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*.

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

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

Keywords parsing, functional, combinator, Earley, context-free

1. Introduction

In previous work [17] 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 [5] 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 [6] take time $O(n^3)$ in the length of the input in the worst case. Therfore, as far as performance is concerned, [17] is not competitive with such techniques. However a more subtle problem, arguably more important, is also present. Experience shows that the previous algorithm *tends to achieve its worst case run time on many classes of grammar*. This is in marked contrast to Earley parsing, which although $O(n^3)$ in the worst case, usually performs very well in practice. For example, common classes of grammar can be parsed in linear time using Earley’s algorithm [15]. 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.

Our chosen interface is parser combinators, and as with the previous work, we desire to parse arbitrary context-free grammars (we motivate these choices in Section 2). The challenge is to achieve these goals whilst 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 17. 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. Note that parsing prefixes of a substring will require more work, which is a source of inefficiency with combinator parsing as traditionally presented.

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 left-recursion we adapt the notion of parsing contexts originally introduced in [17]. A central new challenge is to reconcile the implementation of Earley parsing with that of combinator parsers. For example, consider the following parser 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, typical implementations of Earley parsing require explicit representations of the grammar, such as the following:

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

In this representation of a 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[1]:

```
[("(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.

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

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 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, 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 $(S, A.B, i, k, j)$. 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. Note that it suffices to consider rules of the form $S \to A\ B$, where there are two symbols on the right hand side, because *grammar_of_parser* returns binarized grammars.

The third interpretation of the parsing combinators follows the traditional interpretation, except that, rather than resort to parsing prefixes, we now 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 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 of the input.
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 14 we argue that the performance of this stage is close to optimal. Thus, we argue that our approach results in "good" performance. In Section 14 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[2]. Our implementation language is a small subset of OCaml, essentially the simply typed lambda calculus with integers, strings, recursive functions, records and datatypes[3]. Apart from memoization, the code is purely

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

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

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 `http://www.tom-ridge.com/p3.html`.

The structure of the rest of the paper is as follows. In Section 2 we motivate our goals. In Section 3 we give two key examples, and discuss some common misunderstandings concerning our approach. In Section 4 we discuss preliminaries and in Section 5 we introduce the basic types such as those for substrings and grammars. The subsequent sections modularly introduce different aspects of our approach. The fact that different aspects are handled modularly is a strength of our approach, and greatly facilitates exposition. We start by discussing the types related to parsers in Section 6, and then define the sequencing and alternative combinators in Section 7. In Section 8 we introduce our running example, which we develop further in Sections 10 and 12. Initially we deal with grammars which do not contain the terminal ϵ, and hand code the oracle. As we progress we extend our example so that we end up with efficient, memoized parsers capable of dealing with all context-free grammars. In Section 9 we describe the Earley parsing phase and the construction of the oracle. In Section 11 we recall the notion of parsing context and show how to combine this with our parsing combinators. In Section 13 we implement memoization to make the action phase efficient. In Section 14 we report on various experiments to measure performance. In Section 15 we briefly sketch the correctness argument. In Section 16 we discuss related work, and in Section 17 we conclude.

2. Motivation

We briefly justify the desire to handle all context-free grammars, and for an interface based on parser combinators.

Why all context-free grammars? Currently programmers have a choice: They may use a restricted parsing technique that cannot handle all context-free grammars, but which typically gives extremely good real-world performance. Alternatively, they may use a parsing technique that can handle all context-free grammars, but where the performance is poor in comparison to more specialized techniques.

Where performance is the priority, restricted parsing techniques must be used. This causes problems when the grammar in question does not fall in the restricted class. Typically the programmer must change the grammar so that it does fall in the restricted class. Sometimes, for example with natural language, this may not be possible. Even when this is possible, it is often difficult since it usually involves understanding the details of the restricted parsing algorithm. For example, a parsing failure is often accompanied by an error message, such as “shift/reduce conflict”, which is incomprehensible without some understanding of the underlying parsing machinery. It also introduces further complications, for example, the relation of the transformed grammar to the original may not be clear, so that further modification and maintenance of the grammar becomes difficult. The problem is that the specification of the grammar has been altered because the underlying tool is not capable of supporting the programmers intention.

When performance is not the priority (for example when prototyping parsers, or when performing one-off transformations on data) and where the grammar does not fall in a restricted class (such as natural language), parsers which support all context-free grammars are usually preferred, since the programmer can write a parser without understanding the details of the parsing algorithm being used. Our main motivation for handling all context-free grammars is that this choice makes our library *conceptually simple*: the user never encounters the sorts of difficulties described above with restricted parsers. In Section 14 we show that we currently provide very good performance for general context-free grammars. In future work we aim to provide the best of both worlds, i.e. in addition, for *restricted* classes of grammar, we want the performance to be comparable with the best parsers available for those classes.

Why combinator parsing? The advantages of combinator parsing are well-understood: The structure of such parsers follows closely the grammar, making them easy to write, modify and maintain. As functions, parsers are first-class objects in functional languages, allowing parsers to be parameterized by other parsers, passed as arguments to functions, combined with language features such as type classes and higher-order modules, integrated smoothly with the type system (with all the attendant benefits), and so on. Programmers have great flexibility e.g. to define their own parsing combinators for situations not covered by the library. The top-down recursive-descent nature of combinator parsers means that their behaviour with respect to side-effects is also easy for the programmer to understand. Combinator parsers constructed by hand are used extensively in the real-world, because they offer the most flexible interface to the programmer. The main disadvantages of combinator parsing, as traditionally presented, is that it cannot handle arbitrary grammars, and the performance can be very poor. In this paper we seek to address these shortcomings.

3. Example

The purpose of this section is to introduce some example parsers to illustrate our approach, and to clarify an aspect of our approach that is commonly misunderstood. An efficient parser for the grammar `E -> E E E | "1" |` ϵ is as follows:

```
let tbl = Hashtbl.create 0

let rec parse_E = (fun i → memo_p3 tbl (mkntparser "E" (
  ((parse_E ⊗ parse_E ⊗ parse_E)
    ≫ (fun (x, (y, z)) → NODE(x, y, z)))
  ⊕ (a1 ≫ (fun _ → LEAF(1)))
  ⊕ (eps ≫ (fun _ → LEAF(0))))) i)
```

Our approach is complete, which means that our library guarantees to produce *all* good parse trees (see Section 11 for the definition of good parse tree). Given the empty string, the parser returns a single parse tree (corresponding to the rule `E ->` ϵ). For input length 1, there is again a single good parse tree. For length 2, there are 3 parse trees. For length 4, there are 150 parse trees. The sequence continues as sequence A120590 from the On-line Encyclopedia of Integer Sequences[4]. For example, for input length 19, there are 441152315040444150 parse trees, but as with most exponential behaviours it is not feasible to actually compute all these parse trees. Now let us consider the following parser, which is identical except that it computes (in all possible ways) the length of the input parsed:

```
let tbl = Hashtbl.create 0

let rec parse_E = (fun i → memo_p3 tbl (mkntparser "E" (
  ((parse_E ⊗ parse_E ⊗ parse_E) ≫ (fun (x, (y, z)) → x + y + z))
  ⊕ (a1 ≫ (fun _ → 1))
  ⊕ (eps ≫ (fun _ → 0)))) i)
```

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 are aware, *no other parser can handle such examples*.

[4] `http://oeis.org/A120590`

In the rest of the paper we discuss, amongst other things, the techniques and careful engineering that makes this possible.

4. Mathematical preliminaries

In this section we review some background material, using informal mathematics. In the following sections, where necessary, we clarify the connection between these definitions and the formal code.

Substrings $s_{i,j}$ are triples of a string and two integers, representing the part of the string s between indexes i and j. If the length of s is n, then $0 \leq i \leq j \leq n$. Two substrings $s_{i,k}$ and $s_{k,j}$ can be concatenated to give the substring $s_{i,j}$.

Symbols are the disjoint union of the set of nonterminals and the set of terminals. Nonterminals are written e.g. `E`, terminals e.g. `"1"`. The terminal corresponding to the empty string is traditionally written ϵ. To each terminal is associated a set of substrings corresponding to that terminal. For example, the terminal `"1"` is associated to the set of substrings of the form $s_{i,i+1}$, where the character at index i in s is the character `1`. The terminal ϵ is associated to the set of substrings of the form $s_{i,i}$. A rule is a pair of a nonterminal and a list of symbols, written e.g. $S \to A\ B$ or `E -> "1" E "1"`. The right-hand side of a rule cannot be the empty list. The variables α, β are typically used to represent finite sequences (lists) of symbols. The bar notation is shorthand for different alternatives e.g. `E -> E E E | "1" |` ϵ denotes the three rules `E -> E E E`, `E -> "1"` and `E ->` ϵ. A grammar is a finite set of rules.

Parse trees are finitely branching trees consisting of nodes (each decorated with a nonterminal) and leaves (each decorated with a terminal and a substring representing the part of the input that was parsed by that terminal). The substring at a leaf should be in the set associated to the terminal at that leaf. Given a grammar Γ, a node in a tree decorated by a nonterminal X, and children of that node decorated by symbols $S_0, \ldots, S_m$, there should be a rule $X \to S_0 \ldots S_m$ in Γ. Moreover, the substrings at the leaves should match up in the obvious way: if two adjacent leaves have substrings $s_{i,j}$ and $s'_{j',k}$ then $s' = s$ and $j' = j$.

Given a grammar Γ, parse trees give rise to a relation between symbols X and the substrings $s_{i,j}$ that "can be parsed as" an X. In symbols, this relation is written $\Gamma \vdash X \to^* s_{i,j}$. The substring $s_{i,j}$ can be parsed as the symbol X if there is a parse tree with X at the root, such that the concatenation of the substrings at the leaves gives $s_{i,j}$. Similarly, a substring $s_{i,j}$ can be parsed as the sequence $X\ Y$ if there exists k such that $s_{i,k}$ can be parsed as an X, and $s_{k,j}$ can be parsed as a Y. This definition is extended inductively to finite sequences of symbols in the obvious way.

Given a grammar Γ, a start symbol S, and an input string s of length n, parsing involves determining the parse trees rooted at S such that the concatenation of the substrings at the leaves gives $s_{0,n}$.

5. 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. Alternatives occurring as right-hand sides of rules are not treated as shorthand, but are captured explicitly. 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 SS(s, i, j), a raw parser returns a list of substrings SS(s, i, k) indicating that the prefix 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 [])
```

The oracle type captures the idea that an oracle takes two symbols *sym1*, *sym2*, and a substring SS(s, i, j), and returns those integers k such that SS(s, i, k) can be parsed as *sym1*, and SS(s, k, j) can be parsed as *sym2*.

Finally, the type local_context is discussed in the section on parsing with context, Section 11.

[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 [17].

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

Figure 1. Basic types and trivial values

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

type mid = ⟨ rules7 : parse_rule list;
  tmparsers7 : (term, raw_parser) fmap ⟩
type inm = mid
type outm = mid

type inr = ⟨ ss4 : substring; lc4 : local_context;
  oracle4 : 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 = ⟨rules7 = []; tmparsers7 = empty_fmap⟩
```

Figure 2. Parser types and trivial values

6. Parser types

In this section we discuss the types related to parsers given in Fig. 2. In our approach, a parser should be viewed as a collection of three separate functions. 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:

```
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))
```

Moreover, this function is invertible:

```
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))
```

Here we make use of destructors such as *dest_inl*, where *dest_inl* (Inl x) $=$ x.

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 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 $\to \alpha$ output should be thought of as the sum of three functions, i.e. $p = sum3(f, g, h)$. Here f is the left-component of the parser, of type inl $\to$ outl, g is the middle-component of the parser, of type inm $\to$ outm and h is the right-component of the parser, of type inr $\to \alpha$ outr.

Left component, extracting a symbol from a parser The left component of a parser consists of a function of type inl $\to$ 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 $\to$ outm, where inm and outm are both equal to type mid. The middle component of a parser is therefore of type mid $\to$ mid. Intuitively the mid type represents the grammar associated with a parser. The middle component of a parser such as *parse_E* can be seen as 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. We use the middle component to extract the grammar associated to a parser such as *parse_E*. The idea is that if *parse_E* is a parser for the nonterminal E, then the expression *parse_E* (Inm m) should 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). The function *grammar_of_parser* can then be defined as follows:

```
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 $\to \alpha$ outr, where α outr $= \alpha$ list. The left and middle components allow a certain amount of introspection on the parsing code that forms the body of parsers such as *parse_E*. The right component is more familiar, and 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.ss4* of type substring. Two additional fields are present: *i.oracle4* is an oracle that indicates how to split the input when parsing a sequence of symbols, and *i.lc4* 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.

7. 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 should take two parsers *p1* and *p2* and produce the left component (a function from unit to symbol) of the parser $p1 \otimes p2$. The definition is:

```
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*[6]. By way of example, with the current implementation, evaluating *mk_symbol* (Seq(NT "E", NT "E")) returns (NT "(E*E)") [7].

The middle component for the combination $p1 \otimes p2$, of type mid $\to$ 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.rules7) 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 rules7 = (new_rule :: m.rules7) ⟩ in
    let m2 = g1 m1 in let m3 = g2 m2 in m3))
```

Note that the code first checks whether the nonterminal *nt* corresponding to $p1 \otimes p2$ is already present in the rules. If so, this nonterminal has already been processed, and there is no need to

[6] 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. It should be possible to check this constraint dynamically when parsers are evaluated. An alternative is to use a *gensym*-like technique to construct arguments to *mkntparser* automatically.

[7] 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.

continue further. This check also prevents non-termination 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 should take two parsers *p1* of type α parser3, and *p2* of type β parser3, and produce the right component of the parser $p1 \otimes p2$, of type inr $\rightarrow (\alpha \times \beta)$ outr.

```
let seqr p1 p2 = (fun i0 →
  let sym1 = sym_of_parser p1 in
  let sym2 = sym_of_parser p2 in
  let ks = i0.oracle4 (sym1, sym2) i0.ss4 in
  let SS(s, i, j) = i0.ss4 in
  let f1 k = (
    let rs1 = dest_inr (p1 (Inr ⟨ i0 with ss4 = (SS(s, i, k)) ⟩)) in
    let rs2 = dest_inr (p2 (Inr ⟨ i0 with ss4 = (SS(s, k, j)) ⟩)) in
    list_product rs1 rs2)
  in
  List.concat (List.map f1 ks))
```

The function works by first determining the symbols *sym1* and *sym2* corresponding to the two underlying parsers. It then calls the oracle with the appropriate symbols and substring $i0.ss4 = \mathsf{SS}(s,i,j)$. 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* and List.*concat*. The function *list_product* takes two lists and forms a list of pairs.

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.

```
let altr p1 p2 = (fun i →
  let rs1 = dest_inr (p1 (Inr i)) in
  let rs2 = dest_inr (p2 (Inr i)) in
  List.append rs1 rs2)
```

We are now in a position to define the sequential combination $p1 \otimes p2$. This uses the previously defined functions *seql*, *seqm*, *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 11 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.rules7) then Inm m
    else (
      let sym = sym_of_parser p in
      let new_rule = (nt, Atom sym) in
      p (Inm ⟨ m with rules7 = (new_rule :: m.rules7) ⟩)))
  | 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 $nt \rightarrow sym$. Thus, when passed an argument Inm *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.rules7*, also accumulates named terminal parsers in the field *m.tmparsers7*.

8. 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 all context-free grammars.

We will construct a parser for the grammar `E -> E E E | "1"`. We will make use of the raw parser *raw_a1* from Section 5. First, we define our terminal parser:

```
let a1 = mktmparser "1" raw_a1
```

We can now define the parser *parse_E*. For the actions, we will simply count the number of `1`s that we parse.

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

The definition is straightforward. Recall that the sequencing combinator makes use of the oracle to split the input; in order to run our parser on some input, we need to deal with the oracle in some way. 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*.

```
let oracle = (fun (sym1, sym2) → fun (SS(s, i, j)) → ...)
```

For *parse_E* there are two uses of the sequencing combinator: one corresponding to the occurrence of the combinator in the expression $parse\_E \otimes parse\_E$, and one to the first occurrence of the combinator in the expression $parse\_E \otimes (parse\_E \otimes parse\_E)$[8]. There are two nonterminals that can occur as arguments to the sequencing combinator, the nonterminal `E`, and the nonterminal `(E*E)`. Note that nonterminal `E` corresponds to inputs which are non-empty sequences of the character `1`. So nonterminal `(E*E)` corresponds to sequences of `1`s of length at least two. We introduce an auxiliary function *upto'* such that *upto'* $i\ j = [i+1; \ldots; j-1]$. Then for this example, the oracle may be coded as follows:

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

```
let oracle = (fun (sym1, sym2) → fun (SS(s, i, j)) →
  match (sym1, sym2) with
  | (NT "E", NT("(E*E)")) → (upto' i (j − 1))
  | (NT "E", NT("E")) → (upto' i j))
```

We can define a function to run a parser on a given input, assuming the existence of the oracle:

```
let run_parser3' oracle p s = (
  let i0 = ⟨
    ss4 = (SS(s, 0, String.length s));
    lc4 = empty_context;
    oracle4 = 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 now run the 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 examine the left and middle components of our example parser.

```
let _ = sym_of_parser parse_E
− : symbol = NT "E"
```

More interesting is the middle component:

```
let m = grammar_of_parser parse_E
val m: mid = ⟨rules7 = [("(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)"))];
 tmparsers7 = [("1", < fun >)]⟩
```

The result is a record *m*. The *m.rules7* field contains a concrete representation of the grammar, with nonterminals corresponding to every use of the sequencing and alternative combinators. In addition, the *m.tmparsers7* 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 then use a method such as Earley parsing to perform a parse of the input, determine the information necessary to construct an oracle, and then finally use the oracle to seed the action phase of the parse.

9. Earley parsing and construction of the oracle

In this section we describe how we construct an oracle from the results of an Earley parse. For an explanation of Earley parsing itself we refer the reader to the original paper [6].

Earley parsing involves Earley items, which are tuples of the form (X, α, β, i, j), typically written $(X \rightarrow \alpha.\beta, i, j)$. These items are relative to some input string s and grammar Γ. Such an item can only arise if there is a rule $X \rightarrow \alpha\ \beta \in \Gamma$. The interpretation of such an item is that $\Gamma \vdash \alpha \rightarrow^* s_{i,j}$.

Earley parsing is initialized with a single Earley item $(S \rightarrow .\beta, 0, 0)$, where S is the start symbol. When Earley parsing finishes, it returns a finite set of items, which include all items which correspond to parse trees for the input s. A complete set of Earley items constitutes a *compact representation* of all possible parse trees.

It is possible to construct an Earley parser that returns Earley productions rather than items. An Earley production is a pair consisting of an item and an additional integer l, written $(X \rightarrow \alpha.\beta, i, j, l)$. For productions, the β component is non-empty, that is, β is of the form $Y\beta'$ for some symbol Y. A production $(X \rightarrow \alpha.Y\beta', i, j, l)$ should be interpreted as follows: $(X \rightarrow \alpha.Y\beta', i, j)$ is an Earley item i.e. $\Gamma \vdash \alpha \rightarrow^* s_{i,j}$, and $\Gamma \vdash Y \rightarrow^* s_{j,l}$.

In cases where α is non-empty, a production has the form $(X \rightarrow \alpha' Z.Y\beta', i, j, l)$. If we work with grammars where the right-hand-side has at most two symbols, such a production has the form $(X \rightarrow Z.Y, i, j, l)$. These productions provide exactly the information required by the oracle: if we wish to parse the input between positions i and l, for the symbols Z and Y, we should cut the input at position j (note that the cut position is now the j component of the Earley item). We introduce the following types to represent Earley items and productions:

```
type item = ⟨ nt2 : nonterm;
  a2 : symbol list; b2 : symbol list; i2 : int; j2 : int ⟩
type production = item × int
```

The interface to the Earley parser is via the following function:

```
let earley_prods_of_parser p s = ...
```

This is a function of type α parser3 → string → production list. Given a parser and an input string, it returns a list of productions. We can process these productions to produce an oracle.

```
let oracle_of_prods ps = (fun (sym1, sym2) → fun (SS(s, i', j')) →
  let f1 (itm, l) = (itm.a2 = [sym1])
    && (itm.b2 = [sym2])
    && (i', j') = (itm.i2, l)
  in
  let ps = List.filter f1 ps in
  List.map (fun (itm, _) → itm.j2) ps)
```

Applying *oracle_of_prods* to a list of productions *ps* gives an oracle, that is, a function of type (symbol × symbol) → substring → int list. The function simply picks out those productions $(X \rightarrow \alpha.\beta, i, j, l)$ where α is the single symbol *sym1*, β is the single symbol *sym2*, and the span (i, l) corresponds to the span (i', j') of the substring that we want to parse. The cut indexes are simply those j for such productions[9].

10. Example, with Earley parsing

We continue the example from Section 8. 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 8 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)
```

[9] The definition of *oracle_of_prods* given here is solely for expository purposes; a more efficient implementation is provided in the online distribution.

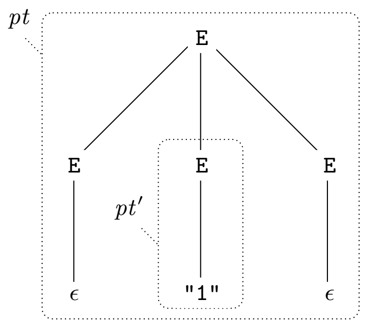

Figure 3.

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

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

11. Context

We cannot yet handle all context-free grammars, at least during the action phase. For this, we need the notion of parsing context. In this section we recall the essential definitions. We also show how parsing context may be smoothly combined with the machinery developed in previous sections. For reasons of space, we give an informal presentation, covering the main ideas, but omitting formal definitions and theorems which can be found in [17].

Consider the grammar `E -> E E E | "1" |` ϵ. This grammar is highly ambiguous. It also gives rise to infinitely many parse trees. In Fig. 3 we show a parse tree formed from this grammar (the leaves are decorated with strings rather than substrings). Note that *pt* and *pt*′ are rooted at the same nonterminal, correspond to the same string (a string consisting of a single `1`), and *pt'* is properly contained in *pt*. If such a situation can occur then the grammar gives rise to an infinite number of parse trees. Moreover, *this is the only way an infinite number of parse trees can arise*. 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: the operation of replacing subtrees such as *pt* by subtrees *pt*′ that are contained in *pt* defines a reduction process on parse trees, which preserves the parsed input. Thus, for a given grammar we have identified a class of parse trees (the good parse trees) that is complete (any string that can be parsed, can be parsed to give a good parse tree) and moreover is *finite*.

We wish to restrict parsing to only the good parse trees. To this end, at a given node the combinator parser records the current nonterminal and input string in a *parsing context*. If the parser encounters the same nonterminal and input string while parsing a subtree, the parse should be abandoned. The context consists of a finite set (represented by a list) of pairs, where each pair consists of a nonterminal, and the input substring.

For example, at the root of *pt*, the context is initially empty. The parser adds the entry $(\texttt{"E"}, \mathsf{SS}(\texttt{"1"}, 0, 1))$ corresponding to the root of *pt* before parsing the subtrees. When the parser reaches *pt*′, it constructs the entry $(\texttt{"E"}, \mathsf{SS}(\texttt{"1"}, 0, 1))$ corresponding to the root of *pt*′ and checks whether it is already contained in the context. In this case, the entry *is* contained in the context, and the parse is abandoned.

The implementation of contexts (finite sets) as lists is straightforward. We need a function to update the context at a node, and a function to check whether the item corresponding to a node is already contained in the context.

```
let update_context (LC(lc)) (nt, SS(s, l, h)) = (
  LC((nt, SS(s, l, h)) :: lc))

let context_contains (LC(lc)) (nt, SS(s, l, h)) = (
  List.exists (fun x → x = (nt, SS(s, l, h))) lc)
```

We can then lift the function *update_context* to the input type:

```
let update_lc4 nt p = (fun i → match i with
  | Inr i →
    p (Inr ⟨ i with lc4 = (update_context i.lc4 (nt, i.ss4)) ⟩)
  | _ → p i)
```

This function takes a string *nt* such as `"E"` representing the nonterminal at the current node, and a parser *p* representing the parser for that nonterminal, and updates the context before calling the underlying parser on the input.

When executing a parser for nonterminal *nt* on input *i*, we first need to check whether the entry $(nt, i.ss4)$ already occurs in the context. If so, we should abandon the parse. Otherwise, we should update the context and continue.

```
let check_and_upd_lc4 p = (fun i → match i with
  | Inr i → (let (f, g, h) = unsum3 p in
    let NT nt = sym_of_parser p in
    let should_trim = context_contains i.lc4 (nt, i.ss4) in
    if should_trim then Inr [] else update_lc4 nt p (Inr i))
  | _ → p i)
```

The function *check_and_upd_lc4 p* affects only the right component. It first determines the nonterminal *nt* corresponding to *p*. It then checks whether the entry $(nt, i.ss4)$ is already contained in the context. If so, it abandons the parse and returns the empty list of results. Otherwise it updates the context and calls the underlying parser *p*. Finally, we construct the function *mkntparser*:

```
let mkntparser nt p = (check_and_upd_lc4 (mkntparser' nt p))
```

With these definitions, we can now handle *all* context-free grammars.

We make two further observations that are relevant to the discussion in Section 13 of the memoization of functions (such as parsers) which take contexts as arguments. The first is that contexts are finite *sets*, so that we might consider using *sorted* lists to represent contexts. The second observation is that many contexts are equivalent in the way they affect a parse. Suppose the entry for the current node is of the form $(nt, \mathsf{SS}(s, i, j))$. All entries in the context will be of the form $(nt', \mathsf{SS}(s, i', j'))$, where $i' \leq i, j \leq j'$. If in fact $i' < i$ (or $j < j'$) then that entry in the context will not cause the parse to be abandoned at the current node. Nor will it cause the parse to be abandoned for any descendants of the current node. Such entries are therefore irrelevant. Contexts which are identical but for the presence of irrelevant entries are equivalent in the way they affect a parse. Given a substring *ss*, we can normalize a given context to remove irrelevant entries:

```
let normalize_context (LC(lc)) ss = (
  LC(List.filter (fun (nt, ss') → ss' = ss) lc))
```

In Section 13, memoization performs best when contexts are *normalized*, and represented using *sorted* lists.

12. Example, with context

Consider the grammar `E -> E E E | "1" |` ϵ. We can define a parser for this grammar as follows.

```
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)
```

Compared to previous examples, the difference is that we can now parse the empty string ϵ as the nonterminal "E". This in turn gives rise to an infinite number of parse trees. However, our parsers can now handle such grammars easily.

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

At this point, we have developed parser combinators that can handle all context-free grammars. However, parsing is still slow because, during the action phase, a parser such as *parse_E* may be called many times on the same substring. To make our parsing efficient we need to use memoization to avoid repeated work.

13. Memoization

Memoization of a function f involves maintaining a partial map, known as a lookup table, from arguments i to values $f\ i$. Instead of computing f on a argument i, memoization first checks whether the value is already stored in the table. If so, the value is returned without further computation involving f. Otherwise, the value of $f\ i$ is computed, the table is updated, and the value is returned. Memoization is only valid if f behaves as a pure function. A direct implementation using OCaml hashtables is as follows:

```
let memo tbl f i = (
  if (Hashtbl.mem tbl i) then (Hashtbl.find tbl i)
  else (let v = f i in let _ = Hashtbl.add tbl i v in v))
```

Here, arguments are uninterpreted, however, it is often the case that several arguments $i_1, i_2, \ldots$ should be considered equivalent because the values $f\ i_1, f\ i_2, \ldots$ are the same. If we have stored the result of evaluating $f\ i_1$ in the memo table, we would like to reuse this value to avoid computing $f\ i_2$. To handle this case, we extend the *memo* function with an auxiliary function *key_of_input* that computes a key k from an input i. It is expected that $i_1, i_2, \ldots$ will all map to the same key, and arguments i, i' should map to the same key only if $f\ i = f\ i'$. The index into the memoization table is now the key, rather than the argument itself.

```
let memo tbl key_of_input f i = (
  let k = key_of_input i in
  match k with | None → (f i) | Some k → (
    if (Hashtbl.mem tbl k) then (Hashtbl.find tbl k)
    else (let v = f i in let _ = Hashtbl.add tbl k v in v)))
```

The function *key_of_input* may return None, which indicates that the value for the given argument should not be memoized. For a parser *p*, we wish to memoize only the right component[10].

```
let key_of_input i = (match i with
  | Inr i → (let lc = normalize_context i.lc4 i.ss4 in
    let k = (lc, i.ss4) in Some k)
  | _ → None)
```

Note that the right component of a parser accepts arguments Inr *i* which, in addition to the substring component *i.ss4* also include a context *i.lc4* and an oracle *i.oracle4*. The oracle does not change during the action phase, and so need not be considered when memoizing. So the input to a parser is effectively a pair of a context and a substring. As observed in Section 11, memoization performs best when the context is normalized relative to the input string at the current node, so we use the function *normalize_context* to normalize *i.lc4* when constructing the key. It is now straightforward to memoize a parser:

```
let memo_p3 tbl p i = (memo tbl key_of_input p i)
```

[10] P3 differs from mini-P3 in that it uses boxing [2] to enable quick computation of hashcodes for substrings.

At this point, we have defined all the functions necessary to understand the example in Section 3.

14. 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[11].

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. For example, P3 should probably not be used to parse XML if absolute performance is the priority. We carried out preliminary experiments with general parsers such as ACCENT[12], Elkhound[13] and SPARK[14], 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 almost the definition of efficiency[15].

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 very general, whereas existing Earley implementations expect 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 both functional correctness and performance correctness (i.e. the implementation should have worst-case $O(n^3)$ performance). For our implementation it should be possible 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

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

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

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

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

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

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 $O(n^3.ln_2\ n)$ 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. These are:

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"`

The grammars `aho_s` and `aho_sml` are taken from a well-known book on parsing [3]. They were used to assess parser performance in related work [9]. 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. It has no "left-right" or "right-left" bias, which is not the case for `aho_s` and `aho_sml`. The final grammar `S_xSx` is an example of a non-ambiguous grammar that cannot be handled using Packrat parsing, taken from [7].

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.[16]

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, here are the results for `aho_s`[17]:

Size	Happy parse time	Earley parse time
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

[16] 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 over 10s, whereas the standard OCaml parser 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, in order to make our approach competitive with OCamlyacc on the restricted class of grammars supported by OCamlyacc.

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

For the grammars `aho_sml` and `S_xSx`, Earley clearly outperforms Happy, but the results are within an order of magnitude or two. For example, here are the results for `aho_sml`:

Size	Happy parse time	Earley parse time
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

Finally the grammar `brackets` caused Happy to appear to loop when parsing input, possibly due to a bug in Happy[18]. 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 $O(n^3.ln_2\ n)$ 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. We believe our Earley implementation is one of the fastest general parsers in existence, and we welcome suggestions for further systems to compare against.

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:

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

We note that it may be possible to construct the oracle, using mutable state, in time $O(n^2)$, but we leave optimization of this phase to future work. We also note that even with oracle construction, 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 in this area, 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

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

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 the following, we include times for all phases to give an idea of the relative costs. Using a naive estimation technique puts the action phase at $O(n^{3.2})$.

Size	Earley phase	Oracle phase	Action phase
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

For the grammars `aho_sml` and `E_EEE` one can reason similarly. The grammar `brackets` might be expected to have similar performance $O(n^3)$ when applying the actions. If the input consists of alternating left and right brackets, this is the case. Our example inputs consist of relatively few blocks of deeply nested brackets, so that most recursive calls execute in constant time rather than $O(n)$, and overall the observed performance is closer to $O(n^2)$. Finally, consider the following code for the grammar `S_xSx`:

let rec $\textit{parse\_S\_xSx} = (\mathsf{fun}\ i \to \textit{memo\_p3 tbl}\ (\textit{mkntparser}\ \texttt{"S"}\ ($
$((\textit{a1} \otimes \textit{parse\_S\_xSx} \otimes \textit{a1}) \gg (\mathsf{fun}\ (\_, (x, \_)) \to 2 + x))$
$\oplus (\textit{a1} \gg (\mathsf{fun}\ \_ \to 1)))) i)$

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.

In summary, the action phase appears to behave according to a straightforward complexity analysis.

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. It may also be possible to use imperative programming techniques to reduce oracle construction time to $O(n^2)$, but we leave this to future work.

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. For the action phase, the asymptotic performance also appears optimal. For all phases, there is scope for improving the real-world performance still further.

15. Correctness

The focus of this paper is on presenting the main ideas involved and providing evidence of parsing performance, but here we briefly sketch a proof that our approach is sound and complete in the sense of our previous work [17]. Correctness falls into several parts: correctness of the Earley implementation and oracle construction, and correctness of the parsing combinators given a correct oracle. Functional correctness of the Earley implementation involves defining "partial parse trees" corresponding to Earley items, and then conducting a fairly routine proof of correctness of the Earley implementation: items correspond to partial parse trees, completed items correspond to complete parse trees, and the Earley algorithm is sound (any item it generates corresponds to a partial parse tree) and complete (it generates all items). Correctness of oracle construction requires showing that, given grammar Γ and input s, if the oracle is provided with nonterminals A and B and indices i, j, and if k is returned in the result list, then $\Gamma \vdash A \to^* s_{i,k}$, and $\Gamma \vdash B \to^* s_{k,j}$. This follows directly from the correctness of the Earley implementation, and the definition of oracle construction in terms of Earley productions. Finally, correctness of the parsing combinators essentially involves showing that the parsing context permits all good parse trees to be returned. This proof was mechanized in [17] for related combinators and prefix-parsing.

In fact, the Earley implementation was developed with partial mechanized definitions and proofs. Probably because of this, we have never found a bug in our Earley implementation, despite extensive use over the last 12 months. The code is sufficiently complex that a hand proof of correctness would almost certainly contain errors. In future work, we intend to provide mechanized proofs of functional and performance correctness.

16. 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 [12] and Earley parsing [6]. The popular GLR parsing approach was introduced in [20]. Combinator parsing and related techniques are probably folklore. An early approach with some similarities is [16]. Versions that are clearly related to the approach taken in this paper were popularized in [11].

The extension of combinator parsing to handle all context-free grammars using a parsing context, as in this paper, appears in [17]. 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 [8–10]. 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 very poor performance on many such grammars (certainly not the expected $O(n^3)$ of the underlying GLR algorithm).

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 [18] 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 [18]. 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[19] 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

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

and we eagerly await future efficient implementations which provide a combinator parsing interface.

17. 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 [4], which are essentially state-based representations. Although the notion of an oracle is not new [13], 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 3, 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[20] 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 optimal. No other parsers (apart from [17] which is $O(n^5)$) support applying actions when working with *arbitrary* context-free grammars, so a real-world comparison is unfortunately not possible. This paper gives almost the full code for the front-end parsing combinators.
- 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 3. For other techniques, such as GLL, even the construction of parse trees can be a significant research contribution [19].
- We developed extensive examples, available in the online distribution. For example, these illustrate how to incorporate traditional aspects of parsing, such as precedence and associativity, in the action phase of our parsers. Further examples illustrate the power of combinator parsing for arbitrary context-free grammars.

[20] ACCENT, Elkhound and SPARK are not competitive here, see Section 14.

References

[1] Happy, a parser generator for Haskell. `http://www.haskell.org/happy/`.

[2] U. A. Acar, G. E. Blelloch, and R. Harper. Selective memoization. POPL '03, pages 14–25. ACM, 2003.

[3] A. V. Aho and J. D. Ullman. *The theory of parsing, translation, and compiling*. Prentice-Hall, Inc., 1972.

[4] R. Atkey. The semantics of parsing with semantic actions. In *LICS '12*, pages 75–84. IEEE, 2012.

[5] A. Barthwal and M. Norrish. A mechanisation of some context-free language theory in HOL4. *Journal of Computer and System Sciences*, 2013.

[6] J. Earley. An efficient context-free parsing algorithm. *Commun. ACM*, 13(2):94–102, 1970. ISSN 0001-0782. .

[7] B. Ford. Packrat parsing: simple, powerful, lazy, linear time, functional pearl. In *ICFP '02*, pages 36–47. ACM, 2002.

[8] R. A. Frost, R. Hafiz, and P. C. Callaghan. Modular and efficient top-down parsing for ambiguous left-recursive grammars. In *IWPT '07*, pages 109–120. ACL, 2007.

[9] R. A. Frost, R. Hafiz, and P. Callaghan. Parser combinators for ambiguous left-recursive grammars. In *PADL*, pages 167–181. Springer, 2008.

[10] R. Hafiz and R. A. Frost. Lazy combinators for executable specifications of general attribute grammars. In *PADL*, pages 167–182. Springer, 2010.

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

[12] 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.

[13] L. Lee. Fast context-free grammar parsing requires fast boolean matrix multiplication. *Journal of the ACM (JACM)*, 49(1):1–15, 2002.

[14] D. Leijen and E. Meijer. Parsec: A practical parser library. *Electronic Notes in Theoretical Computer Science*, 41(1):1–20, 2001.

[15] J. M. I. M. Leo. A general context-free parsing algorithm running in linear time on every LR(k) grammar without using lookahead. *Theoretical Comp. Sci.*, 82(1):165 – 176, 1991.

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

[17] T. Ridge. Simple, functional, sound and complete parsing for all context-free grammars. In *CPP*, pages 103–118. Springer, 2011.

[18] E. Scott and A. Johnstone. GLL parsing. *Electronic Notes in Theoretical Computer Science*, 253(7):177–189, 2010.

[19] E. Scott and A. Johnstone. Gll parse-tree generation. *Science of Computer Programming*, 78(10):1828–1844, 2013.

[20] M. Tomita. LR parsers for natural languages. In *Proc. of the 10th Int. Conf. on Computational linguistics*, pages 354–357. ACL, 1984.