Simple, efficient, sound and complete combinator parsing for all context-free grammars, using an oracle

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. Traditionally parser combinators have struggled to handle all features of context-free grammars, such as left recursion.
Previous work introduced novel parser combinators that could be used to parse all context-free grammars. A parser generator built using these combinators was proved both *sound* and *complete* in the HOL4 theorem prover. Unfortunately the performance was not as good as other parsing methods such as Earley parsing.
In this paper, we build on this previous work, and combine it in novel ways with existing parsing techniques such as Earley parsing. The result is a sound-and-complete combinator parsing library that *can handle all context-free grammars*, and *has good performance*.

1 Introduction

In previous work [13] the current author introduced novel parser combinators that could be used to parse all context-free grammars. For example, a parser for the grammar `E -> E E E | "1" |` ϵ can be written in OCaml as:

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

In [4] Barthwal and Norrish discuss this work:

> [Ridge] presents a verified parser for all possible context-free grammars, using an admirably simple algorithm. The drawback is that, as presented, the algorithm is of complexity $O(n^5)$.

Existing techniques such as Earley parsing [5] take time $O(n^3)$ in the length of the input in the worst case. Therefore, as far as performance is concerned, [13] is not competitive with such techniques. In this work, we seek to address these performance problems. We have three main goals for our parsing library.

- The library should provide an interface based on parser combinators.

– The library should handle all context-free grammars.
– The library should have "good" performance.

The challenge is to improve on our previous work by providing Earley-like performance: $O(n^3)$ in the worst case but typically much better on common classes of grammar. Our main contribution is to show how to combine a combinator parsing interface with an efficient general parsing algorithm such as Earley parsing. We list further contributions in Section 11. We now briefly outline our new approach, and then give an overview of the rest of the paper.

Consider the problem of parsing an input string s, given a grammar Γ (a finite set of rules) and a nonterminal start symbol S. In general, we will work with substrings $s_{i,j}$ of the input s between a low index i and a high index j, where $i \leq j$. In symbols we might write the parsing problem as $\Gamma \vdash S \rightarrow^* s_{i,j}$. Suppose the grammar contains the rule $S \rightarrow A\ B$. Then one way to derive $\Gamma \vdash S \rightarrow^* s_{i,j}$ is to derive $\Gamma \vdash A \rightarrow^* s_{i,k}$ and $\Gamma \vdash B \rightarrow^* s_{k,j}$:

$$\frac{\Gamma \vdash A \rightarrow^* s_{i,k} \qquad \Gamma \vdash B \rightarrow^* s_{k,j}}{\Gamma \vdash S \rightarrow^* s_{i,j}}\ (S \rightarrow A\ B) \in \Gamma$$

This rule resembles the well-known *Cut* rule of logic, in that it introduces an unknown k in the search for a derivation. The problem is that there is no immediate way to determine the possible values of k when working from the conclusion of the rule to the premises. Put another way, a top-down parse of the substring $s_{i,j}$ must divide the substring into two substrings $s_{i,k}$ and $s_{k,j}$, but there is no information available to determine the possible values of k. Attempting to parse for all k such that $i \leq k \leq j$ results in poor real-world performance.

The traditional combinator parsing solution is to parse *prefixes* of the substring $s_{i,j}$. Since $s_{i,j}$ is trivially a prefix of itself, a solution to this more general problem furnishes a solution to the original. Moreover, this approach gives possible candidates for the value k: We first attempt to find all parses for nonterminal A for prefixes of input $s_{i,j}$; the results will be derivations for $s_{i,k}$ where $k \leq j$. We can then attempt to parse nonterminal B for prefixes of $s_{k,j}$, since possible values of k are now known.

We propose a different solution: assume the existence of an oracle that can provide the unknown values of k. As we show later, this allows one to solve the problem of parsing context-free grammars using combinator parsing. However, in the real world we must also provide some means to construct the oracle. Our answer is simple: use some other parsing technique, preferably one that has good performance. In this work we use Earley parsing, but *any* other general parsing technique would suffice.

There are several technical problems that must be addressed. For example, to handle arbitrary grammars, including features such as left-recursion, we adapt the notion of parsing contexts originally introduced in [13]. A central new challenge is to reconcile the implementation of Earley parsing with that of com-

binator parsers. For example, consider the following parser[1] for the grammar `E -> E E E | "1" | ϵ`.

```
let rec parse_E = (fun i → mkntparser "E" (
  ((parse_E ⊗ parse_E ⊗ parse_E) ≫ (fun (x,(y,z)) → x+y+z))
  ⊕ (a1 ≫ (fun _ → 1)) ⊕ (eps ≫ (fun _ → 0))) i)
```

This parser uses parsing actions to count the length of the parsed input. The parsing code *implicitly* embodies the grammar. However, implementations of Earley parsing require *explicit* representations of the grammar, such as:

```
let g = [("E",[NT "E";NT "E";NT "E"]); ("E",[TM "1"]); ("E",[TM "eps"])]
```

In this representation of the grammar (a finite set of rules, here represented using a list), rules are pairs, where the left-hand side is a nonterminal (identified by a string) and the right-hand side is a list of symbols, either nonterminal symbols such as `NT "E"` or terminal symbols such as `TM "eps"`.

Our solution to this challenge requires interpreting the parsing combinators in three different ways. The first interpretation embeds a symbol with a given parser. With this we can define a function *sym_of_parser* which takes a parser as an argument and returns the associated symbol. For example, *sym_of_parser parse_E* evaluates to `NT "E"`. The second interpretation builds on the first to associate a concrete representation of the grammar with each parser. With this we can define a function *grammar_of_parser* which takes a parser as an argument and returns the associated grammar. For example, evaluating *grammar_of_parser parse_E* returns a record with a field whose value is the following[2]:

```
[("(E*E)", Seq (NT "E", NT "E")); ("(E*(E*E))", Seq (NT "E", NT "(E*E)"));
("((E*(E*E))+1)", Alt (NT "(E*(E*E))", TM "1"));
("(((E*(E*E))+1)+eps)",Alt(NT "((E*(E*E))+1)",TM "eps"));
("E", Atom (NT "(((E*(E*E))+1)+eps)"))]
```

This is a *binarized* version of the previous grammar. Note that nonterminals now have unusual names, such as `(E*E)`. Right-hand sides are either atoms, binary sequences (of symbols, not nonterminals cf. Chomsky Normal Form), or binary alternatives. The function *grammar_of_parser* allows us to inspect the structure of the parser, in order to extract a grammar, which can then be fed to an Earley parser. The Earley parser takes the grammar, and a start symbol, and parses the input string s. The output from the Earley parsing phase can be thought of as a list of Earley productions of the form $(X \to \alpha.\beta, i, j, l)$. Here X

[1] In the following sections we have lightly typeset the OCaml code. The sequencing combinator `***>` is written $\otimes$ and associates to the right; the alternative combinator `||||` is written $\oplus$; and the action function `>>>>` is written $\gg$. The notation $s.[i]$ denotes the i th character of the string s. Records with named fields are written e.g. $\langle f1 = v1;\ f2 = v2\rangle$. Functional record update is written $\langle r$ with $f1 = v1;\ f2 = v2\rangle$. Otherwise the OCaml syntax we use should be readily understandable by anyone familiar with functional programming.

[2] A second field records the terminal parsers that are used, such as *a1* and *eps*.

is a nonterminal, α and β are sequences of symbols (β is non-empty), and i, j, l are integers. The meaning of such a production is that there is a rule $X \to \alpha\ \beta$ in the grammar such that the substring $s_{i,j}$ could be parsed as the sequence α, and moreover the substring $s_{j,l}$ could be parsed as the sequence β. These productions can be used to construct an oracle.

The oracle is designed to answer the following question: given a grammar Γ, a rule $S \to A\ B$ in Γ, and a substring $s_{i,j}$, what are the possible values of k such that $\Gamma \vdash A \to^* s_{i,k}$ and $\Gamma \vdash B \to^* s_{k,j}$? To determine the values of k we look for Earley productions of the form $(S \to A.B, i, k, j)$. Such a production says exactly that the substring $s_{i,j}$ could be parsed as the sequence $A\ B$ and that $s_{i,k}$ could be parsed as A and $s_{k,j}$ could be parsed as B.

The third interpretation of the parsing combinators follows the traditional interpretation, except that *we do not parse prefixes*, but instead *we use the oracle* to determine where to split the input string during a parse. In fact, all necessary *parsing* information has already been deduced from the input s during the Earley phase, so this phase degenerates into using the oracle to apply *parsing actions* appropriately, in the familiar top-down recursive manner. During this phase we make use of a parsing context to handle features such as left recursion, and memoization for efficiency.

In outline, our algorithm cleanly decomposes into 3 phases. Given a parser p and an input string s we perform the following steps.

1. Extract grammar Γ and start symbol S from the parser p and feed Γ, S and s to the Earley parser, which performs a traditional Earley parse.
2. Take the Earley productions that result and construct the oracle.
3. Use the oracle to guide the action phase.

Earley parsing is theoretically efficient $O(n^3)$ and performs well in practice. The construction of the oracle involves processing the Earley productions, which have the same bound as the Earley parser itself, $O(n^3)$. Parsing actions involve arbitrary user-supplied code, so it is not possible to give an *a priori* bound on the time taken during the action phase, however, in Section 9 we argue that the performance of this stage is close to optimal. Thus, we argue that our approach overall results in close-to-optimal (i.e. "good") asymptotic performance. In Section 9 we also provide real-world evidence to support these claims.

In this paper we present a version of our code, called mini-P3, that focuses on clarity for expository purposes, whilst preserving all important features. The full P3 code follows exactly the structure we outline here with only minor differences[3]. Our implementation language is a small subset of OCaml, essentially the simply-typed lambda calculus with integers, strings, recursive functions, records and datatypes. Apart from memoization, the code is purely functional. It should be very easy to re-implement our approach in other functional languages such as Haskell, Scheme and F♯. The full code for mini-P3 and P3 can be found in the online resources at `http://www.tom-ridge.com/p3.html`.

[3] Footnotes describe how mini-P3 differs from P3.

The structure of the rest of the paper is as follows. In Section 2 we give two key examples, and discuss some common misunderstandings concerning our approach. In Section 3 we introduce the basic types such as those for substrings and grammars, and discuss the types related to the parser combinators. The subsequent sections modularly introduce different aspects of our approach. We start by defining the sequencing and alternative combinators in Section 4. In Section 5 we introduce our running example, which we develop further in Section 7. In Section 6 we describe the Earley parsing phase and the construction of the oracle. In Section 8 we discuss the role of parsing context and the use of memoization to make the action phase efficient. In Section 9 we report on various experiments to measure performance. In Section 10 we discuss related work, and in Section 11 we conclude.

An extended version of this paper appears in the online resources. This includes further sections discussing motivation, mathematical preliminaries, further examples, parsing context, memoization and soundness and completeness. For space reasons this material cannot be included here.

2 Example

We introduce some example parsers to illustrate our approach, and clarify aspects of our approach that are commonly misunderstood. An efficient parser for the grammar `E -> E E E | "1" | ϵ` is:

$$\begin{array}{l}\mathsf{let}\ \mathit{tbl}\ =\ \mathsf{Hashtbl}.\mathit{create}\ 0\\ \mathsf{let\ rec}\ \mathit{parse\_E}\ =\ (\mathsf{fun}\ i \to \mathit{memo\_p3}\ \mathit{tbl}\ (\mathit{mkntparser}\ \texttt{"E"}\ (\\ \quad((\mathit{parse\_E} \otimes \mathit{parse\_E} \otimes \mathit{parse\_E}) \gg (\mathsf{fun}\ (x,(y,z)) \to \mathsf{NODE}(x,y,z)))\\ \quad\oplus\ (\mathit{a1} \gg (\mathsf{fun}\ \_ \to \mathsf{LEAF}(1))) \oplus (\mathit{eps} \gg (\mathsf{fun}\ \_ \to \mathsf{LEAF}(0))))))\ i)\end{array}$$

Our approach is complete in that it returns all "good"[4] parse trees. There are an exponential number of such parse trees. For example, for input length 19, there are more than $4 * 10^{17}$ parse trees, but as with most exponential behaviours it is not feasible to actually compute all these parse trees. The following parser is identical except that, rather than returning parse trees, it computes (in all possible ways) the length of the input parsed:

$$\begin{array}{l}\mathsf{let}\ \mathit{tbl}\ =\ \mathsf{Hashtbl}.\mathit{create}\ 0\\ \mathsf{let\ rec}\ \mathit{parse\_E}\ =\ (\mathsf{fun}\ i \to \mathit{memo\_p3}\ \mathit{tbl}\ (\mathit{mkntparser}\ \texttt{"E"}\ (\\ \quad((\mathit{parse\_E} \otimes \mathit{parse\_E} \otimes \mathit{parse\_E}) \gg (\mathsf{fun}\ (x,(y,z)) \to x+y+z))\\ \quad\oplus\ (\mathit{a1} \gg (\mathsf{fun}\ \_ \to 1)) \oplus (\mathit{eps} \gg (\mathsf{fun}\ \_ \to 0))))\ i)\end{array}$$

Naively we might expect that this also exhibits exponential behaviour, since presumably the parse trees must all be generated, and the actions applied. *This expectation is wrong.* Running this example parser on an input of size 19 returns in 0.02 seconds with a single result 19. For an input of size 100, this parser returns a single result 100 in 5 seconds, and over a range of inputs this parser exhibits polynomial behaviour rather than exponential behaviour. As far as we

[4] The notion of "good" parse tree is defined in [13].

```
type (α,β) fmap = (α × β) list    type substring = SS of string × int × int
type term = string    type nonterm = string    type symbol = NT of nonterm | TM of term

type rhs = Atom of symbol | Seq of symbol × symbol | Alt of symbol × symbol
type parse_rule = nonterm × rhs    type grammar = parse_rule list

type raw_parser = substring → substring list
type ty_oracle = (symbol × symbol) → substring → int list
type local_context = LC of (nonterm × substring) list

let empty_fmap = []    let empty_context = (LC [])
let empty_oracle = (fun (sym1, sym2) → fun ss → [])
```

Fig. 1. Basic types and trivial values

are aware, *no other parser can handle such examples*. To make such examples possible requires: careful engineering of the backend parser (here based on Earley parsing) so that it is $O(n^3)$ in the length of the input; a *compact* representation of parse results (using an oracle) that does not require more than $O(n^3)$ time to construct; a semantically-meaningful notion of action when there are an infinite number of possible parse trees (handled by the parsing context); careful use of the oracle to guide the action phase; and memoization during the action phase so that exponentially many possible actions are reduced to a polynomial number of actual actions. The code above combines all of these aspects whilst presenting a standard combinator-parsing interface to the programmer. In the rest of the paper we discuss the techniques and careful engineering that make this possible.

3 Types

Basic types In Fig. 1 we give types for finite maps (represented by association lists), substrings, terminals, nonterminals, symbols, the right-hand sides of parse rules, parse rules, and grammars. Note that the `rhs` type permits only unary rules (e.g. `E -> F`) and binary rules (e.g. sequences `E -> A B` or alternatives `E -> A | B`). This is a restriction on the *internal representation* of the rules and not on the user of the library.

Raw parsers capture the set of substrings associated to a given terminal. They can be more-or-less arbitrary OCaml code[5]. Given a substring $\mathsf{SS}(s, i, j)$, a raw parser returns a list of substrings $\mathsf{SS}(s, i, k)$ indicating that the prefix $\mathsf{SS}(s, i, k)$ could be parsed as the corresponding terminal. For example, the raw parser *raw_a1* consumes a single `1` character from the input:

```
let raw_a1 (SS(s,i,j)) = (if i < j && s.[i] = '1' then [SS(s,i,i+1)] else [])
```

[5] A raw parser should behave as a pure function, and should return prefixes of its argument. For a fully formal treatment of the parsers associated with terminals see [13].

```
type (α, β, γ) sum3 = Inl of α | Inm of β | Inr of γ
type inl = unit    type outl = symbol

type mid = ⟨ rules : parse_rule list; tmparsers : (term, raw_parser) fmap ⟩
type inm = mid    type outm = mid

type inr = ⟨ ss : substring; lc : local_context; oracle : ty_oracle ⟩
type α outr = α list

type input = (inl, inm, inr) sum3    type α output = (outl, outm, α outr) sum3
type α parser3 = (input → α output)

let empty_mid = ⟨rules = []; tmparsers = empty_fmap⟩
```

Fig. 2. Parser types and trivial values

The oracle type captures the idea that an oracle takes two symbols *sym1*, *sym2*, and a substring $\mathsf{SS}(s, i, j)$, and returns those integers k such that $\mathsf{SS}(s, i, k)$ can be parsed as *sym1*, and $\mathsf{SS}(s, k, j)$ can be parsed as *sym2*. Finally, the type local_context represents the parsing context, see Section 8.

Parser types The types related to parsers are given in Fig. 2. In our approach, a parser should be viewed as a collection of three separate functions[6]. We first discuss the sum3 type, and the function *sum3* which converts three separate functions to a single function, and the function *unsum3* which converts a single function of the appropriate form to three separate functions. Following this, we discuss the particular instances of the sum3 type that we use for our parsers.

The sum3 type The sum3 type generalizes the familiar binary sum to three components. Given three functions of type $\alpha \to \delta$, $\beta \to \epsilon$ and $\gamma \to \zeta$, we can form a composite function of type (α, β, γ) sum3 $\to (\delta, \epsilon, \zeta)$ sum3. We can define this composite function explicitly, and moreover define an inverse:

```
let dest_inl (Inl x) = x ...

let sum3 (f, g, h) = (fun i → match i with
  | Inl l → Inl(f l) | Inm m → Inm(g m) | Inr r → Inr(h r))

let unsum3 u = (
  let f = (fun x → dest_inl (u (Inl x))) in
  let g = (fun x → dest_inm (u (Inm x))) in
  let h = (fun x → dest_inr (u (Inr x))) in
  (f, g, h))
```

We use the functions *sum3* and *unsum3* extensively when defining the parser combinators. In particular, as a function from inputs to outputs, a parser satisfies

[6] This implementation of the combinators is just one of those we have experimented with, and alternatives are certainly possible.

the extra conditions (not explicit in the type): given an argument of the form Inl *x*, the parser produces a result of the form Inl *x'*, and similarly for Inm and Inr. Parsers *p* of type input $\rightarrow \alpha$ output should be thought of as the sum of three functions, i.e. $p = sum3\ (f, g, h)$.

Left component, extracting a symbol from a parser The left component of a parser consists of a function of type inl $\rightarrow$ outl, that is, from unit to symbol. If *parse_E* is a parser for the nonterminal E, then the expression *parse_E* (Inl ()) should evaluate to Inl (NT "E"). We define the following auxiliary function:

let *sym_of_parser p* = (*dest_inl* (*p* (Inl ())))

Middle component, extracting a grammar from a parser The middle component of a parser consists of a function of type inm $\rightarrow$ outm, where inm and outm are both equal to type mid. The middle component of a parser is therefore of type mid $\rightarrow$ mid. The mid type represents the grammar associated with a parser. The middle component of a parser such as *parse_E* is a grammar transformer, that takes a grammar and extends it with extra rules. The type mid is a record type with two fields. The first is a list of parse rules. The second is a finite map from terminals to raw parsers. If *parse_E* is a parser for the nonterminal E, then the expression *parse_E* (Inm *m*) will evaluate to a value of the form Inm *m'*, where *m'* is *m* augmented with rules for the nonterminal E (and all nonterminals reachable from E), and the terminal parsers involved in the definition of *parse_E* (and all terminal parsers involved in the definition of nonterminals reachable from E). We can then define *grammar_of_parser*:

let *grammar_of_parser p* = (*dest_inm* (*p* (Inm *empty_mid*)))

Right component, recursive descent parser The right component is a function of type inr $\rightarrow \alpha$ outr, where α outr = α list. This resembles the traditional type of a combinator parser: a function from a string to a list of possible values. We work with substrings rather than strings, so an input *i* of type inr contains a component *i.ss* of type substring. Two additional fields are present: *i.oracle* is an oracle that indicates how to split the input when parsing a sequence of symbols, and *i.lc* is a parsing context that allows combinator parsers to handle all context-free grammars. We discuss these additional fields further in the following sections. The output type α outr is simply a list of values at an arbitrary type α.

4 Parsing combinators

In the previous section we discussed the α parser3 type and related types. In this section we give the definition of the sequencing combinator *p1* $\otimes$ *p2*. The definition of the alternative combinator *p1* $\oplus$ *p2* follows the sequencing combinator *mutatis mutandis*. The following section illustrates the use of these combinators on a simple example.

Consider the left component of the sequencing combinator. This takes two parsers *p1* and *p2* and produces the left component (a function from unit to symbol) of the parser *p1* $\otimes$ *p2*:

```
let seql p1 p2 = (fun () → let (f1, _, _) = unsum3 p1 in
  let (f2, _, _) = unsum3 p2 in let rhs = Seq(f1 (), f2 ()) in mk_symbol rhs)
```

The left component is a function from unit argument () to a symbol representing the sequential combination of the two underlying parsers. We use the auxiliary function *mk_symbol* to generate new symbols for possible right hand sides. These new symbols are always nonterminals. The requirement on *mk_symbol* is simply that it should be injective on its argument: if *mk_symbol rhs'* = *mk_symbol rhs* then *rhs'* = *rhs*[7]. For example, with the current implementation, evaluating *mk_symbol* (Seq(NT "E", NT "E")) returns (NT "(E*E)") [8].

The middle component for the combination *p1* ⊗ *p2*, of type mid → mid, transforms a list of rules by adding a new rule representing the sequencing of *p1* and *p2*. It should also call the underlying parsers so that they in turn add their rules.

```
let seqm p1 p2 = (fun m → let NT nt = seql p1 p2 () in
  if List.mem nt (List.map fst m.rules) then m else (
    let (f1, g1, _) = unsum3 p1 in
    let (f2, g2, _) = unsum3 p2 in
    let new_rule = (nt, Seq(f1 (), f2 ())) in
    let m1 = ⟨ m with rules = (new_rule :: m.rules) ⟩ in
    let m2 = g1 m1 in let m3 = g2 m2 in m3))
```

Note that the code first checks whether the nonterminal *nt* corresponding to *p1* ⊗ *p2* is already present in the rules. If so, this nonterminal has already been processed, and there is no need to continue further. This check also prevents nontermination of *seqm* when dealing with recursive grammars. If the nonterminal is not present, then the new rule is constructed, added to the list of rules, and then the middle components *g1* and *g2* of the parsers *p1* and *p2* are invoked in turn, to add their rules.

The right component of the sequencing combinator takes two parsers *p1* of type α parser3, and *p2* of type β parser3, and produces the right component of the parser *p1* ⊗ *p2*, of type inr → ($\alpha \times \beta$) outr.

---

[7] Related to this is the requirement that users do not annotate two *different* parsers with the *same* nonterminal; the following must be avoided:

```
let rec parse_E = (fun i → mkntparser "E" ... i)
and parse_F = (fun i → mkntparser "E" ... i)
```

There seems no way to enforce this constraint using types. An alternative is to use a *gensym*-like technique to construct arguments to *mkntparser* automatically. This ensures uniqueness of names, but requires non-purely-functional techniques.

[8] Generated names should not clash with user names. The traditional solution is to incorporate a "forbidden" character, not available to users, into generated names. A better approach would use a more structured datatype than strings for the names of nonterminals. For simplicity, we stick with strings and assume the user does not use symbols such as * in the names of nonterminals.

```
let seqr p1 p2 = (fun i0 →
  let sym1 = sym_of_parser p1 in let sym2 = sym_of_parser p2 in
  let ks = i0.oracle (sym1, sym2) i0.ss in
  let SS(s, i, j) = i0.ss in
  let f1 k = (
    let rs1 = dest_inr (p1 (Inr ⟨ i0 with ss = (SS(s, i, k)) ⟩)) in
    let rs2 = dest_inr (p2 (Inr ⟨ i0 with ss = (SS(s, k, j)) ⟩)) in
    list_product rs1 rs2) in
  List.concat (List.map f1 ks))
```

The function *seqr* first determines the symbols *sym1* and *sym2* corresponding to the two underlying parsers. It then calls the oracle with the appropriate symbols and substring $i0.ss = \mathsf{SS}(s, i, j)$. The resulting values for k are bound to the variable *ks*. For each of these values k, parser *p1* is called on the substring $\mathsf{SS}(s, i, k)$ and *p2* is called on the substring $\mathsf{SS}(s, k, j)$. The results are combined using the library functions *list_product* (which takes two lists and forms a list of pairs) and List.*concat*. The corresponding right component *altr* for the alternative combinator is much simpler: as with traditional combinator parsers, the results of the parsers *p1* and *p2* are simply appended.

We can now define the sequential combination $p1 \otimes p2$. This uses *seql*, *seqm* and seqr to construct a new parser of type $(\alpha \times \beta)$ parser3 from a parser *p1* of type α parser3 and a parser *p2* of type β parser3.

```
let p1 ⊗ p2 = (fun i0 → let f = seql p1 p2 in
  let g = seqm p1 p2 in let h = seqr p1 p2 in sum3 (f, g, h) i0)
```

The alternative combination $p1 \oplus p2$ is identical, except that *seql* becomes *altl* and so on. We also define the "semantic action" function, which takes a parser p of type α parser3 and a function f from α to β and returns a parser of type β parser3, by mapping the function f over the list of values in the right component. Apart from the fact that we now have three components to deal with, this is the approach taken by traditional parser combinators.

```
let p ≫ f = (fun i → match i with | Inl _ → (Inl (dest_inl (p i)))
 | Inm _ → (Inm (dest_inm (p i))) | Inr _ → (Inr (List.map f (dest_inr (p i))))))
```

Finally, we turn to the auxiliary function *mkntparser*. This function allows the user to introduce concrete *names* for nonterminals, to label the corresponding code for parsers: let *parse_E* = (fun $i \rightarrow$ *mkntparser* `"E"` ... i). At this stage, we introduce a version of *mkntparser* that does not deal with context. In Section 8 we add the ability to handle context.

```
let mkntparser' nt p = (fun i → match i with
  | Inl () → Inl (NT nt)
  | Inm m → (if List.mem nt (List.map fst m.rules) then Inm m else (
      let sym = sym_of_parser p in
      let new_rule = (nt, Atom sym) in
      p (Inm ⟨ m with rules = (new_rule :: m.rules) ⟩))))
  | Inr r → (let Inr rs = p i in Inr (unique rs)))
```

For the left component, *mkntparser'* simply returns a symbol NT *nt* corresponding to the user supplied label *nt*. For the middle component, the parser *p* has a corresponding symbol *sym*. In terms of the grammar, we should add a new rule $nt \to sym$. Thus, when passed an argument lnm *m* we add this new rule before recursively invoking the underlying parser *p*. The right component is unchanged except that as an optimization we return only unique results.

As well as *mkntparser'*, we have an auxiliary function *mktmparser* whose purpose is similar: to introduce concrete names for terminals. This is necessary because the middle component *m*, as well as accumulating the grammar rules in the field *m.rules*, also accumulates named terminal parsers in the field *m.tmparsers*.

5 Example

We can now define an example parser. At this stage, we have no way to construct an oracle automatically, so we will hand-code this aspect of the parser. In addition, we have not dealt with the parsing context, so we will not be able to handle grammars such as `E -> E E E | "1" |` ϵ. We will make use of the raw parser *raw_a1* from Section 3. First, we define our terminal parser:

$$\mathsf{let}\ a1\ =\ mktmparser\ \texttt{"1"}\ raw\_a1$$

A parser for the grammar `E -> E E E | "1"`, where the actions count the number of `1`s, is:

$$\begin{array}{l}\mathsf{let\ rec}\ parse\_E\ =\ (\mathsf{fun}\ i \to mkntparser'\ \texttt{"E"}\ (\\ \quad ((parse\_E \otimes parse\_E \otimes parse\_E) \gg (\mathsf{fun}\ (x,(y,z)) \to x+y+z))\\ \quad \oplus\ (a1 \gg (\mathsf{fun}\ \_ \to 1)))\ i)\end{array}$$

In order to run our parser on some input, we need to supply an oracle. At this point, we simply hand-code the oracle. The role of the oracle is to determine, given two symbols *sym1*, *sym2*, where to cut an input substring $\mathsf{SS}(s,i,j)$ into two pieces $\mathsf{SS}(s,i,k)$ and $\mathsf{SS}(s,k,j)$, so that the first can be parsed as *sym1* and the second can be parsed as *sym2*.

$$\mathsf{let}\ oracle\ =\ (\mathsf{fun}\ (sym1, sym2) \to \mathsf{fun}\ (\mathsf{SS}(s,i,j)) \to\ \ldots)$$

For *parse_E* there are two uses of the sequencing combinator: one corresponding to the expression $parse\_E \otimes parse\_E$, and one to the first occurence in the expression $parse\_E \otimes (parse\_E \otimes parse\_E)$[9]. The two nonterminals that can occur as arguments to the sequencing combinator are `E` (corresponding to inputs which are non-empty sequences of the character `1`) and `(E*E)` (corresponding to sequences of length at least two). We introduce an auxiliary function *upto'* such that $upto'\ i\ j\ =\ [i+1;\ldots;j-1]$ and code the oracle as:

$$\begin{array}{l}\mathsf{let}\ oracle\ =\ (\mathsf{fun}\ (sym1, sym2) \to \mathsf{fun}\ (\mathsf{SS}(s,i,j)) \to \mathsf{match}\ (sym1, sym2)\ \mathsf{with}\\ \quad |\ (\mathsf{NT}\ \texttt{"E"}, \mathsf{NT}(\texttt{"(E*E)"})) \to (upto'\ i\ (j-1))\\ \quad |\ (\mathsf{NT}\ \texttt{"E"}, \mathsf{NT}(\texttt{"E"})) \to (upto'\ i\ j))\end{array}$$

[9] Recall that the sequencing combinator associates to the right.

We can then run a parser on an input, assuming the existence of the oracle:

```
let run_parser3' oracle p s = (let i0 = ⟨ ss = (SS(s, 0, String.length s));
    lc = empty_context; oracle = oracle ⟩ in
  let rs = dest_inr (p (Inr i0)) in unique rs)
```

This simply evaluates the right component of the parser and returns unique results. We can run the example parser in the OCaml top-level, and OCaml responds with the expected result:

```
let _ = run_parser3' oracle parse_E "1111111"
- : int list = [7]
```

We can also examine the left and middle components of our example parser. Most interesting is the middle component:

```
let m = grammar_of_parser parse_E
val m: mid = ⟨rules = [("(E*E)", Seq (NT "E", NT "E"));
    ("(E*(E*E))", Seq (NT "E", NT "(E*E)"));
    ("((E*(E*E))+1)", Alt (NT "(E*(E*E))", TM "1"));
    ("E", Atom (NT "((E*(E*E))+1)"))];
  tmparsers = [("1", < fun >)]⟩
```

The result is a record m. The *m.rules* field contains a concrete representation of the grammar, with nonterminals corresponding to every use of the sequencing and alternative combinators. In addition, the *m.tmparsers* field represents a finite map from terminals to the corresponding raw parsers. In this example, there is only one entry for the terminal `"1"`.

In this section we have worked through the definition of a simple parser, and seen how the machinery introduced in previous sections allows us to extract a concrete representation of the grammar from code such as *parse_E*. With a concrete representation of the grammar, we can use a method such as Earley parsing to determine the information necessary to construct an oracle, and then finally use the oracle to guide the action phase of the parse.

6 Earley parsing and construction of the oracle

We feed the concrete representation of the grammar, with the input string and start symbol, to an Earley parser. The resulting Earley productions can then be processed to form an oracle. As described in Section 1 an Earley production is of the form $(X \rightarrow \alpha.\beta, i, j, l)$, where $(X \rightarrow \alpha.\beta, i, j)$ is an Earley item, β is non-empty, and l indicates that β could be parsed between input positions j and l. We introduce a function *earley_prods_of_parser* of type α parser3 $\rightarrow$ string $\rightarrow$ production list, which takes a parser and an input and returns a list of productions. We process these productions using a function *oracle_of_prods* of type production list $\rightarrow$ ty_oracle. For a given parser and input, these two functions produce a parsing oracle which we use to guide the action phase. Further details of our approach to Earley parsing are included in the extended version of this paper, available in the online resources.

7 Example, with Earley parsing

We continue the example from Section 5. Deriving the productions for a given input and constructing the oracle is straightforward:

```
let ps = earley_prods_of_parser parse_E "1111111"
let oracle = oracle_of_prods ps
```

We can query the oracle, for example, to find out where to split the input if we wish to parse a sequence of two symbols:

```
let _ = oracle (NT "E", NT "(E*E)") (SS("1111111", 0, 7))
- : int list = [1; 3; 5]
```

The resulting list $[1;3;5]$ reveals that the sequence of two nonterminals `E (E*E)` can be used to parse an input `"1111111"` by splitting the input at positions 1, 3 and 5. In Section 5 we hand coded the oracle. We can now improve on this by automatically constructing the oracle from the parser itself.

```
let run_parser3 p s = (let ps = earley_prods_of_parser p s in
  let oracle = oracle_of_prods ps in run_parser3' oracle p s)
```

We can then run our parser in the OCaml top-level as before:

```
let _ = run_parser3 parse_E "1111111"
- : int list = [7]
```

8 Context and memoization

Parsing context, introduced in [13], forces all top-down parse attempts to terminate, which means that *arbitrary* context-free grammars, such as those including direct and indirect left recursion, can be handled by combinator parsers. In addition, it can be shown that using parsing context preserves the *completeness* of parsing. The technical development involves the definition of the concept of a "good" parse tree, and all good parse trees are guaranteed to be returned by our parsers. In the current setting, we use parsing context *only when applying actions*. The function *mkntparser'* of Section 4 associates a concrete symbol with a parser, but does not otherwise take parsing context into account. We also define the function *mkntparser* (used in the example in Section 2), which is identical except that it takes parsing context into account. With this change, we can handle all context-free grammars. Fully formal mechanized definitions are given in [13], and further discussion on the integration of parsing context in the current setting is given in the extended version of this paper.

Memoization is a standard technique that involves storing the results of a function. When invoking the function on an input that has already been seen, the stored result is returned without re-executing the function. We use memoization in the action phase to avoid recomputing parse results for parts of the input for which the results have already been computed. Since this material is standard, we omit further details, which can be found in the extended version of the paper.

9 Experiments and performance

In this section we discuss performance, mainly by comparing our approach to the popular Haskell Happy parser generator [1]. We assess the performance of P3 and Happy across 5 different grammars. P3 outperforms Happy on all of these grammars, often by a large margin. There are clear opportunities to improve the performance of P3 even further, so these initial results are extremely encouraging[10].

Why Happy? We should compare P3 against a parser that can handle all context-free grammars: On restricted classes of grammar, we expect that P3 has good asymptotic performance, but absolute performance will not compare favourably with specialized parsing techniques. We carried out preliminary experiments with general parsers such as ACCENT[11], Elkhound[12] and SPARK[13], but encountered problems that were seemingly hard to resolve. For example, the author of SPARK confirmed that SPARK cannot directly handle grammars such as `E -> E E E | "1" |` ϵ. The underlying reason appears to be that SPARK does not make use of a compact representation of parse trees, but works instead with abstract syntax trees, which is problematic in this case because a single input can give rise to a possibly infinite number of parse trees. On the other hand, it was relatively straightforward to code up example grammars in Happy, and extract the results using a compact representation. We believe Happy represents a demanding target for comparison because it is mature, well-tested and extensively optimized code. For example, the authors of the Parsec library take Happy performance to be the definition of efficiency[14].

What to measure? We measure the time taken for each of the three phases separately. First we compare the time to compute a compact representation of all parses. This involves comparing our core implementation of Earley's algorithm with the core GLR implementation in Happy. Second, we examine the overhead of constructing the oracle. Third, we examine the cost of applying parsing actions. As a very rough guide, we expect the Earley parsing phase to be $O(n^3)$. The construction of the oracle essentially involves iterating over the list of productions, which is $O(n^3)$ in length, so we might expect that this phase should also take time $O(n^3)$. The time taken to apply the actions depends on the actions themselves, but we can analyse particular actions on a case-by-case basis to check that the observed times for this phase are reasonable.

Earley implementation P3 relies on a back-end parser. P3 terminal parsers are effectively arbitrary *functions*, whereas existing Earley implementations expect non-epsilon terminal parsers to parse a single character. For this reason, it was necessary to extend Earley's algorithm to treat corresponding "terminal items". We implemented an Earley parser from scratch in OCaml, emphasizing

[10] Details of the test infrastructure can be found in the online resources.

[11] `http://accent.compilertools.net/`

[12] `http://scottmcpeak.com/elkhound/`

[13] `http://pages.cpsc.ucalgary.ca/~aycock/spark/`

[14] "[Our real-world requirements on the combinators]... they had to be efficient (ie. competitive in speed with happy and without space leaks)" [10]

Identifier	Grammar
`aho_s`	`S -> "x" S S | ϵ`
`aho_sml`	`S -> S S "x" | ϵ`
`brackets`	`E -> E E | "(" E ")" | ϵ`
`E_EEE`	`E -> E E E | "1" | ϵ`
`S_xSx`	`S -> "1" S "1" | "1"`

Table 1. Grammars and identifiers

Size	Happy	Earley
20	0.10	0.10
40	3.18	0.10
60	28.88	0.11
80	144.50	0.13
100	512.09	0.17

Table 2. `aho_s`: time to compute compact representation

Size	Happy	Earley
100	0.22	0.19
200	2.22	0.53
300	9.75	1.24
400	28.56	2.61
500	71.08	4.42

Table 3. `aho_sml`: time to compute compact representation

Size	Earley	Oracle
100	0.21	0.35
200	0.67	2.33
300	1.84	6.68
400	3.68	15.21

Table 4. `E_EEE`: Earley parse time and oracle construction time

Size	Earley	Oracle	Action
100	0.19	0.05	0.22
200	0.49	0.50	2.18
300	1.15	2.19	6.25
400	2.49	4.60	15.4
500	4.35	9.10	31.4

Table 5. `aho_s`: Earley parse time, oracle construction time, and time to apply actions

both functional correctness and performance correctness (i.e. the implementation should have worst-case $O(n^3)$ performance). For our implementation we plan to *mechanize* correctness proofs for functional correctness (the traditional target of verification) and performance correctness (which as far as we are aware has not been tackled by the verification community for non-trivial examples). The implementation is purely functional, but is parameterized by implementations of sets and maps. The sets and maps are used linearly, so it is safe for the compiler to substitute implementations which use mutable state and in-place update. The OCaml compiler does not support this optimization currently, so we introduce mutable set and map implementations manually. The timings we give here are for the default configuration which uses mutable state in cases where the input length is less than 10000, and purely functional datastructures otherwise. Falling back on purely-functional datastructures results in worst-case $O(n^3 \lg n)$ performance, but has the advantage that space consumption is typically *much reduced*, which allows us to tackle much bigger inputs than would be possible with a solely imperative implementation. Of course, for the user the library always behaves as though it is purely functional.

Grammars and inputs We selected 5 grammars as representative examples of general context-free grammars, see Table 1. The grammars `aho_s` and `aho_sml`

are taken from a well-known book on parsing [2]. They were used to assess parser performance in related work [7]. The grammar `brackets` is a simple grammar for well-bracketed expressions. The grammar `E_EEE` is the example grammar we have used throughout the paper. The final grammar `S_xSx` is an example of a non-ambiguous grammar that cannot be handled using Packrat parsing, taken from [6]. These grammars attempt to cover different points in the grammar space: `aho_s` favours parsers which produce left-most derivations; `aho_sml` favours those that produce right-most derivations (e.g. GLR parsers such as Happy); `E_EEE` is the simplest highly-ambiguous grammar with no "left-right" or "right-left" bias. `S_xSx` parses unambiguously, and also favours parsers that produce right-most derivations. `brackets` is a standard grammar which tends to expose bugs in general parsers[15].

We used binarized versions of these grammars when measuring the performance of our Earley parser, because the P3 library feeds only binarized grammars to the Earley parser. We tried to check whether binarized versions of the grammars improved the performance of Happy, but at least with a binarized version of the grammar `E_EEE`, Happy appeared to hang on non-empty input strings.

For inputs, we simply used strings consisting of the characters `x` or `1`, or well-bracketed expressions, of varying lengths. For `S_xSx` all inputs were of odd length.

Results: computation of compact representation Our Earley parser clearly outperformed Happy across *all* grammars. For the grammars `aho_s` and `E_EEE` the results are dramatic. For example, Table 2 gives the results for `aho_s`[16]. For the grammars `aho_sml` and `S_xSx` which favour the GLR approach of Happy, Earley clearly outperforms Happy, but the results are within an order of magnitude or two. For example, the results for `aho_sml` are given in Table 3. Finally the grammar `brackets` caused Happy to appear to loop when parsing input, possibly due to a bug in Happy[17]. In addition to absolute performance, we can also check whether our Earley parser has the expected time complexity. Across all grammars we observe that our Earley implementation has worst-case performance $O(n^3)$ with mutable set and map implementations, and $O(n^3 \lg n)$ with purely functional set and map implementations. In conclusion, Earley clearly outperforms Happy on all grammars, sometimes dramatically so. On several grammars, Happy appeared to loop when attempting to parse inputs.

[15] One criticism of these grammars is that they are all "small". We also experimented with a large real-world grammar, the current ocamlyacc grammar for OCaml. For a sample 7,580 byte OCaml program, parsing takes about 1s, whereas ocamlyacc can parse this file in a fraction of a second. ocamlyacc has several features, such as precedence and associativity annotations, which make parsing deterministic. Our Earley implementation does not have such features, and thus produces all possible parses ignoring precedence and associativity. Future work should investigate supporting these sorts of annotation in Earley parsing. Importantly, Earley parsing using the OCaml grammar over a range on inputs resulted in almost-linear behaviour.

[16] All times in this section are measured in seconds. All sizes are measured in characters.

[17] Reported to the authors of Happy on 2013-06-24.

Results: oracle construction How long should we expect the construction of the oracle to take? One way to construct the oracle is by iterating over the $O(n^3)$ Earley productions. We expect that oracle construction should be $O(n^3)$, and this is what we observe in practice. For example, for the grammar `E_EEE`, the times for the Earley phase, and the times to construct the oracle, are given in Table 4. We note that even when oracle construction time is included in the parse time, our approach outperforms Happy across all grammars.

Results: applying parsing actions We now examine the overhead of applying parsing actions. Our approach restricts to good parse trees, which are finite in number. Parsers such as Happy do not restrict to good parse trees, and so attempting to construct parse trees, or apply actions to, parsing results for a grammar such as `E -> E E E | "1" |` ϵ will result in non-termination. Thus, *it is not possible to compare the performance of P3 and Happy*, but we can look at the behaviour of P3 itself.

How long should we expect the action phase to take? Consider the `aho_s` grammar `S -> "x" S S |` ϵ, where the actions count the number of characters parsed. Without memoization we expect the action phase to take an exponential amount of time. With memoization we can argue as follows. Suppose the time to apply the actions is dominated by the non-memoized recursive calls, so that we can ignore the time taken for memoized calls. There are $O(n^2)$ non-memoized calls to parse an `S` (corresponding to different spans (i, j) of the input string). For each call, the input must be split in $O(n)$ places, and the single result from each subparse combined. Thus, each call takes $O(n)$ time, giving an overall $O(n^3)$ execution time for the action phase. In practice, the time taken to look up a precomputed value in the memoization table cannot be ignored, thus we observe slightly worse than $O(n^3)$ performance. In Table 5 we include times for all phases to give an idea of the relative costs. Using a naive estimation technique puts the action phase at $O(n^{3.2})$. For the grammars `aho_sml`, `E_EEE` and `brackets` one can reason similarly. Finally, consider the following code for the grammar `S_xSx`:

```
let rec parse_S_xSx = (fun i → memo_p3 tbl (mkntparser "S" (
  ((a1 ⊗ parse_S_xSx ⊗ a1) ≫ (fun (_, (x, _)) → 2 + x))
  ⊕ (a1 ≫ (fun _ → 1)))) i)
```

For an input of length $n+1$ there should be $n/2$ recursive calls when applying the actions, each of which takes a constant time to execute, giving expected $O(n)$ cost for applying the actions. In practice, the time to apply the actions is negligible compared to the other two phases.

Conclusion The Earley parser outperforms Happy across all grammars, often dramatically so. Even though these results are very good, we note that the performance of our Earley parser is not critical: our approach can be adapted to use any general parsing implementation as a back end, so we can take advantage of faster, optimized back-end parsers if they become available.

Constructing the oracle currently involves processing all productions from the Earley stage. A more intelligent approach would be to process only those productions that contribute to a valid parse. For example, for the grammar

`S_xSx` there are only $O(n)$ such items. This optimization should reduce the oracle construction time significantly for many grammars.

Finally, the observed cost of applying the actions for our chosen grammars agrees with a basic complexity analysis, but there is some scope for reducing the real-world execution time further e.g. by using more sophisticated memoization techniques.

Overall, our implementation meets the expected worst-case bound of $O(n^3)$ for parsing and oracle construction, and has very good real-world performance when compared to Happy. For the action phase, the asymptotic performance also appears optimal. For all phases, there is scope for improving the real-world performance still further.

10 Related work

Research on parsing has been carried out over many decades by many researchers. We cannot hope to survey all of this existing work, and so we here restrict ourselves to consideration of only the most directly related work. The first parsing techniques that can handle arbitrary context-free grammars are based on dynamic programming. Examples include CYK parsing [9] and Earley parsing [5]. The popular GLR parsing approach was introduced in [16]. Combinator parsing and related techniques are probably folklore. An early approach with some similarities is [12].

The extension of combinator parsing to handle all context-free grammars using a parsing context, as in this paper, appears in [13]. The performance of this approach is $O(n^5)$, which is not competitive with the approach presented here (as confirmed by real-world experiments, which we omit for space reasons). Experiments showed that this previous approach outperformed Happy on the grammar `E_EEE`, but it seems clear that Happy has poor real-world performance on many such grammars. As described in that paper, the use of a parsing context is related to a long line of work that uses the length of the input to force termination [8]. Grammar extraction from combinator parsers, and the use of a separate back-end parser, was first described in [11]. Our approach improves on this by providing an efficient back-end, using an oracle (rather than parse trees), context (to provide meaningful semantics via the notion of "good" parse trees), and memoization to make the action phase efficient.

Our work is motivated by the desire to provide a combinator parsing interface with performance competitive with $O(n^3)$ general algorithms such as Earley parsing. In [14] the authors "develop the fully general GLL parsing technique which is recursive descent-like, and has the property that the parse follows closely the structure of the grammar rules". The desire is to improve on the shortcomings of GLR: "Nobody could accuse a GLR implementation of a parser for, say, C++, of being easy to read, and by extension easy to debug." This work is very similar in its aims to ours. Prototype hand-coded implementations of recognizers for several grammars, based on the GLL algorithm, are described in [14]. These do not provide a combinator parsing interface. An implementation of GLL in Scala

that provides the desired combinator parsing interface can be found online[18] but the author admits "at the moment, performance is basically non-existent." However, we believe that the GLL algorithm represents the main competition to our approach and we eagerly await future efficient implementations which provide a combinator parsing interface.

11 Conclusion

We presented an approach to parsing that provides a flexible interface based on parsing combinators, together with the performance of general approaches such as Earley parsing. The contributions of our work are:

- We introduced the idea of using an oracle as a compact, functional representation of parse results. This contrasts with traditional representations such as shared packed parse forests [3], which are essentially state-based representations. The idea of using an oracle as the basis of a parsing implementation is novel.
- We introduced the design of a parsing library split into a front-end combinator parsing library, and a back-end parser (here based on Earley's algorithm), connected via the oracle. This combines the well-known benefits of combinator parsing with the efficiency of general-purpose parsing algorithms such as Earley. This separation has many benefits, for example, the combinator parsers are very simple to implement, and the back-end parser can be swapped, potentially increasing performance without altering the combinator interface. This split also allows examples, such as those in Section 2, that are not possible with any other parser currently available.
- To allow arbitrary functions (of the correct type) to be used as terminal parsers, we extended Earley parsing to deal with "terminal items".
- We engineered a back-end Earley implementation. This implementation is functionally correct, and is observed to fit the worst-case time bound of $O(n^3)$ across all our example grammars. As a general parser, it has very good real-world performance, outperforming the Haskell Happy parser generator[19] across all our example grammars, often dramatically so. In future work, we intend to give mechanized proofs of functional and performance correctness for this back-end parser.
- We provided the results of real-world experiments that support our performance claims.
- We showed how to define front-end parsing combinators which allow a concrete representation of the grammar (and terminal parsers) to be extracted in order to be fed to the Earley parser. These combinators then use the results of Earley parsing to guide the action phase. We argued that the performance of the action phase, when memoized, was asymptotically close to

[18] `http://www.cs.uwm.edu/~dspiewak/papers/generalized-parser-combinators.pdf`

[19] ACCENT, Elkhound and SPARK are not competitive here, see Section 9.

optimal. No other parsers (apart from [13] which is $O(n^5)$) support applying actions when working with *arbitrary* context-free grammars, so a real-world comparison is unfortunately not possible.
- We showed how to integrate cleanly many different techniques, including combinator parsing, Earley parsing, the oracle, memoization, and parsing contexts. In addition the online distribution integrates the technique of boxing, allowing the input type to be arbitrary. This permits both scannerless parsing, and parsing with an external lexer. Even with all these different techniques, the code is extremely concise and simple.
- We showed how to combine semantic action functions with an Earley parser. For example, using our approach it is trivial to define parsers that return parse trees, see Section 2. For other techniques, such as GLL, the construction of parse trees can itself be a significant research contribution [15].
- We developed extensive examples, available in the online distribution, that demonstrate the power of our approach.

References

1. Happy, a parser generator for Haskell. `http://www.haskell.org/happy/`
2. Aho, A.V., Ullman, J.D.: The theory of parsing, translation, and compiling. Prentice-Hall, Inc. (1972)
3. Atkey, R.: The semantics of parsing with semantic actions. In: LICS '12. pp. 75–84. IEEE (2012)
4. Barthwal, A., Norrish, M.: A mechanisation of some context-free language theory in HOL4. Journal of Computer and System Sciences (2013)
5. Earley, J.: An efficient context-free parsing algorithm. Commun. ACM 13(2), 94–102 (1970)
6. Ford, B.: Packrat parsing: simple, powerful, lazy, linear time, functional pearl. In: ICFP '02. pp. 36–47. ACM (2002)
7. Frost, R.A., Hafiz, R., Callaghan, P.: Parser combinators for ambiguous left-recursive grammars. In: PADL. pp. 167–181. Springer (2008)
8. Hafiz, R., Frost, R.A.: Lazy combinators for executable specifications of general attribute grammars. In: PADL. pp. 167–182. Springer (2010)
9. Kasami, T.: An efficient recognition and syntax analysis algorithm for context-free languages. Tech. Rep. AFCRL-65-758, Air Force Res. Lab., Massachusetts (1965)
10. Leijen, D., Meijer, E.: Parsec: A practical parser library. Electronic Notes in Theoretical Computer Science 41(1), 1–20 (2001)
11. Ljunglöf, P.: Pure functional parsing (2002), Göteborg University and Chalmers University of Technology, Gothenburg, Sweden
12. Pratt, V.R.: Top down operator precedence. In: Proceedings ACM Symposium on Principles Prog. Languages (1973)
13. Ridge, T.: Simple, functional, sound and complete parsing for all context-free grammars. In: CPP, pp. 103–118. Springer (2011)
14. Scott, E., Johnstone, A.: GLL parsing. Electronic Notes in Theoretical Computer Science 253(7), 177–189 (2010)
15. Scott, E., Johnstone, A.: GLL parse-tree generation. Science of Computer Programming 78(10), 1828–1844 (2013)
16. Tomita, M.: LR parsers for natural languages. In: Proc. of the 10th Int. Conf. on Computational linguistics. pp. 354–357. ACL (1984)