X) and a node $x \in T$, the subtree (T^x, X^x) is also final for the substring formed from concatenating the singletons at the leaves.

Definition 14 (parsed)**.** *Given a grammar G, a substring s can be parsed as a terminal or nonterminal Y iff there is a final parse tree (T, X) for s such that $X_{[]} = Y$ ie the root of the tree is decorated with nonterminal Y*

For the grammar formed from the nonterminal rule `E -> E "+" E | 1`, the example finite parse tree shows that the substring $(s, 0, 3)$ can be parsed as an `E`.

Mathematically a parse tree may be infinite, but at the implementation level, parse trees are finite.

```
type parse_tree = PT of symbol * (parse_tree list)

let leaf tm s = PT(TM tm,[PT(SN s,[])])
```

5. Commitment-based parsing

In Sect. 1 we introduced commitment-based parsing, which allows traditional combinator parsing to handle a wide class of left-recursive grammars. Commitment-based parsing does not need a context, which makes it simpler than the general case discussed in the next section. The central idea of commitment-based parsing is to wrap parsers in the parser transformer `ignr_last`, which reduces the input when calling the underlying parser, thereby ensuring termination. The purpose of this section is to describe commitment-based parsing in detail, and to introduce much of the machinery that will be used in the following sections.

Traditionally a parser is a function from an input string s to a result, that is, a pair (v, s') of a value (here, a parse tree) and the remainder of the string that failed to parse. If a grammar is ambiguous, then multiple results may be returned in a list. Later, we refine the input type, but for the moment we identify it with the type of substrings. The functions `substr` and `toinput` convert from inputs to substrings and vice versa; here they are trivial. The function `lift` converts a function on substrings to a function on inputs.

```
type input = substring

let id = fun x -> x
let (substr,toinput,lift) = (id,id,id)

type 'a parser = input -> ('a * substring) list
```

A basic requirement is that a parser should be sound. For the following definition, note that a parser that returns no results is sound, as is a parser that fails to return at all.

Definition 15 (sound parser)**.** *Given a grammar G and a substring s, a parser is sound for terminal or nonterminal Y iff, if it returns, then it returns only (implementations of) final parse trees (T, X) with $X_{[]} = Y$.*

Ideally a parser should also be complete, that is, if there is a final parse tree for a substring s, then the parser should return at least one such parse tree.

Definition 16 (complete parser)**.** *Given a grammar G and a substring s, a parser is complete for a terminal or nonterminal Y iff, if s can be parsed as a Y, then the parser returns at least one (implementation of a) final parse tree (T, X) for s with $X_{[]} = Y$.*

Note that completeness does not require that all final parse trees for s are returned – there may be an infinite number of them. Note also that the definitions of soundness and completeness are not convenient for real implementations which typically return pairs of a final parse trees for a *prefix* of the initial input, together with the remainder of the input that was not parsed.

Definition 17 (prefix-sound parser)**.** *Given a grammar G and a substring (s, l, h), a parser is prefix-sound for terminal or nonterminal Y iff it returns only pairs $((T, X), (s, l', h)))$ such that*

- *(T, X) is a final parse tree for the prefix (s, l, l') of (s, l, h)*
- *$X_{[]} = Y$*
- *(s, l', h) is a suffix of (s, l, h)*

Definition 18 (prefix-complete parser)**.** *Given a grammar G and a substring (s, l, h), a parser is prefix-complete for a terminal or nonterminal Y iff, for any prefix (s, l, l') of (s, l, h) that can be parsed as a Y, the parser returns at least one pair $((T, X), (s, l', h))$ such that*

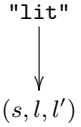
- *(T, X) is a final parse tree for the prefix (s, l, l') of (s, l, h)*
- *$X_{[]} = Y$*
- *(s, l', h) is a suffix of (s, l, h)*

We define $\mathcal{S}_{\texttt{"lit"}}$ to be substrings whose content is the string `lit`. Given the following definition, eg the terminal `"1"` would be implemented as `(a "1")` in OCaml.

```
(* string -> parse_tree parser *)
let a lit = fun i ->
  let n = String.length lit in
  let s = substr i in
  if
    (n <= len s)
    && (String.sub (string s) (low s) n = lit)
  then
    let (s1,l,h) = s in
    let s2 = (s1,l,l+n) in
    let r = leaf ("\"" ^ lit ^ "\"") s2 in
    [(r,inc_low n s)]
  else
    []
```

Lemma 19. *For any grammar G, the parser* `(a "lit")` *is a terminating, prefix-sound and prefix-complete parser for the terminal* `"lit"`.

Proof. Termination is clear. A final parse tree (T, X) for the terminal `"lit"` has the following form



where the content of (s, l, l') is the string `lit`. Assume (s, l, l') is a prefix of (s, l, h), so that (s, l', h) represents the remainder of (s, l, h) that is not parsed. For prefix-soundness, it is clear that given any input (s, l, h) the parser only returns final parse trees of this form. For prefix-completeness, for all final parse trees (T, X) of this form, the parser when called on input (s, l, h) returns (the implementation of) (T, X) paired with (s, l', h). □

Prefix-soundness and prefix-completeness are motivated by implementation concerns, and detract from the mathematical presentation. In what follows, we informally use the terms soundness and completeness to mean prefix-soundness and prefix-completeness, but we elide details of the proofs that manage the unmemorable book-keeping involved in dealing with prefixes.

We need two combinators, representing sequencing (parser 1 then parser 2) and alternation (parser 1 or parser 2, equivalently, several rules for the same nonterminal).

```
(* 'a parser -> 'b parser -> ('a * 'b) parser *)
let ( **> ) p1 p2 = fun s ->
  let f (e1,s1) =
    List.map (fun (e2,s2) -> ((e1,e2),s2)) (p2 s1)
  in
  (List.concat $ (List.map f) $ p1) s

(* 'a parser -> 'a parser -> 'a parser *)
let (|||) p1 p2 = fun s -> List.append (p1 s) (p2 s)
```

Our example BNF directly translates to the following parser.

```
(* E -> "(" E "+" E ")" | "1" *)
let rec pE = fun i ->
  let p1 =
    (a "(") **> pE **> (a "+") **> pE **> (a ")")
  in
```

```
  let f = fun (bra,(e1,(plus,(e2,ket)))) ->
    PT(NT "E",[bra;e1;plus;e2;ket])
  in
  ((p1 >> f) ||| (a "1")) i
```

Note that the `**>` operator associates to the right. The previous definition makes use of the `>>` value transformer, which takes a parser `p` and a function `f` and applies `f` to the values returned by `p`.

```
(* 'a parser -> ('a -> 'b) -> 'b parser *)
let (>>) p f =
  (List.map (fun (e,s) -> (f e, s))) $ p
```

Commitment-based parsing uses a parser transformer that drops the last character of the input before invoking the underlying parser, then adds that character back when returning the results.

```
(* 'a parser -> 'a parser *)
let ignr_last p = fun i ->
  if len (substr i) = 0 then [] else
  let inc_high (e,s) = (e,inc_high 1 s) in
  ((List.map inc_high) $ p $ (lift (dec_high 1))) i
```

The example commitment-based parser from Sect. 1 is then:

```
(* E -> E "+" E | "1" *)
let rec pE = fun i ->
  let p1 = (ignr_last pE) **> (a "+") **> pE in
  let f = fun (e1,(plus,e2)) ->
    PT(NT "E",[e1;plus;e2])
  in
  ((p1 >> f) ||| (a "1")) i
```

For which we have the following

Lemma 20. *The example parser is a terminating, sound and complete parser for the grammar* `E -> E "+" E | "1"`.

Proof. For termination, calls to `(a "1")` and `(a "+")` are terminating by Lemma 19, and recursive calls to `pE` are given strictly smaller substrings than the parent.

For soundness, when given a substring s as an argument, the parser returns only terminal parse trees for s, by induction on the length of the substring, again using Lemma 19 for the base cases.

Completeness is immediate from the stronger property that, given a substring s, the example parser returns every possible final parse tree (T, X) for s, which is proved by induction on (T, X), again using Lemma 19 for the base cases. □

We end this section by generalizing the two combinators from pairs of parsers to lists of parsers.

```
let always = fun i -> [([],substr i)]

let never = fun i -> []

let rec then_list ps = match ps with
| [] -> always
| p::ps -> (p **> (then_list ps))
    >> (fun (x,xs) -> x::xs)

let rec or_list ps = match ps with
| [] -> never
| p::ps -> (p ||| (or_list ps))
```

6. The general case

The central result of this section is Thm. 22 which gives a characterization of infinite parse trees. Informally this theorem describes how any depth-first parser might fail to terminate. We use this theorem to derive an effective test for potential non-termination. In the next section we describe how this can form the basis of sound and complete parsers for arbitrary context-free grammars.

We first define a matched parse tree, which is a parse tree with additional information at each node about what substring was parsed by the subtree at that node.

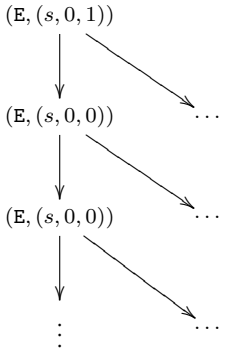
Definition 21 (matched parse tree)**.** *A matched parse tree is a triple (T, X, S) where (T, X) is a parse tree, and (T, S) is a decorated tree such that*

- *for every leaf x with singleton $X_x = \{(s, l_x, h_x)\}$ we have $S_x = (s, l_x, h_x)$*
- *for every parent p with children $i \ldots j$, the substrings $S_i \ldots S_j$ are contiguous, and their concatenation gives S_p.*

Theorem 22 (infinite-characterization)**.** *For all infinite matched parse trees (T, X, S), there exists a natural number k, and an infinite branch $x_0, x_1, \ldots, x_k, \ldots$ starting at the root such that the sequence $S_{x_k}, S_{x_{k+1}}, \ldots$ is constant.*

Proof. The infinite parse (T, X) is finitely branching since the right-hand side of each rule in G is finite, so by König's lemma there exists an infinite branch of nodes x_i decorated with substrings S_{x_i}, starting from the root. The low index of each substring increases along this branch, and is bounded above. The high index of each substring decreases along the branch, and is bounded below. Thus, eventually the low index and high index of each substring is constant, say, from x_k. Later nodes are decorated with the same substring (s, l, h). □

The following shows an infinite path in an infinite matched parse tree for the grammar `E -> E E | "1" | ""`. Nodes x have been annotated with pairs (X_x, S_x).

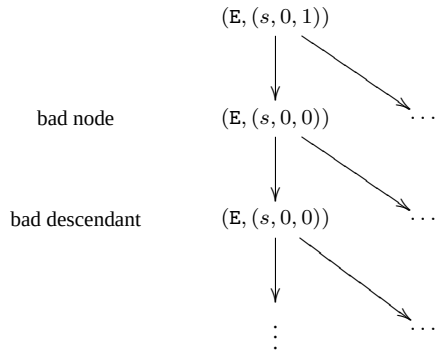


Corollary 23. *For all infinite matched parse trees (T, X, S), there exists an infinite branch $x_0, x_1, \ldots$ starting at the root such that S and X are constant on some infinite subsequence of $x_0, x_1, \ldots$.*

Proof. From the previous theorem, there exists k such that S is constant on the sequence $x_k, x_{k+1}, \ldots$. Clearly for any x in this sequence, X_x is a nonterminal. Only finitely many nonterminals can appear in a parse tree, so some nonterminal must appear infinitely often in the sequence $X_{x_k}, X_{x_{k+1}}, \ldots$ □

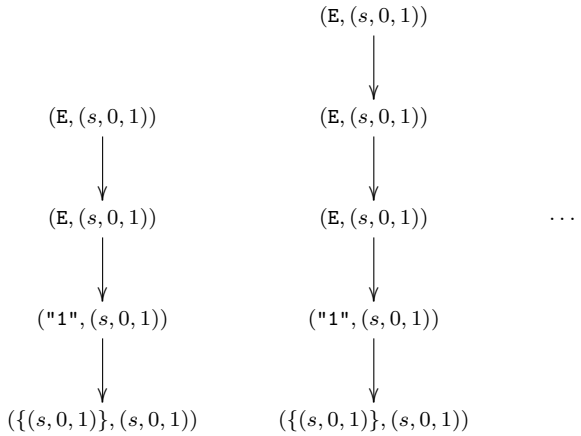
As a further corollary, if there is only one infinite branch in a parse tree then indeed we must have a rule X `->+ "" ... ""` X `"" ... ""`. This confirms the analysis from Sect. 1. It also strongly suggests the following definition.

Definition 24 (bad node, bad descendant)**.** *Given a matched parse tree (T, X, S), a bad node is a node $x \in T$, with a descendant $x' \in T$, such that $X_{x'} = X_x$ and $S_{x'} = S_x$. In this case, we say that x' is a bad descendant.*



Lemma 25 (infinite-bad)**.** *Every infinite matched parse tree contains infinitely many bad nodes.*

Of course, some perfectly valid final matched parse trees may contain bad nodes as well. For example, if s is a string of length one containing `1`, then the rule `E -> E | "1"` clearly gives rise to an infinite number of final parse trees containing bad nodes.



Lemma 26 (exist-final-bad)**.** *There exist final matched parse trees with bad nodes.*

However, final matched parse trees with bad nodes can be pruned from a set of final matched parse trees without affecting completeness. We now make precise exactly what we are giving up.

Lemma 27 (bad-node-removal)**.** *For each final matched parse tree (T, X, S) containing bad nodes, there is another final matched parse tree (T', X', S') containing strictly less bad nodes such that $X'_{[]} = X_{[]}$ and $S'_{[]} = S_{[]}$.*

Proof. A bad node x, with bad descendant x', can be transformed to remove the bad node: just replace the subtree T^x with the subtree $T^{x'}$, and adjust X and S in the obvious way. This does not affect the value of S and X at x, nor at ascendants of x. In particular, $X_{[]}$ and $S_{[]}$ are preserved. □

Clearly we can repeat this process to eliminate all bad nodes from a final matched parse tree. Even though avoiding bad nodes eliminates some parse trees, it still preserves completeness.

Theorem 28 (no-bad-complete)**.** *If a substring s can be parsed as a nonterminal X by a final matched parse tree, then it can be parsed as an X by a final matched parse tree containing no bad nodes.*

A consequence of this theorem is that when reasoning about completeness, we need only consider final matched parse trees with no bad nodes.

7. Parsing strategy

Given the preceding results, the obvious strategy is to check for bad nodes while parsing. When attempting to parse some substring (s, l, h) as a nonterminal X, the information $(X, (s, l, h))$ should be recorded in a context before making recursive calls to other parsers. A recursive call to parse some input (s', l', h') as a nonterminal X' should first examine the context. If $(X', (s', l', h'))$ is already in the context, then some ascendant call is attempting to parse the same input as the same nonterminal. A successful parse of the entire substring, followed by a successful parse of the same substring by the ascendant, results in a bad node. To avoid a bad node, the parse attempt should be abandoned.

The above reasoning assumes we are trying to parse an entire substring. However, with left-to-right parsing, it is more common to attempt to parse some prefix of the substring. The presence of $(X, (s, l, h))$ in the context is then taken to mean that some ascendant is attempting to parse a prefix of (s, l, h) as a nonterminal X. A successful parse of a prefix of (s, l, h) by the descendant does not necessarily result in a bad node, because the ascendant might parse a strictly longer prefix. However, we should avoid a successful parse of the entire substring, since in that case a successful parse by the ascendant *will* result in a bad node. Thus, if we ignore the last character, and attempt to parse $(s, l, h-1)$ we know

- that we are omitting only parse trees with bad nodes
- that we ensure termination, since recursive calls to parse X are given strictly smaller substrings

8. Implementation

We illustrate the implementation strategy outlined in the previous section. We introduce a context representing the parent recursive calls. For each ascendant, we record the substring given as input, and what nonterminal was being parsed.

```
type context = (nonterm * substring) list
```

The input type is now a pair of a context and a substring.

```
type input = context * substring

let substr (c,s) = s
let toinput s = ([],s)
let lift f (c,s) = (c,f s)

type 'a parser = input -> ('a * substring) list
```

The remainder of the code is unchanged, except that we need to modify the `**>` combinator, to pass the context to the subparsers. Note that the context `c` is the same when `p1` is invoked as when `p2` is invoked.

```
(* 'a parser -> 'b parser -> ('a * 'b) parser *)
let ( **> ) p1 p2 = fun (c,s) ->
  let f (e1,s1) =
    List.map (fun (e2,s2) -> ((e1,e2),s2)) (p2 (c,s1))
  in
  (List.concat $ (List.map f) $ p1) (c,s)
```

Following the strategy of the previous section, we define a parametrised parser transformer to update the context. The extra parameter records the nonterminal that is being parsed.

```
(* nonterm -> 'a parser -> 'a parser *)
let update_ctxt nt p = fun (c,s) -> p ((nt,s)::c,s)
```

As well as updating the context, we need to check whether any ascendant call is attempting to parse the same substring as the same nonterminal, in which case we should trim the input by ignoring the last character in the substring when calling the underlying parser.

```
(* nonterm -> 'a parser -> 'a parser *)
let check_and_upd_ctxt nt p = fun (c,s) ->
  let should_trim = List.exists ((=) (nt,s)) c in
  if should_trim && (len s = 0) then
    []
  else if should_trim then
    (ignr_last (update_ctxt nt p)) (c,s)
  else
    (update_ctxt nt p) (c,s)
```

In general, to avoid non-termination when parsing a nonterminal X, simply wrap the body of the parser for X in a call to `check_and_upd_ctxt`. For example, the following is a terminating, sound and complete parser for the highly ambiguous grammar `E -> E E | "1" | ""`.

```
let rec pE = fun i -> check_and_upd_ctxt "E"
  (((pE **> pE) >> (fun (s,t) -> PT(NT "E",[s;t])))
   ||| (a "1")
   ||| (a ""))
  i
```

When attempting to prove correctness and in particular completeness, we need to show that a parser will produce some given parse tree. Whether it will or not depends on the context. The following definition characterizes those parse trees that may be returned by a parser when supplied with a given context.

Definition 29 (admits)**.** *Let (T, X, S) be a finite matched parse tree. Let $S_{[]} = (s, l, h)$ be the substring matched by the root. A context c admits (T, X, S) iff*

- *Ascendants are attempting to parse longer substrings ie if $(X_i, (s, l_i, h_i)) \in c$ then (s, l, h) is contained in (s, l_i, h_i).*
- *The substring (s, l, h) matched by the root is not bad ie it is not the case that $(X_{[]}, (s, l, h)) \in c$.*
- *Let c' be the extension of c by the pair $(X_{[]}, (s, l, h))$. If x_i is a child of the root then c' must admit the decorated subtree $(T^{x_i}, X^{x_i}, S^{x_i})$.*

Note that the initial empty context admits all finite parse trees with no bad nodes.

Lemma 30. *The implementation of the example parser is terminating, sound and complete for the grammar* `E -> E E | "1" | ""`.

Proof. For termination, it suffices to define a measure that decreases with each recursive call. Suppose the finite set of nonterminals in the grammar is $\{X_0, \ldots, X_i\}$. If the input is $(c, (s, l, h))$ then for each X_i the context c contains a pair $(X_i, (s, l_i, h_i))$ such that the length $m_i = h_i - l_i$ is minimal (if X_i does not appear in the context, take $m_i = |s| + 1$). The measure $\Sigma_i m_i$ decreases on each recursive call.

For soundness, from Lemma 19 the base parsers `(a "")` and `(a "1")` are sound. From termination, we induct on the length of execution to show that the example parser is sound, noting that `check_and_upd_ctxt "E" p` only removes parse trees that might be returned by the underlying parser `p`.

For completeness, from Theorem 28 it suffices to show that every finite matched parse tree with no bad nodes is returned by the example parser. The following is immediate by induction on T: for all finite matched parse trees (T, X, S) and all contexts c that admit (T, X, S), if $S_{[]} = (s, l, h)$ then the example parser when called with input $(c, (s, l, h))$ returns (the implementation of) (T, X). □

```
let parse_while pred = fun i ->
  let s = substr i in
  if len s = 0 then [] else
  let rec f = fun n ->
    if n = len s then len s else
    let c = String.sub (string s) ((low s)+n) 1 in
    if pred c then f (n+1) else n
  in
  let n = f 0 in
  let r = (string s, low s, (low s)+n) in
  [(r,inc_low n s)]

let not_epsilon p = fun i ->
  List.filter (fun (v,_) -> not (len v = 0)) (p i)

let parse_AZS =
  let pred c =
    (String.compare "A" c <= 0)
    && (String.compare c "Z" <= 0)
  in
  not_epsilon (parse_while pred)
```

Figure 1. Implementation of terminal parser `?AZS?`

Note that if the grammar contains m nonterminals, and the initial substring is $(s, 0, |s|)$, then the empty context has measure $m(|s| + 1)$.

9. A parser generator

In this section we use our approach to implement a parser generator, which takes an arbitrary context-free grammar and produces a terminating, sound and complete parser for that grammar. This allows us to state a general theorem concerning the correctness of our approach.

The following is a syntax for BNF. We have adopted two features from Extended BNF: nonterminals do not have to be written within angled brackets, and arbitrary terminals can be written within question marks. The terminal `?ws?` accepts non-empty strings of whitespace, `?notdquote?` (resp. `?notsquote?`) accepts strings of characters not containing a double (resp. single) quote character, `?AZS?` accepts non-empty strings of capital letters, and `?azAZs?` accepts non-empty strings of letters.

```
RULES -> RULE | RULE ?ws? RULES
RULE -> SYM ?ws? "->" ?ws? SYMSLIST
SYMSLIST -> SYMS | SYMS ?ws? "|" ?ws? SYMSLIST
SYMS -> SYM | SYM ?ws? SYMS
SYM -> '"' ?notdquote? '"' | "'" ?notsquote? "'"
       | ?AZS? | "?" ?azAZs? "?"
```

The implementation of a parser for this grammar is direct. The most complex parser is the one for `SYM`, see Fig. 2. This uses terminal parsers such as `?AZS?` whose implementation is given in Fig. 1. The top-level parser for `RULES` returns a list of rules. To turn the list of rules into a parser, we use the parser generator in Fig. 3.

The parser generator `rules_to_parser` is parametrised by a function `tm_to_p` which converts terminals such as `?AZS?` to parsers such as `parse_AZS`. The argument `rs` is a grammar (a list of rules), and the argument `nt` identifies which nonterminal the parser

```
let parse_SYM = fun i ->
  ((((a "\"") **> parse_notdquote **> (a "\"")) >> (fun (_,(s,_)) -> TM("\"" ^ (content s) ^ "\""))
  ||| (((a "'") **> parse_notsquote **> (a "'")) >> (fun (_,(s,_)) -> TM("'" ^ (content s) ^ "'")))
  ||| (parse_AZS >> (fun s -> NT (content s)))
  ||| (((a "?") **> parse_azAZs **> (a "?")) >> (fun (_,(s,_)) -> TM("?" ^ (content s) ^ "?"))))
    i
```

Figure 2. A `SYM` parser

```
(* (term -> parse_tree parser) -> grammar -> nonterm -> parse_tree parser *)
let rec rules_to_parser tm_to_p rs nt = fun i ->
  let rules = List.filter (fun (a,b) -> a = nt) rs in    (* rules for nt *)
  let rhss = List.map snd rules in                       (* right-hand sides of rules *)
  let alts1 = List.concat rhss in                        (* alternatives are lists of symbols *)
  let sym_to_p sym =                                     (* transform a symbol to a parser *)
    match sym with
    | TM lit -> tm_to_p lit                              (* base case *)
    | NT nt -> rules_to_parser tm_to_p rs nt             (* recursive case *)
    | SN _ -> never                                      (* impossible, no singleton in rhs *)
  in
  let alts2 = List.map (List.map sym_to_p) alts1 in      (* alternatives are now lists of parsers *)
  let g ps = then_list ps >> (fun xs -> PT(NT nt,xs)) in (* transform an alternative to a parser *)
  let alts3 = List.map g alts2 in                        (* alternatives are now parsers *)
  let p = or_list alts3 in                               (* a single parser for all alternatives *)
  check_and_upd_ctxt nt p i                              (* apply parser to input *)
```

Figure 3. A parser generator

is for. The code gathers a list of alternatives from the right-hand sides of rules for `nt`. Each alternative is a list of terminals and nonterminals. Nonterminals are transformed to parsers by calling `rules_to_parser` recursively. Terminals are transformed to basic parsers via `tm_to_p`. The list of alternatives is then transformed to a single parser using the combinators `then_list` and `or_list`. Finally, to avoid non-termination, the result parser is wrapped in a call to `check_and_upd_ctxt`.

Theorem 31 (parser-generator-correct)**.** *Given a total mapping* `tm_to_p` *such that* `tm_to_p` X *is a terminating, sound and complete parser for all terminals* X*, and a grammar* G *formed from a set of nonterminal parse rules* `rs`*, then for all nonterminals* `nt`*, the parser generator* `rules_to_parser` *produces a sound, complete, and terminating parser for* `nt`*.*

Proof. The proof of correctness for an arbitrary grammar G and arbitrary nonterminal `nt` is a direct generalization of the correctness proof for the particular grammar and particular nonterminal from Lemma 30. For example, for termination we show simultaneously that the parsers for each nonterminal `nt` are terminating on input $(c, (s, l, h))$ using exactly the same measure from Lemma 30. □

Of course, the parser generator can accept its own grammar as input.

10. Memoization

The main focus of this paper is on correctness, in particular completeness. Completeness and efficiency are hard to reconcile. Our parsers are complete in the sense that they return all final parse trees with no bad nodes. Some highly-ambiguous left-recursive grammars generate an exponential number of such parse trees. If we enforce completeness, we seemingly must accept that our parsers take an exponential amount of time. Of course, many grammars do not generate exponentially many non-redundant parse trees.

Even if we insist on completeness, we still want our parsers to be as efficient as possible. An obvious way to increase efficiency of combinator parsers is to use memoization. Following [Norvig 1991] memoized combinator parsing is polynomial time for non-left-recursive grammars. In the presence of a context, efficient memoization is not straightforward, so here we briefly outline the issues.

Given a function f taking a single argument x and to a result f_x, memoization involves storing the pair (x, f_x) in a lookup table when f first returns, and on subsequent invocations of f on the same argument x, returning the result f_x without actually executing f.

A parser p is a function which takes inputs as arguments. Here, an input is a pair (c, s) of a context c and a substring s. Naively adopting memoization is inefficient, because the context c is very redundant: there are many contexts c' which are different from c, but for which a parser, when called with the pair (c', s) will give the same result as when called with the pair (c, s): c and c' are equivalent as far as p is concerned. If we fail to take this into account, then we will compute $p\,(c, s)$ and $p\,(c', s)$ (and there may be many such c') whereas we need only compute $p\,(c, s)$.

The most obvious source of redundancy is that contexts are semantically sets, but are implemented as lists. Thus, if c' is a permutation of c, then c' is equivalent to c. A typical fix is to ensure that there is only one way to represent contexts that are permutations of each other eg by sorting contexts lexicographically on the first component, and by substring length on the second (the substrings are nested, so this is indeed a total order).

When a parser for nonterminal X is given a substring (s, l, h) and a context c as argument, then the context contains pairs of a terminal X_i and a substring (s, l_i, h_i). It is clear that any ascendant must be parsing a substring that contains the substring currently being considered. Moreover, the only items in the context that affect the current call are those such that $(l_i, h_i) = (l, h)$. Thus, two contexts c and c' which agree on the nonterminals X_i in the

context with $(l_i, h_i) = (l, h)$ are equivalent. The obvious fix when memoizing a function call with context c as an argument is to discard all those terminals $X_j \in c$ where $(l_j, h_j) \neq (l, h)$.

Memoization can dramatically improve parsing performance, especially for highly ambiguous grammars. However, existing arguments about the efficiency of memoization for combinator parsing [Norvig 1991] cannot be applied directly. The problem lies with the input component s which is a substring rather than the usual string. Traditionally parsers try to parse prefixes of some suffix s of the original input s_0. Memoization ensures that the parser will only ever be called once for a given suffix s, equivalently, the parser will only ever be called once to parse from position i in the original string s_0. With our approach, even using memoization, the parser *may be called many times* to parse a substring starting at position i, since we do not try to parse the rest of the original input s_0, but the rest of the substring (s_0, i, j), and j can take many values. This is the major source of inefficiency with our approach.

11. Mechanization in a theorem prover

Mechanization of definitions and proofs in a theorem prover can provide strong guarantees of correctness that surpass what is possible with informal mathematics. In this section we give a brief guide to mechanizing the results of the preceding sections.

The easiest approach to mechanizing these results is to rephrase the OCaml code in the internal programming language of a theorem prover such as Coq [The Coq Team], Isabelle [Paulson et al.] or Hol [Norrish et al.]. The OCaml code is simply-typed and purely functional and can therefore be transcribed directly into these theorem provers. An alternative would be to mechanize the operational semantics of OCaml and prove that the OCaml implementation is correct. This provides little further assurance but is significantly more difficult.

The top-level goal should be to prove a mechanized version of Thm. 31. This theorem only requires finite parse trees and the only results needed from Sect. 6 are those concerning bad nodes. In particular, the infinite characterization of Thm. 22 which involves a detour via König's lemma does not need to be mechanized. Working with finite parse trees makes the mechanization significantly easier. For example, the matched parse trees of Defn. 21 can be defined as a *function* from parse trees to substrings using primitive recursion on the structure of the tree.

12. Related work

A large amount of valuable research has been done in the area of parsing. We do not aim to survey the entire field, but instead aim to give complete references to work that is most directly related to our own.

Context-free grammars were first identified in [Chomsky 1956]. Context-free grammars enforce block structure. Block-structured programming was introduced by the ALGOL programming language [Naur et al. 1960], two of the designers of which gave their names to the Backus-Naur Form, a formalism for defining context-free grammars.

The most famous parser generator is Yacc [Johnson 1975], but it cannot handle arbitrary context-free grammars. The first parser generators that can handle arbitrary context-free grammars are based on dynamic programming. Examples include CYK parsing [Kasami 1965] and Earley parsing [Earley 1970]. In these early works, the emphasis is on implementation concerns, and in particular completeness is often not clear. For example [Tomita 1986] notes that Earley parsing is not complete for rules involving the empty string terminal `""` (also known as epsilon). However, it is *in principle* clear that variants of these approaches can be proven complete for arbitrary context-free grammars.

Combinator parsing and related techniques are probably folklore. An early approach with some similarities is [Pratt 1973]. Versions that are clearly related to the approach taken in this paper were popularized in [Hutton 1992; Wadler 1985].

Ambiguous grammars can give rise to an infinite number of parse trees for a given input. Parsers that return a list of parse trees must therefore omit some. Our approach avoids "bad" nodes, which indirectly forces termination, but most other approaches attempt to bound the recursion directly, based on the length of the input string. The first approach to use the length of the input to force termination is [Kuno 1965]; the focus is on implementation and efficiency, and completeness is not addressed. Similar current work by Frost et al., which is the most closely related to our own, was discussed in Sect. 1.

[Norvig 1991] shows that polynomial time complexity can be achieved in mutually-recursive top-down parsing by using memoization. Unfortunately his argument "assumes that there are no left-recursive rules". This restriction is lifted in [Johnson 1995]. The cost is a much more complicated implementation via continuations, and it is not clear that this preserves polynomial time complexity, or correctness when combined with memoization.

An interesting recent development is the formal verification of parsers. Current examples such as [Barthwal and Norrish 2009; Koprowski and Binsztok 2010] do not use combinator parsing, and cannot handle all context-free grammars.

13. Conclusion

We presented a parser generator that takes an arbitrary grammar as an argument and produces a parser for the grammar as a result. The result parser is terminating, sound and complete for the grammar. In addition, we noted that any approach that, given input s, returns all parse trees where the maximum nesting depth of a parser for a nonterminal X is $|s| + 1$ will be complete for *arbitrary context-free grammars*. This gives a correctness proof for the approach of [Frost et al. 2007, 2008; Hafiz and Frost 2010].

Our proofs use informal mathematics. Following Sect. 11 it should be straightforward to mechanize the proofs in a theorem prover such as Hol [Norrish et al.], Isabelle [Paulson et al.] or Coq [The Coq Team].

Existing formalisms, such as BNF, deal only with grammar rules. Real implementations typically also deal with "semantic actions", that is, what to do with the results of a parse. We use very basic actions since our parsers produce only parse trees. It may be worth formalizing the notion of semantic action, and extending BNF to take actions into account. The paper [Koprowski and Binsztok 2010] does this for parsing expression grammars, by simply allowing an arbitrary Coq function as part of the syntax of the grammar. However, to be generally useful, some particular syntax for actions should be defined.

The focus of our work is on correctness, and we believe our approach inherits all the benefits of combinator parsing. For practical applications, however, efficiency is often the overriding concern. We hope that alternative implementation strategies based on our ideas can be made competitive with the most efficient alternatives, such as Packrat parsing [Ford 2002], at least when applied to the same restricted grammar classes that these alternatives target. We leave alternative implementation strategies, and efficient implementation for future work.

Parsing and pretty-printing are related activities. Future work should aim to produce a companion pretty-printer for a given parser, such that pretty-printing followed by parsing is the identity function. To be practically useful, a pretty-printer should try to minimize the amount of redundant information such as superfluous brackets.

The main motivation for this work was to produce simple, sound and complete parsers for all context-free grammars based on combinator parsing. We feel the parser generator is optimal in terms of clarity and elegance. We would be very pleased if these techniques were taken up and used in real applications. In particular, we have no insight as to whether the efficiency problems outlined in Sect. 10 are a problem in practice, and we therefore welcome feedback on these issues.

References

J. Aycock and R. N. Horspool. Practical earley parsing. *Comput. J.*, 45(6): 620–630, 2002.

A. Barthwal and M. Norrish. Verified, executable parsing. In G. Castagna, editor, *ESOP*, volume 5502 of *Lecture Notes in Computer Science*, pages 160–174. Springer, 2009. ISBN 978-3-642-00589-3.

N. Chomsky. Three models for the description of language. *IRE Trans. Info. Theory*, 1:113–124, 1956.

J. Earley. An efficient context-free parsing algorithm. *Commun. ACM*, 13 (2):94–102, 1970. ISSN 0001-0782. doi: http://doi.acm.org/10.1145/362007.362035.

B. Ford. Packrat parsing: simple, powerful, lazy, linear time, functional pearl. In *ICFP 02: Proceedings of the seventh ACM SIGPLAN international conference on Functional programming*, volume 37/9, pages 36–47, New York, NY, USA, 2002. ACM. doi: 10.1145/583852.581483. URL `http://pdos.csail.mit.edu/~baford/packrat/icfp02/packrat-icfp02.pdf`.

R. Frost and J. Launchbury. Constructing natural language interpreters in a lazy functional language. *Comput. J.*, 32(2):108–121, 1989. ISSN 0010-4620. doi: http://dx.doi.org/10.1093/comjnl/32.2.108.

R. A. Frost. Constructing programs as executable attribute grammars. *Comput. J.*, 35(4):376–389, 1992.

R. A. Frost. Using memoization to achieve polynomial complexity of purely functional executable specifications of non-deterministic top-down parsers. *SIGPLAN Notices*, 29(4):23–30, 1994.

R. A. Frost and R. Hafiz. A new top-down parsing algorithm to accommodate ambiguity and left recursion in polynomial time. *SIGPLAN Notices*, 41(5):46–54, 2006.

R. A. Frost, R. Hafiz, and P. C. Callaghan. Modular and efficient top-down parsing for ambiguous left-recursive grammars. In *IWPT '07: Proceedings of the 10th International Conference on Parsing Technologies*, pages 109–120, Morristown, NJ, USA, 2007. Association for Computational Linguistics. ISBN 978-1-932432-90-9.

R. A. Frost, R. Hafiz, and P. Callaghan. Parser combinators for ambiguous left-recursive grammars. In P. Hudak and D. S. Warren, editors, *PADL*, volume 4902 of *Lecture Notes in Computer Science*, pages 167–181. Springer, 2008. ISBN 978-3-540-77441-9.

R. Hafiz and R. A. Frost. Lazy combinators for executable specifications of general attribute grammars. In M. Carro and R. Peña, editors, *PADL*, volume 5937 of *Lecture Notes in Computer Science*, pages 167–182. Springer, 2010. ISBN 978-3-642-11502-8.

G. Hutton. Higher-order functions for parsing. *J. Funct. Program.*, 2(3): 323–343, 1992.

M. Johnson. Memoization in top-down parsing. *Computational Linguistics*, 21(3):405–417, 1995.

S. C. Johnson. Yacc: Yet another compiler compiler. Computer Science Technical Report #32, Bell Laboratories, Murray Hill, NJ, 1975.

T. Kasami. An efficient recognition and syntax analysis algorithm for context-free languages. Technical Report AFCRL-65-758, Air Force Cambridge Research Laboratory, Bedford, Massachusetts, 1965.

A. Koprowski and H. Binsztok. TRX: A formally verified parser interpreter. In A. D. Gordon, editor, *ESOP*, volume 6012 of *Lecture Notes in Computer Science*, pages 345–365. Springer, 2010. ISBN 978-3-642-11956-9.

S. Kuno. The predictive analyzer and a path elimination technique. *Commun. ACM*, 8(7):453–462, 1965. ISSN 0001-0782. doi: http://doi.acm.org/10.1145/364995.365689.

X. Leroy et al. OCaml. `http://caml.inria.fr/`.

P. Naur, J. W. Backus, et al. Report on the algorithmic language ALGOL 60. *Communications of the ACM*, 3(5):299–314, May 1960.

M. Norrish et al. The HOL4 theorem prover. `http://hol.sourceforge.net/`.

P. Norvig. Techniques for automatic memoization with applications to context-free parsing. *Comput. Linguist.*, 17(1):91–98, 1991. ISSN 0891-2017.

L. Paulson, T. Nipkow, and M. Wenzel. The Isabelle distribution. `http://www.cl.cam.ac.uk/Research/HVG/Isabelle`.

V. R. Pratt. Top down operator precedence. In *Proceedings ACM Symposium on Principles Prog. Languages*, 1973.

The Coq Team. The Coq Theorem Prover. `http://coq.inria.fr/`.

M. Tomita. *Efficient Parsing for Natural Language: A Fast Algorithm for Practical Systems*. Kluwer, Boston, 1986.

P. Wadler. How to replace failure by a list of successes: A method for exception handling, backtracking, and pattern matching in lazy functional languages. In *FPCA*, pages 113–128, 1985.