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

Tom Ridge

University of Leicester

Abstract. 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 then construct a generic parser generator and show that generated parsers are sound and complete. The formal proofs are mechanized using the HOL4 theorem prover. Memoized parsers based on our approach are polynomial-time in the size of the input. Preliminary real-world performance testing on highly ambiguous grammars indicates our parsers are faster than those generated by the popular Happy parser generator.

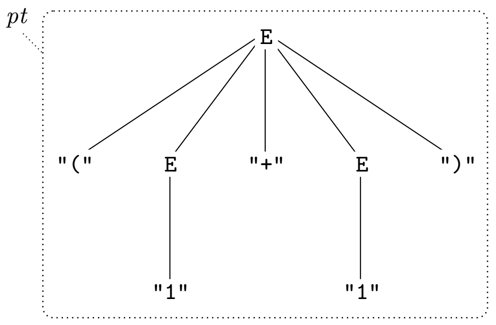
1 Introduction

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 a set of rules in Backus-Naur Form (BNF). An example[1] of a simple grammar with a single rule for a nonterminal `E` (with two alternative expansions) is `E -> "(" E "+" E ")" | "1"`. A parse tree is a finite tree where each node is formed according to the grammar rules. We can concatenate the leaves of a parse tree *pt* to get a string (really, a substring option) substring_of *pt* accepted by the grammar, see Fig. 1. A parser for a grammar is a program that takes an input string and returns parse trees for that string.

A popular parser implementation strategy is combinator parsing, wherein 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]

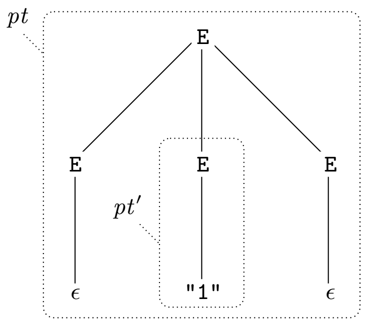
[1] Real BNF requires a nonterminal such as `E` to be written as `<E>`.

[2] The examples are based on real OCaml implementations, but are formally pseudocode because OCaml function names must start with a lowercase letter.



substring_of pt = SOME "(1+1)"

Fig. 1.



substring_of pt = substring_of pt'

Fig. 2.

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

The code 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 clear because recursive calls to `E` are given strictly less input to parse.

Combinator parsing cannot be used directly if the grammar contains rules such as `E -> E E E` that are left-recursive. For example, a naive attempt to implement the grammar `E -> E E E | "1" |` ϵ[3] gives

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

This code would attempt to parse an `E` by first expanding to `E E E`, and then recursively attempting to parse an `E` on the same input, leading to non-termination. One solution is to alter the original grammar specification to avoid left recursion, but this is undesirable for several reasons, and may not always be possible. Despite these drawbacks combinator parsing is conceptually simple, almost trivial to implement, and integrates smoothly with the host language (typically a simply-typed functional programming language). For these reasons *combinator parsing is extremely popular*, and many systems include hand-crafted parsers based on combinator parsing. In contrast, the complicated implementations of other parsing techniques have many associated drawbacks. For example, Yacc produces error messages, such as "shift/reduce conflict", that are *incomprehensible* without a good knowledge of the underlying implementation technology.

Contribution 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. The heart of our contribution is a parser wrapper (a function from parsers to parsers) `check_and_upd_lctxt` which wraps

[3] The terminal representing the empty string `""` is usually written ϵ.

```
grammar_to_parser $p\_of\_tm$ $g$ $sym$ $i$ = case $sym$ of
  TM $tm \rightarrow ((p\_of\_tm\ tm) \gg (\lambda\ v.\ \mathsf{LF}(tm, v)))\ i$ || NT $nt \rightarrow$
  let $rules$ = FILTER $(\lambda\ (nt', rhs).\ nt' = nt)\ g$ in
  let $alts1$ = (FLAT $\circ$ (MAP SND)) $rules$ in
  let $alts2$ = MAP (MAP ($\lambda\ sym.$ grammar_to_parser $p\_of\_tm\ g\ sym$)) $alts1$ in
  let $p$ = or_list (MAP (then_list2 $nt$) $alts2$) in
  check_and_upd_lctxt $nt\ p\ i$)
```

The parser generator grammar_to_parser is parameterized by: a function p_of_tm which gives a parser for each terminal; the grammar g (a list of BNF-type rules); and sym, the symbol corresponding to the parser that should be generated. If sym is a terminal tm then $p\_of\_tm\ tm$ gives the appropriate parser. If sym is a nonterminal nt then the relevant *rules* are filtered from the grammar, the right hand sides are combined into a list of alternatives *alts1*, grammar_to_parser is recursively mapped over *alts1*, and finally the results are combined using the parser combinators or_list and then_list2 to give a parser p. In order to prevent nontermination p is wrapped by check_and_upd_lctxt.

Fig. 3. A verified, sound and complete parser generator (HOL4)

the body of an underlying parser and eliminates some parse attempts whilst preserving completeness. For example, for the grammar `E -> E E E | "1" |` ϵ, a terminating, sound and complete parser can be written as follows:

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

The first argument `"E"` to `check_and_upd_lctxt` is necessary to indicate which nonterminal is being parsed in case the grammar contains more than one nonterminal. In Fig. 3 we define a parser generator for arbitrary context-free grammars based on this parser wrapper (the reader should not expect to understand the code at this point). We prove the parser generator correct using the HOL4 theorem prover. Our approach retains the simplicity of combinator parsing, including the ability to incorporate standard extensions such as "semantic actions". The worst-case time complexity of our algorithm when memoized is $O(n^5)$. In real-world performance comparisons on highly ambiguous grammars, our parsers are consistently faster than those generated by the Happy parser generator [1].

Key ideas Consider the highly ambiguous grammar `E -> E E E | "1" |` ϵ. This gives rise to an infinite number of parse trees. A parser cannot hope to return an infinite number of parse trees in a finite amount of time. However, many parse trees pt have proper subtrees pt' such that both pt and pt' are rooted at the same nonterminal, and substring_of pt = substring_of pt', see Fig. 2. *This is the both the cause of the infinite number of parse trees, and the underlying cause of non-termination in implementations of combinator parsing.*

We call a parse tree bad if it *contains* a subtree such as pt. If we rule out bad trees we can still find a good tree for any parse-able input. Moreover, given a context-free grammar g and input s, it turns out that there are at most a *finite* number of good parse trees pt such that substring_of pt = SOME s. Thus for a given grammar we have identified a class of parse trees (the good parse trees) that is complete (any input that can be parsed, can be parsed to give a good parse tree) and moreover is *finite*.

At the implementation level, we construct a function `check_and_upd_lctxt` which wraps the body of an underlying parser and eliminates parse attempts that would lead to nontermination by avoiding bad parse trees. This requires the parser input type to be slightly modified to include information about the parsing context (those parent parses that are currently in progress), but crucially this is invisible to the parser writer who simply makes use of standard parser combinators. Generalizing this approach gives the parser generator in Fig. 3.

Structure of the paper In Sect. 2 we define the types used in later sections, and give a brief description of the formalization of substrings. Sect. 3 discusses the relationship between grammars and parse trees, whilst Sect. 4 discusses the relationship between parse trees and the parsing context. The standard parsing combinators are defined in Sect. 5. The new functions relating to the parsing context, including check_and_upd_lctxt, are defined in Sect. 6. The remainder of the body of the paper is devoted to correctness. In Sect. 7 we discuss termination and soundness. In Sect. 8 we formalize informal notions of completeness, and in Sect. 9 we show that our parser generator produces parsers that are complete. In Sect. 10 we discuss implementation issues, such as memoization and performance. Finally we discuss related work and conclude. Our implementation language is OCaml and the complete OCaml code and HOL4 proof script are available online[4], together with example grammars and sample inputs. For reasons of space, in this paper we occasionally omit definitions of straightforward well-formedness predicates such as wf_grammar. We give outlines of the proofs, including the main inductions, but do not discuss the proofs in detail. The interested reader will find all definitions and proofs in the mechanized HOL4 proof scripts online.

Notation BNF grammars are written using `courier`, as is OCaml code and pseudo-code. Mechanized HOL4 definitions are written using sans_serif for defined constants, and *italic* for variables. Common variable names are displayed in Fig. 4, but variations are also used. For example, if x is a variable of type α then xs is a variable of type α list. Similarly suffixing and priming are used to distinguish several variables of the same type. For example, s, s', s_pt, s_rem and s_tot are all common names for variables of type substring. For presentation purposes, we occasionally blur the distinction between strings and substrings. Records are written $\langle$ fld $= v;\ \ldots\ \rangle$. Update of record r is written r with $\langle$ fld $= v$ $\rangle$. Function application is written $f\ x$. List cons is written $x :: xs$. The empty list is $[]$. List membership is written MEM $x\ xs$. Other HOL4 list functions should be comprehensible to readers with a passing knowledge of functional programming.

2 Types and Substrings

Figure 4 gives the basic types we require. In the following sections, it is formally easier to work with substrings rather than strings. A substring (s, l, h) repre-

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

s	:string
l, h	:num
s	:substring
tm	:term = ty_term
nt	:nonterm = ty_nonterm
sym	:symbol = TM of term | NT of nonterm
$rhs, alts$	:(symbol list) list
$r, rule$	:parse_rule = nonterm $\times$ ((symbol list) list)
g	:grammar = parse_rule list
pt	:parse_tree = NODE of nonterm $\times$ parse_tree list | LF of term $\times$ substring
q	:simple_parser = substring $\rightarrow$ parse_tree list
lc	:context = (nonterm $\times$ substring) list
i	:ty_input = $\langle$ lc : context; sb : substring $\rangle$
p	:α parser = ty_input $\rightarrow$ (α $\times$ substring) list
ss_of_tm	:ty_ss_of_tm = term $\rightarrow$ substring set
p_of_tm	:ty_p_of_tm = term $\rightarrow$ substring parser

Fig. 4. Common variable names for elements of basic types, with type definitions

$\mathsf{string}\ s = \mathsf{let}\ (s,l,h) = s\ \mathsf{in}\ s$

$\mathsf{low}\ s = \mathsf{let}\ (s,l,h) = s\ \mathsf{in}\ l$

$\mathsf{high}\ s = \mathsf{let}\ (s,l,h) = s\ \mathsf{in}\ h$

$\mathsf{len}\ s = \mathsf{let}\ (s,l,h) = s\ \mathsf{in}\ h\ -\ l$

$\mathsf{wf\_substring}\ (s,l,h) = l \leq h \wedge h \leq |s|$

$\mathsf{inc\_low}\ n\ s = \mathsf{let}\ (s,l,h) = s\ \mathsf{in}\ (s, l+n, h)$

$\mathsf{dec\_high}\ n\ s = \mathsf{let}\ (s,l,h) = s\ \mathsf{in}\ (s, l, h-n)$

$\mathsf{inc\_high}\ n\ s = \mathsf{let}\ (s,l,h) = s\ \mathsf{in}\ (s, l, h+n)$

$\mathsf{full}\ s = (s, 0, |s|)$

$\mathsf{toinput}\ s = \langle\ \mathsf{lc} = [];\ \mathsf{sb} = s\ \rangle$

$\mathsf{concatenate\_two}\ s1\ \ s2 = \mathsf{if}\ (\mathsf{string}\ s1 = \mathsf{string}\ s2) \wedge (\mathsf{high}\ s1 = \mathsf{low}\ s2)\ \mathsf{then}$
$\quad \mathsf{SOME}\ ((\mathsf{string}\ s1, \mathsf{low}\ s1, \mathsf{high}\ s2))\ \mathsf{else}\ \mathsf{NONE}$

$\mathsf{concatenate\_list}\ ss = \mathsf{case}\ ss\ \mathsf{of}\ [] \rightarrow \mathsf{NONE}\ \|\ [s1] \rightarrow (\mathsf{SOME}\ s1)\ \|\ s1 :: ss1 \rightarrow ($
$\quad \mathsf{case\ concatenate\_list}\ ss1\ \mathsf{of\ NONE} \rightarrow \mathsf{NONE}\ \|\ \mathsf{SOME}\ s2 \rightarrow \mathsf{concatenate\_two}\ s1\ \ s2)$

Fig. 5. Common functions on substrings

sents the part of a string s between a low index l and a high index h. The type substring consists only of well-formed triples (s, l, h). Common substring functions, including the well-formedness predicate, are defined in Fig. 5. Returning to Fig. 4, the type of terminals is term; the type of nonterminals is nonterm. Formally terminals and nonterminals are kept abstract, but in the OCaml implementation they are strings. Symbols are the disjoint union of terminals and nonterminals. A parse rule such as `E -> E E E | "1" |` ϵ consists of a nonterminal l.h.s. and several alternatives on the r.h.s. (an alternative is simply a list of symbols). A grammar is a list of parse rules (really, a finite set) and a parse tree consists of nodes (each decorated with a nonterminal), or leaves (each decorated with a terminal and the substring that was parsed by that terminal). A simple parser takes an input substring and produces a list of parse trees.

Combinator parsers typically parse prefixes of a given input, and return (a list of) a result value paired with the substring that remains to be parsed: α preparser = substring $\rightarrow$ (α $\times$ substring) list. Rather than taking just a substring as input, our parsers need additional information about the context. The context, type $\mathsf{context} = (\mathsf{nonterm} \times \mathsf{substring})\ \mathsf{list}$, records information about which nonterminals are already in the process of being parsed, and the substring that each parent parse took as input. The input i for a parser is just a record with two fields: the usual substring $i.\mathsf{sb}$, and the context $i.\mathsf{lc}$. We emphasize that this slight increase in the complexity of the input type is invisible when using our parser combinators as a library: the only code that examines the context is `check_and_upd_lctxt`.

Whilst BNF grammars clearly specify how to expand nonterminals, in practice the specification of terminal parsers is more-or-less arbitrary. Formally, we should keep terminal parsers loosely specified. The set $\mathsf{pts\_of}\ \mathit{ss\_of\_tm}\ g$ of parse trees for a grammar is therefore parameterized by a function $\mathit{ss\_of\_tm}$ such that $\mathsf{LF}(\mathit{tm}, s)$ is a parse tree only if $s \in \mathit{ss\_of\_tm}\ \mathit{tm}$. A function of type $\mathsf{ty\_p\_of\_tm}$ gives a parser for each terminal.

3 Grammars and Parse Trees

Parse trees pt and pt' match if they have the same root symbol, and $\mathsf{substring\_of}\ pt = \mathsf{substring\_of}\ pt'$. A parse tree has a bad root if it contains a proper subtree that matches it. For example, in Fig. 2, pt has a bad root because subtree pt' matches it. A good tree is a tree such that no *subtrees* have bad roots. The following theorem implies that any parse-able input can be parsed to give a good tree.

Theorem 1 (good_tree_exists_thm). *Given a grammar g, for any parse tree pt one can construct a good tree pt' that matches.*

$$\begin{array}{l}\boxed{\begin{array}{l}\mathsf{good\_tree\_exists\_thm} = \forall\ \mathit{ss\_of\_tm}.\ \forall\ g.\ \forall\ pt.\ \exists\ pt'.\\ \quad pt \in (\mathsf{pts\_of}\ \mathit{ss\_of\_tm}\ g)\\ \quad \longrightarrow pt' \in (\mathsf{pts\_of}\ \mathit{ss\_of\_tm}\ g) \wedge (\mathsf{matches}\ pt\ pt') \wedge (\mathsf{good\_tree}\ pt')\end{array}}\end{array}$$

Proof. If pt_0 is not good, then it contains a subtree pt and a proper subtree pt' of pt that matches pt. If we replace pt by pt' we have reduced the number of subtrees which have bad roots. The transformed tree is well-formed according to the grammar and matches the original. If we repeat this step, we can eliminate all subtrees with bad roots.

4 Parse Trees and the Parsing Context

In this section we define an inductive relationship $\mathsf{admits}\ lc\ pt$ between parsing contexts lc and parse trees pt. We use a context during parsing to eliminate parse attempts that would lead to bad parse trees and potential non-termination. If lc is the empty context $[]$, then the function $\lambda\ pt.\ \mathsf{admits}\ []\ pt$ actually characterizes good trees ie $\mathsf{admits}\ []\ pt \leftrightarrow \mathsf{good\_tree}\ pt$. The definition of $\mathsf{good\_tree}$ is wholly in terms of parse trees, whilst the parse trees returned by our parsers depend not only on the parsing context, but on

```
$p1$ **> $p2 = \lambda\ i.$
  let $f\ (e1, s1) =$
    MAP $(\lambda\ (e2, s2).\ ((e1, e2), s2))\ (p2\ \langle$ lc=$i$.lc; sb=$s1\ \rangle)$
  in
  (FLAT $\circ$ (MAP $f$) $\circ$ $p1$) $i$

$p1\ |||\ p2 = \lambda\ i.$ APPEND $(p1\ i)\ (p2\ i)$        $p \gg f =$ (MAP $(\lambda\ (e, s).\ (f\ e,\ s))) \circ p$

always $= (\lambda\ i.\ [([], \mathsf{substr}\ i)]) : \alpha$ list parser    never $= (\lambda\ i.\ []) : \alpha$ parser

then_list ($ps : \alpha$ parser list) = case $ps$ of    or_list ($ps : \alpha$ parser list) = case $ps$ of
  $[] \rightarrow$ always                                  $[] \rightarrow$ never
  $||\ p :: ps \rightarrow ((p$ **> (then_list $ps$))            $||\ p :: ps \rightarrow (p\ |||$ (or_list $ps$))
    $\gg (\lambda\ (x, xs).\ x :: xs))$

then_list2 $nt = \lambda\ ps.$
  then_list $ps \gg (\lambda\ xs.$ NODE$(nt, xs))$
```

Fig. 6. Parser combinators

complicated implementation details of the parsers themselves. The definition of admits serves as a bridge between these two, incorporating the parsing context, but usefully omitting complicated parser implementation details.

```
admits $lc\ pt$ =
  let $s\_pt$ = THE(substring_of $pt$) in
  case $pt$ of
    NODE$(nt, pts) \rightarrow (\neg$(MEM $(nt, s\_pt)\ lc) \wedge$ EVERY (admits $((nt, s\_pt) :: lc))\ pts)$
    || LF(_, _) $\rightarrow$ T
```

The function THE is the projection from the option type: THE(SOME x) $= x$. Let s_pt = THE(substring_of pt). This definition states that, for a parse tree pt with root nt to be admitted, the pair (nt, s_pt) must not be in the context lc, *and moreover* if we extend the context by the pair (nt, s_pt) to get a context lc', then every immediate subtree of pt must be admitted by lc'. Leaf nodes are always admitted. As an example, consider the bad parse tree pt in Fig. 2 which is not admitted by the empty context, $\neg$ (admits [] pt). In this case, $(nt, s\_pt) = (\texttt{E}, \texttt{"1"})$, so that $lc' = [(\texttt{E}, \texttt{"1"})]$, and clearly $\neg$ (admits $lc'\ pt'$). Our parsers return all parse trees admitted by the empty context, which is initially empty. The following theorem guarantees that this includes all good trees.

Theorem 2 (admits_thm). *Good parse trees are admitted by the empty context.*

```
admits_thm = $\forall\ pt.$ wf_parse_tree $pt \longrightarrow$ good_tree $pt \longrightarrow$ admits [] $pt$
```

Proof. Induction on the size of pt.

5 Terminal Parsers and Parser Combinators

The basic parser combinators are defined in Fig. 6. The standard definition of the alternative combinator ||| appends the output of one parser to the output of

```
update_lctxt $nt$ ($p : \alpha$ parser) = $\lambda$ $i$.
  $p$ ($i$ with ⟨ lc=($nt, i$.sb)::$i$.lc ⟩)

ignr_last ($p : \alpha$ parser) = $\lambda$ $i$.
  if len (substr $i$) = 0 then [] else
  let dec = dec_high 1 in
  let inc ($e, s$) = ($e$, inc_high 1 $s$) in
  ((MAP inc) ∘ $p$ ∘ (lift dec)) $i$

check_and_upd_lctxt $nt$ ($p : \alpha$ parser) = $\lambda$ $i$.
  let should_trim =
    EXISTS ((=) ($nt, i$.sb)) $i$.lc in
  if should_trim ∧ (len $i$.sb = 0) then
    []
  else if should_trim then
    (ignr_last (update_lctxt $nt$ $p$)) $i$
  else
    (update_lctxt $nt$ $p$) $i$
```

Fig. 7. Updating the parsing context

another. The sequential combinator $**>$ uses the first parser to parse prefixes of the input, then applies the second parser to parse the remaining suffixes. This definition is almost standard, except that the parsing context i.lc is the same for *p1* as for *p2*. The "semantic action" combinator $\gg$ simply applies a function to the results returned by a parser. We generalize the basic combinators to handle lists (then_list, then_list2 and or_list).

The definition of grammar_to_parser in Fig. 3 is parametric over a function *p_of_tm* which gives a parser *p_tm* = *p_of_tm tm* for each terminal *tm*. At the implementation level, *p_tm* can be more-or-less arbitrary. However, for our results to hold, *p_of_tm* is required to satisfy a well-formedness requirement wf_p_of_tm. For soundness, parse trees *pt* returned by terminal parser *p_tm* must be such that substring_of *pt* is a prefix of the input. For completeness the parse trees produced by a terminal parser *p_tm* for a given prefix of the input should not change when the input is extended. These conditions are very natural, but the completeness condition is subtle, and has some interesting consequences, for example, it rules out lookahead which is common in lexer implementations.

6 Updating the Parsing Context

The parsing context is used to eliminate parse attempts that might lead to non-termination. In Fig. 7, update_lctxt *nt* is a parser wrapper parameterized by a nonterminal *nt*. During a parse attempt the nonterminal *nt* corresponds to the node that is currently being parsed, that is, all parse trees returned by the current parse will have root *nt*. The parser *p* corresponds to the parser that will be used to parse the *immediate subtrees* of the current tree. The wrapper update_lctxt *nt* ensures that the context i.lc is extended to $(nt, i.\mathsf{sb})::i.\mathsf{lc}$ before calling the underlying parser *p* on the given input *i*.

The parser wrapper ignr_last calls an underlying parser *p* on the input *minus the last character* (via dec_high); the unparsed suffix of the input then has the last character added back (via inc_high) before the results are returned. The purpose of ignr_last is to force termination when recursively parsing the same nonterminal, by successively restricting the length of the input that is available to parse.

The heart of our contribution is the parser wrapper check_and_update_lctxt *nt* which is also parameterized by a nonterminal *nt*. This combinator uses the context to eliminate parse attempts. As before, nt corresponds to the node that is currently being parsed. The boolean should_trim is true iff the context i.lc contains a pair $(nt, i.\mathsf{sb})$. If this is the case, then we can safely restrict our parse attempts to *proper prefixes* of the input i.sb, by wrapping update_lctxt nt p in ignr_last. Theorem main_thm in Sect. 9 guarantees that this preserves completeness. At this point we have covered the definitions required for the parser generator in Fig. 3.

7 Termination, Soundness and Prefix-Soundness

In this section we show that the definition of grammar_to_parser in Fig. 3 is well-formed and terminating by giving a well-founded measure that decreases with each recursive call. We then define formally what it means for a parser to be sound. We also define the stronger property of prefix-soundness. The parser generator grammar_to_parser generates prefix-sound parsers.

The following well-founded measure function is parameterized by the grammar g and gives a natural number for every input i. In Fig. 3, recursive calls to grammar_to_parser are given inputs with strictly less measure, which ensures that the definition is well-formed and that all parses terminate. The function SUM computes the sum of a list of numbers.

```
measure $g$ $i$ =
  let $nts$ = nonterms_of_grammar $g$ in
  let $f$ $nt$ = len $i$.sb + (if MEM $(nt, i.\mathsf{sb})$ $i$.lc then 0 else 1) in
  SUM(MAP $f$ $nts$)
```

Theorem 3. *Recursive calls to* grammar_to_parser *are given inputs i' whose measure is strictly less than the measure of the input i provided to the parent.*

Proof. The proof proceeds in two steps. First, we show by analysis of cases that the invocation of p when evaluating check_and_upd_lctxt nt p i is called with an input i' with strictly less measure than that of i.

$$((\lambda\ i.\ [(i,\ s0)]) = p)$$
$$\longrightarrow \mathsf{MEM}\ nt\ (\mathsf{nonterms\_of\_grammar}\ g)$$
$$\longrightarrow \mathsf{MEM}\ (i', s)\ (\mathsf{check\_and\_upd\_lctxt}\ nt\ p\ i) \longrightarrow \mathsf{measure}\ g\ i' < \mathsf{measure}\ g\ i$$

Second, we observe that recursive calls to grammar_to_parser are nested under then_list2 nt, so that each recursive call receives an input i'' where either $i''.\mathsf{sb} = i'.\mathsf{sb}$ or len i''.sb < len i'.sb. In the latter case we have

$$\mathsf{len}\ s'' < \mathsf{len}\ s' \longrightarrow \mathsf{measure}\ g\ \langle\ \mathsf{lc} = lc;\ \mathsf{sb} = s''\ \rangle \leq \mathsf{measure}\ g\ \langle\ \mathsf{lc} = lc;\ \mathsf{sb} = s'\ \rangle$$

We now turn our attention to soundness. The simplest form of soundness requires that any parse tree pt that is returned by a parser q_sym for a symbol sym when called on input s should conform to the grammar g, have a root symbol sym, and be such that substring_of pt = SOME s.

```
sound ss_of_tm g sym q_sym = ∀ s. ∀ pt.
  wf_grammar g
  ∧ MEM pt (q_sym s)
  ⟶
  pt ∈ (pts_of ss_of_tm g)
  ∧ (root pt = sym)
  ∧ (substring_of pt = SOME s)
```

Standard implementations of the sequential combinator attempt to parse all prefixes *s_pt* of a given input substring *s_tot* and return a (list of pairs of) a parse tree *pt* and the remainder *s_rem* of the input that was not parsed. In this case, we should ensure that concatenating *s_pt* and *s_rem* gives the original input *s_tot*.

```
prefix_sound ss_of_tm g sym p_sym = ∀ s_tot. ∀ pt. ∀ s_rem. ∃ s_pt.
  wf_grammar g
  ∧ MEM (pt, s_rem) (p_sym (toinput s_tot))
  ⟶
  pt ∈ (pts_of ss_of_tm g)
  ∧ (root pt = sym)
  ∧ (substring_of pt = SOME s_pt)
  ∧ (concatenate_two s_pt s_rem = SOME s_tot)
```

Theorem 4 (prefix_sound_grammar_to_parser_thm). *Parsers generated by grammar_to_parser are prefix-sound.*

```
prefix_sound_grammar_to_parser_thm = ∀ p_of_tm. ∀ g. ∀ sym.
  let p = grammar_to_parser p_of_tm g sym in
  let ss_of_tm = ss_of_tm_of p_of_tm in
  wf_p_of_tm p_of_tm ∧ wf_grammar g ⟶ prefix_sound ss_of_tm g sym p
```

Proof. Unfolding the definition of prefix_sound, we need to show a property of parse trees *pt*. The formal proof proceeds by an outer induction on the size of *pt*, and an inner structural induction on the list of immediate subtrees of *pt*.

We now observe that a prefix-sound parser can be easily transformed into a sound parser: just ignore those parses that do not consume the whole input. For this we need simple_parser_of *p*, which returns those parse trees produced by *p* for which the entire input substring was consumed.

```
simple_parser_of (p : parse_tree parser) =
  λ s. (MAP FST ∘ FILTER (λ (pt, s'). substring_of pt = SOME s)) (p (toinput s))
```

Theorem 5 (sound_grammar_to_parser_thm). *Parsers generated by grammar_to_parser are sound when transformed into simple parsers.*

```
sound_grammar_to_parser_thm = ∀ p_of_tm. ∀ g. ∀ sym.
  let p = grammar_to_parser p_of_tm g sym in
  let ss_of_tm = ss_of_tm_of p_of_tm in
  wf_p_of_tm p_of_tm ∧ wf_grammar g ⟶ sound ss_of_tm g sym (simple_parser_of p)
```

8 Completeness and Prefix-Completeness

In previous sections we have talked informally about completeness. In this section we define what it means for a parser to be complete with respect to a grammar. We also define the stronger property of prefix-completeness.

The simplest form of completeness requires that any parse tree pt that conforms to a grammar g and has a root symbol sym should be returned by the parser q_sym for sym when called on a suitable input string.

$$\boxed{\begin{array}{l}
\mathsf{unsatisfactory\_complete}\ ss\_of\_tm\ g\ sym\ q\_sym = \forall\ s.\ \forall\ pt.\\
\quad pt \in (\mathsf{pts\_of}\ ss\_of\_tm\ g)\\
\quad \wedge\ (\mathsf{root}\ pt = sym)\\
\quad \wedge\ (\mathsf{substring\_of}\ pt = \mathsf{SOME}\ s)\\
\quad \longrightarrow\\
\quad \mathsf{MEM}\ pt\ (q\_sym\ s)
\end{array}}$$

A grammar g and an input s can give rise to a potentially infinite number of parse trees pt, but a parser can only return a finite list of parse trees in a finite amount of time. For such non-trivial grammars, no parser can be complete in the sense of the definition above. Thus, this definition of completeness is unsatisfactory. If we accept that some parse trees must be omitted, we can still require that any input that can be parsed is actually parsed, and *some* parse tree pt' is returned.

$$\boxed{\begin{array}{l}
\mathsf{complete}\ ss\_of\_tm\ g\ sym\ q\_sym = \forall\ s.\ \forall\ pt.\ \exists\ pt'.\\
\quad pt \in (\mathsf{pts\_of}\ ss\_of\_tm\ g)\\
\quad \wedge\ (\mathsf{root}\ pt = sym)\\
\quad \wedge\ (\mathsf{substring\_of}\ pt = \mathsf{SOME}\ s)\\
\quad \longrightarrow\\
\quad \mathsf{matches}\ pt\ pt' \wedge \mathsf{MEM}\ pt'\ (q\_sym\ s)
\end{array}}$$

Of course, our strategy is to return parse trees pt' as witnessed by good_tree_exists_thm.A more precise definition of completeness would require a parser to return all good trees, and main_thm from Sect. 9 shows that our parsers are complete in this sense, given the characterization of good trees via admits. One advantage of the definition above is that one does not need to understand the definition of good tree to understand our statement of completeness. We now introduce the related notion of prefix-completeness. As in the previous section, prefix-complete parsers can be transformed into complete parsers.

$$\boxed{\begin{array}{l}
\mathsf{prefix\_complete}\ ss\_of\_tm\ g\ sym\ p\_sym = \forall\ s\_tot.\ \forall\ s\_pt.\ \forall\ s\_rem.\ \forall\ pt.\ \exists\ pt'.\\
\quad (\mathsf{concatenate\_two}\ s\_pt\ s\_rem = \mathsf{SOME}\ s\_tot)\\
\quad \wedge\ pt \in (\mathsf{pts\_of}\ ss\_of\_tm\ g)\\
\quad \wedge\ (\mathsf{root}\ pt = sym)\\
\quad \wedge\ (\mathsf{substring\_of}\ pt = \mathsf{SOME}\ s\_pt)\\
\quad \longrightarrow\\
\quad \mathsf{matches}\ pt\ pt' \wedge \mathsf{MEM}\ (pt', s\_rem)\ (p\_sym\ (\mathsf{toinput}\ s\_tot))
\end{array}}$$

Theorem 6 (prefix_complete_complete_thm). *If p is prefix-complete, then simple_parser_of p is complete.*

$$\boxed{\begin{array}{l}
\mathsf{prefix\_complete\_complete\_thm} = \forall\ ss\_of\_tm.\ \forall\ g.\ \forall\ sym.\ \forall\ p.\\
\quad \mathsf{prefix\_complete}\ ss\_of\_tm\ g\ sym\ p \longrightarrow \mathsf{complete}\ ss\_of\_tm\ g\ sym\ (\mathsf{simple\_parser\_of}\ p)
\end{array}}$$

9 Parser Generator Completeness

Theorem 7 (main_thm). *A parser p for symbol sym generated by* grammar_to_parser *is complete for prefixes s_pt of the input, in the sense that p returns all parse trees pt that are admitted by the context lc.*

main_thm $= \forall\ p\_of\_tm.\ \forall\ g.\ \forall\ pt.\ \forall\ sym.\ \forall\ s\_pt.\ \forall\ s\_rem.\ \forall\ s\_tot.\ \forall\ lc.$
  let $p =$ grammar_to_parser $p\_of\_tm\ g\ sym$ in
  let $ss\_of\_tm =$ ss_of_tm_of p_of_tm in
  wf_p_of_tm p_of_tm
  $\wedge$ wf_grammar g
  $\wedge$ wf_parse_tree pt
  $\wedge$ $pt \in$ (pts_of $ss\_of\_tm\ g$)
  $\wedge$ (root $pt = sym$)
  $\wedge$ (substring_of $pt =$ SOME s_pt)
  $\wedge$ (concatenate_two $s\_pt\ s\_rem =$ SOME s_tot)
  $\wedge$ admits $lc\ pt$
  $\longrightarrow$
  MEM (pt, s_rem) $(p\ \langle$ lc $= lc$; sb $= s\_tot\ \rangle)$

Proof. The proof is by an outer induction on the size of pt, with an inner structural induction on the list of immediate subtrees of pt.

A parser is initially called with an empty parsing context ie input i is such that i.lc $= []$. We can use admits_thm to change the assumption admits $lc\ pt$ in the statement of main_thm to the assumption good_tree pt. We can further use good_tree_exists_thm to give the following corollary to the main theorem:

Corollary 1. *Parsers generated by* grammar_to_parser *are prefix-complete.*

corollary $= \forall\ p\_of\_tm.\ \forall\ g.\ \forall\ sym.$
  let $ss\_of\_tm =$ ss_of_tm_of p_of_tm in
  let $p =$ grammar_to_parser $p\_of\_tm\ g\ sym$ in
  wf_p_of_tm p_of_tm $\wedge$ wf_grammar g $\longrightarrow$ prefix_complete $ss\_of\_tm\ g\ sym\ p$

This combined with prefix_complete_complete_thm gives:

Theorem 8 (complete_grammar_to_parser_thm). *Parsers generated by* grammar_to_parser *are complete when transformed into simple parsers.*

complete_grammar_to_parser_thm $= \forall\ p\_of\_tm.\ \forall\ g.\ \forall\ sym.$
  let $ss\_of\_tm =$ ss_of_tm_of p_of_tm in
  let $p =$ grammar_to_parser $p\_of\_tm\ g\ sym$ in
  wf_p_of_tm p_of_tm $\wedge$ wf_grammar g
  $\longrightarrow$ complete $ss\_of\_tm\ g\ sym$ (simple_parser_of p)

10 Implementation Issues

Code extraction The HOL4 definitions required for grammar_to_parser are executable within HOL4 itself, using either basic term rewriting or the more efficient strategies embodied in `EVAL_CONV`. We should expect that evaluating code inside

a theorem prover is relatively slow compared to interpreting similar code using one of the usual functional languages (OCaml, SML, Haskell). Our OCaml implementations are manually extracted from the HOL4 definitions. An alternative is to use the HOL4 code extraction facilities which involves pretty-printing the HOL4 definitions as OCaml, but it is not clear that this step preserves soundness (the problem is HOL4 type definitions and their relation to ML type definitions).
Terminal parsers Terminal parsers are required to satisfy a well-formedness requirement, but are otherwise more-or-less arbitrary. The OCaml implementation includes several common terminal parsers that arise in practice. For example, the function `parse_AZS` is a terminal parser that parses a sequence of capital letters. Verification of these terminal parsers is left for future work.
Parsing a grammar specification The parser generator is parameterized by a grammar g (a list of rules). However, grammars are typically written concretely using BNF syntax which must itself be parsed. We therefore define the following syntax of 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? "?"
```

Implementing a parser for this grammar is straightforward. The top-level parser for `RULES` returns a grammar. To turn the grammar into a parser, we use the parser generator in Fig. 3.
Memoization For efficient implementations it is necessary to use memoization on the function `grammar_to_parser`. Memoization takes account of two observations concerning the argument `i`. First, as mentioned previously, the context `i.lc` is implemented as a list but is used as a set. Therefore care must be taken to ensure that permutations of the context are treated as equivalent during memoization. The simplest approach is to impose an order on elements in `i.lc` and ensure that `i.lc` is always kept in sorted order. Second, the only elements `(nt,s)` in `i.lc` that affect execution are those where `s = i.sb`. Thus, before memoization, we discard all elements in `i.lc` where this is not the case. For future work it should be straightforward to add the memoization table as an extra argument to grammar_to_parser and then prove correctness.
Theoretical performance Many grammars generate an exponential number of good parse trees in terms of the size of the input string. Any parser that returns all such parse trees must presumably take an exponential amount of time to do so. However, several parsing techniques claim to be able to parse arbitrary context-free grammars in sub-exponential time. In fact, these parsing techniques do not

return parse trees, but instead return a "compact representation" of all parse trees in polynomial time, from which a possibly infinite number of actual parse trees can be further constructed. The compact representation records which symbols could be parsed for which parts of the input: it is, in effect, a list of pairs, where each pair consists of a symbol and a substring. If we modify our parsers so that they return a dummy value instead of parse trees, then the memoization table is itself a form of compact representation. If we further assume that terminal parsers execute in constant time, then the time complexity of our algorithm is $O(n^5)$ in the length of the input, since there are $O(n^2)$ substrings, each appearing as input in at most $O(n^2)$ calls to the parser, each of which takes time $O(n)$ to execute[5]. Absolute real-world performance is better than this would suggest, because most calls to the parser simply involve looking up pre-existing values in the memoization table, and so execute very quickly.

Real-world performance Roughly speaking, the larger the class of grammar that a parsing technique can handle, the worse the performance. For example, Packrat parsing [5] takes time linear in the size of the input, but cannot deal with even simple non-ambiguous grammars such as `S -> "x" S "x" | "x"`. Of the three existing verified parsers, only the Packrat-based TRX parser [11] has any performance data, a comparison with the purpose-built Aurochs XML parser and the similar xml-light: as expected TRX is significantly slower. Preliminary testing using a simple XML grammar indicates that our parsers are competitive: an unmemoized version of our algorithm can parse a 1.4MB XML file in 0.31 seconds (better than Aurochs, and slightly worse than xml-light). More importantly, our algorithm is linear time in the size of the input. Aurochs and xml-light are purpose built XML parsers, and TRX does not handle all context-free grammars, however, there are some techniques such as GLR parsing that can handle arbitrary context-free grammars. There are very few implementations, but the popular Happy parser generator [1] is one such. Executing a compiled version of our memoized parser generator (which interprets the grammar) and comparing the performance with a compiled version of a parser produced by Happy in GLR mode (where the parser code directly encodes the grammar) on the grammar `E -> E E E | "1" |` ϵ, with input a string consisting solely of `1`s, gives the following figures. Noticeably, the longer the input, the better our parsers perform relative to Happy parsers. In fact, parsers generated by Happy in GLR mode appear to be $O(n^6)$ although GLR is theoretically $O(n^3)$ in the worst case. We leave investigation of this discrepancy, and further real-world performance analysis and tuning, to future work.

Input size/# characters	Happy parse time/s	Our parse time/s	Factor
20	0.19	0.11	1.73
40	9.53	3.52	2.71
60	123.34	30.46	4.05

[5] The time complexity is not obvious, and was informed by careful examination of real-world execution traces. For comparison, the time complexity of Earley parsers, CYK parsers, and GLR parsers is $O(n^3)$.

11 Related work

A large amount of valuable research has been done in the area of parsing. We cannot survey the entire field here, but instead aim to give references to work that is most closely related to our own. A more complete set of references is contained in our previous work [15].

The first parsing techniques that can handle arbitrary context-free grammars are based on dynamic programming. Examples include CYK parsing [10] and Earley parsing [4]. In these early works, the emphasis is on implementation concerns, and in particular completeness is often not clear. For example [16] notes that Earley parsing is not correct for rules involving ϵ. Later the approach in [16] was also found to be incorrect. However, it is *in principle* clear that *variants* of these approaches can be proved complete for arbitrary context-free grammars. Combinator parsing and related techniques are probably folklore. An early approach with some similarities is [14]. Versions that are clearly related to the approach taken in this paper were popularized in [9].

The first approach to use the length of the input to force termination is [12]. The work most closely related to ours is that of Frost et al. [8, 6, 7], who limit the depth of recursion to $m * (1 + |s|)$, where m is the number of nonterminals in the grammar and $|s|$ is the length of the input. They leave correctness of their approach as an open question. For example, they state: "Future work includes proof of correctness ..." [7]; and "We are constructing formal correctness proofs ..." [8]. A major contribution of this paper, and the key to correctness, is the introduction of parsing context and the definition of `good_tree`. Amazingly, the measure function from Sect. 7 gives the same worst-case limit on the depth of recursion as that used by Frost et al. (although typically our measure function decreases faster because it takes the context into account), and so *this work can be taken as proof that the basic approach of Frost et al. is correct.*

The mechanical verification of parsers, as here, is a relatively recent development. Current impressive examples such as [2, 11, 3] cannot handle all context-free grammars. Recent impressive work on verified compilation such as [13] is complementary to the work presented here: our verified parser can extend the guarantees of verified compilation to the front-end parsing phase.

12 Conclusion

We presented a parser generator for arbitrary context-free grammars, based on combinator parsing. The code for a minimal version of the parser generator is about 20 lines of OCaml. We proved that generated parsers are terminating, sound and complete using the HOL4 theorem prover. The time complexity of the memoized version of our algorithm is $O(n^5)$. Real-world performance comparisons on the grammar `E -> E E E | "1" |` ϵ indicate that we are faster than the popular Happy parser generator running in GLR mode across a wide range of inputs.

There is much scope for future work, some of which we have mentioned previously. One option is to attempt to reduce the worst case time complexity from $O(n^5)$. In an ideal world this could be done whilst preserving the essential beauty and simplicity of combinator parsing; in reality, it may not be possible to reduce the time complexity further without significantly complicating the underlying implementation.

References

1. Happy, a parser generator for Haskell. `http://www.haskell.org/happy/`.
2. Aditi Barthwal and Michael Norrish. Verified, executable parsing. In Giuseppe Castagna, editor, *ESOP*, volume 5502 of *Lecture Notes in Computer Science*, pages 160–174. Springer, 2009.
3. Nils Anders Danielsson. Total parser combinators. In Paul Hudak and Stephanie Weirich, editors, *ICFP*, pages 285–296. ACM, 2010.
4. Jay Earley. An efficient context-free parsing algorithm. *Commun. ACM*, 13(2):94–102, 1970.
5. Bryan 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.
6. Richard A. Frost, Rahmatullah Hafiz, and Paul Callaghan. Parser combinators for ambiguous left-recursive grammars. In Paul Hudak and David Scott Warren, editors, *PADL*, volume 4902 of *Lecture Notes in Computer Science*, pages 167–181. Springer, 2008.
7. Richard A. Frost, Rahmatullah Hafiz, and Paul 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.
8. Rahmatullah Hafiz and Richard A. Frost. Lazy combinators for executable specifications of general attribute grammars. In Manuel Carro and Ricardo Peña, editors, *PADL*, volume 5937 of *Lecture Notes in Computer Science*, pages 167–182. Springer, 2010.
9. Graham Hutton. Higher-order functions for parsing. *J. Funct. Program.*, 2(3):323–343, 1992.
10. 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.
11. Adam Koprowski and Henri Binsztok. TRX: A formally verified parser interpreter. In Andrew D. Gordon, editor, *ESOP*, volume 6012 of *Lecture Notes in Computer Science*, pages 345–365. Springer, 2010.
12. Susumu Kuno. The predictive analyzer and a path elimination technique. *Commun. ACM*, 8(7):453–462, 1965.
13. Xavier Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, April 2009.
14. V. R. Pratt. Top down operator precedence. In *Proceedings ACM Symposium on Principles Prog. Languages*, 1973.
15. Tom Ridge. Simple, functional, sound and complete parsing for all context-free grammars, 2010. Unpublished draft available at `http://www.cs.le.ac.uk/~tr61`.
16. Masaru Tomita. *Efficient Parsing for Natural Language: A Fast Algorithm for Practical Systems.* Kluwer, Boston, 1986.