GLL Parsing with Flexible Combinators

L. Thomas van Binsbergen
Royal Holloway, University of London
ltvanbinsbergen@acm.org

Elizabeth Scott
Royal Holloway, University of London
e.scott@rhul.ac.uk

Adrian Johnstone
Royal Holloway, University of London
a.johnstone@rhul.ac.uk

Abstract

At SLE in 2014, Ridge presented the P3 combinator library with which parsers can be developed for left-recursive, non-deterministic and ambiguous grammars. A combinator expression in P3 yields a binarised grammar reflecting the expression's structure. The grammar is given to an underlying, generalised parsing procedure computing all derivations.

In this paper we present a combinator library with a similar architecture to P3, adjusting it to avoid grammar binarisation. Avoiding binarisation has a significant positive effect on the running times of the underlying parsing procedure, which we demonstrate using real-world grammars. Binarisation is avoided by restricting the applicability of combinators, resulting in combinator expressions closely resembling BNF fragments. Usability is recovered by defining coercions that automatically convert expressions where necessary. As the underlying parsing procedure, we use a purely functional variant of generalised top-down (GLL) parsing.

ACM Reference Format:
L. Thomas van Binsbergen, Elizabeth Scott, and Adrian Johnstone. 2018. GLL Parsing with Flexible Combinators. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

The syntax of a software language is usually defined by a BNF description of a context-free grammar. Parser generators such as Yacc or Happy implement (variants of) BNF and, depending on the underlying parsing technology, generate parsers for different classes of context-free grammars. Generalised parsing algorithms such as GLR [39] and GLL [29, 31] admit all context-free grammars and compute all possible derivations of an input sentence [30, 33]. Several parser generators generate generalised parsers [19, 24, 41].

In the first part of this paper we give a purely functional description and implementation of a generalised top-down (GLL) parsing algorithm in Haskell. In contrast to the original descriptions [29, 31], we explain GLL's data structures as abstract sets with basic operations, rather than specialised implementations, focussing on the algorithm's logic.

In languages with higher-order functions, functions can be defined that take parsers as arguments and combine them to form new parsers. These so-called parser combinators are popular in functional programming communities and often serve as an example of the benefits of functional programming. The basic approach suffers from non-termination due to left-recursion, and potential from inefficiency due to backtracking, as standard implementations are based on some form of recursive descent parsing. Several methods have been suggested to generalise the approach and increase the class of terminating and efficient combinator parsers. For example, memoisation can overcome some of the inefficiencies of backtracking [26], lookahead can reduce backtracking [36], sophisticated memoisation can handle left-recursion [17], and left-recursion can be removed automatically [2]. Alternatively, a grammar can be extracted from the combinator expressions and given to a stand-alone parsing procedure [6, 23]. This is the approach taken by P3, presented by Ridge at SLE in 2014 [28]. In P3, the extracted grammars are restricted to a binarised form and this impacts performance, as demonstrated in our evaluation section.

In the second part of this paper we present a 'BNF combinator' library which is similar to P3, but the extracted grammars are not binarised and are given to our GLL implementation. In our evaluation section we show that the BNF combinators can be used to implement parsers for real-world software languages. Although presented in Haskell, the code of this paper transfers to other functional languages.

Section 2 defines our representation of grammars and derivations. Section 3 describes and implements a recursive descent parsing procedure, which is generalised to GLL in Section 4. Section 5 explains basic parser combinators and Section 6 introduces our 'BNF combinators' as an alternative. The BNF combinators are subsequently defined in Sections 7, 8, and 9. Section 10 evaluates the BNF combinators demonstrates the negative effects of grammar binarisation. Sections 11 and 12 conclude and discuss related work.

2 Grammars and Derivations

In this section we explain the Haskell representation of grammars and derivations used throughout this paper. We assume familiarity with the theory of syntax analysis. For an introduction we refer the reader to [14].

Conference'17, July 2017, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

A context-free grammar, simply grammar hereafter, is a mapping from nonterminals to sets of right-hand sides (also called alternates), where a right-hand side is a list of symbols. A symbol is either a terminal (a value of some type t) or a nonterminal (a value of some type n):

```
type Grammar n t = M.Map n (S.Set (Rhs n t))
type Rhs n t     = [Symbol n t]
data Symbol n t  = Term t | Nt n deriving Show
type Input t     = Array Int t
```

(Maps are imported from *Data.Map* under the qualified name *M* and sets are imported from *Data.Set* under the qualified name *S*.) Parser input is represented as an array of terminals. In our examples we use *String* for nonterminals and *Char* for terminals.

A nonterminal in a grammar derives a sequences of terminal symbols (sentences). Sentence I is derived by nonterminal x if I can be obtained by choosing a right-hand side of x and repeatedly replacing nonterminals within it by one of their right-hand sides. The set of sentences that can be derived from a nonterminal x in a grammar *gram* is the language generated by x in *gram*. For example, nonterminal `"tuple"` generates the language {`"()"`,`"(a)"`,`"(a,a)"`,...} in the grammar:

```
tupleRR =    M.fromListWith S.union
  [("tuple",S.singleton [Term '(',Nt "as",Term ')'])
  ,("as"    ,S.singleton [])    -- empty right-hand side
  ,("as"    ,S.singleton [Term 'a',Nt "more"])
  ,("more"  ,S.singleton [])    -- empty right-hand side
  ,("more"  ,S.singleton [Term ',',Term 'a',Nt "more"])]
```

A recognition procedure is an algorithm that, given a grammar, a nonterminal x, and a sentence I, determines whether I is in the language generated by x in the grammar. A parsing procedure is a recognition procedure that provides proof that this is the case, typically in the form of a derivation tree. A recognition or parsing procedure is general if it terminates and gives correct results for all grammars. A procedure is complete if it is general and provides proof for all possible derivations of the input sentence. A grammar is ambiguous if one sentence has multiple derivations.

State-of-the-art complete parsers compute Binarised Shared Packed Parse Forest (BSPPFs), a data structure capable of representing all possible derivations of a sentence in $O(n^3)$ space[1] [30, 33]. Rather than building BSPPFs as sets of nodes and edges, the parsing procedures of this paper construct sets of *extended packed nodes*[2] (EPNs) containing sufficient information to construct all other nodes and the edges between them. For details on BSPPFs, we refer the reader to [33].

```
type Slot n t  = (n,Rhs n t,Rhs n t)
type EPN n t   = (Slot n t,Int,Int,Int)
type EPNs n t ≡ S.Set (EPN n t)
```

[1]Where n is the length of the input sentence.

[2]EPNs coincide with the 'Earley productions' used by Ridge.

A *grammar slot* is a triple (x,α,β) where $\alpha\beta$ is an alternate of nonterminal x. An EPN is a quadruple $((x,\alpha,\beta),l,k,r)$, with (x,α,β) a grammar slot and integers $l \leqslant k \leqslant r$, indicating that α derives the subsentence ranging from l to $r-1$. This fact may be true in different contexts, represented by x and β. The existence of other EPNs of the form $((x,\alpha,\beta),l,k',r)$ shows that this fact may be established in several ways; the pivot k indicates that the last symbol in α derives the subsentence from k to r (if α empty then $l=k$). We use $\equiv$ to *suggest* a possible definition for EPNs. In actuality we use a more efficient definition based on Patricia trees [27].

The following operations are used to construct EPN sets:

```
emptyPNs    :: EPNs n t
singlePN    :: EPN n t    → EPNs n t
unionPNs    :: EPNs n t   → EPNs n t → EPNs n t
unionsPNs   :: [EPNs n t] → EPNs n t
unionsPNs = foldr unionPNs emptyPNs
fromListPNs :: [EPN n t]  → EPNs n t
fromListPNs = foldr (unionPNs ∘ singlePN) emptyPNs
```

3 Recursive Descent

This section describes and implements a straightforward but naive recursive descent parsing procedure. First we give a recognition procedure which is subsequently extended to compute EPN sets.

The recognition procedure Recognition procedure FUNRDR is implemented by *funrdr* and receives a grammar *gram*, a nonterminal x, and a sentence *inp*, and returns a Boolean to indicate whether *inp* is derived by x in *gram*.

```
funrdr gram x inp = upper + 1 ∈ descend₁ gram inp x 0
  where (_,upper) = bounds inp
```

The function *funrdr* 'descends' nonterminal x. Descending a nonterminal x involves selecting a subset of the alternates of x for 'processing'. The selection can be based on lookahead sets computed for each of the alternates and backtracking can be used to select the first successful alternate. We view lookahead as an optional (and orthogonal) optimisation and leave it out of our presentation for simplicity[3]. We process all alternates, using Wadler's 'list of successes' method [42] for backtracking whilst retaining all results.

```
descend₁ gram inp x l =
  concatMap (λβ → process₁ gram inp β l) alts
  where alts = maybe [] S.toList (M.lookup x gram)
```

Function $descend_1$ is given, besides a grammar and a sentence, a nonterminal and an index (referred to as *left extent*) and returns a sequence of indices (referred to as *right extents*) If $descend_1$ *gam inp x l* returns right extents $[r_1,\ldots,r_n]$ then x derives the subsentences of *inp* ranging from l to r_i-1, for $1 \leqslant i \leqslant n$.

[3]The implementation evaluated in Section 10 uses lookahead.

```
process1 gram inp β k = case β of
  [ ]       → [k]   -- completed processing
  (w : β') → concatMap (process1 gram inp β') $ case w of
               Nt x    → descend1 gram inp x k
               Term t | match1 inp t k → [k + 1]   -- match
                      | otherwise      → [ ]       -- mismatch
```

Processing a sequence of symbols β with index k involves finding right extents $[r_1, \ldots, r_n]$ for the first symbol w in β. If w is nonterminal, the right extents are found by descending w. If w is terminal, and w is at the k-th position of *inp*, then $n = 1$ and $r_1 = k + 1$. This check is performed by $match_1$:

```
match1 :: Eq t ⇒ Input t → t → Int → Bool
match1 inp t k = k ⩾ lower ∧ k ⩽ upper ∧ inp ! k ≡ t
  where (lower, upper) = bounds inp
```

A recursive call is made for each r_i to process the rest of the sequence (β') with $k = r_i$. All the right extents returned by all the recursive calls are concatenated. The base case of the recursion is $\beta = [\,]$, returning the singleton sequence k.

Left-recursion Function *funrdr* is applied to a grammar and a nonterminal to obtain a recogniser, a function from *Input* to *Bool*. For example, *funrdr tupleRR* `"tuple"` recognises the language generated by "tuple". However, if the given grammar *gram* has a left-recursive[4] nonterminal x, and if x is descended, then the recogniser fails to terminate. This is because expression $descend_1$ *gram inp x k* reduces to $process_1$ *gram inp* $(x : \beta)$ *k*, for all *inp* and *k* and some β, if x is left-recursive. The latter reduces to $descend_1$ *gram inp x k*, causing nontermination.

We extend FUN-RDR to a parsing procedure FUN-RDP by computing EPN sets. In Section 4 we generalise FUN-RDP to work for all grammars.

The parsing procedure Function $process_2$ is given a grammar slot (x, α, β) (rather than just β, compared to $process_1$), containing all relevant information of the alternate being processed. A call to $process_2$ is also given an additional index l: the left-extent of the call to *descend* leading up to this call to *process*. The function $process_{init}$ makes the initial call to $process_2$ for a particular alternate β of nonterminal x.

```
processinit gram inp x l β = process2 gram inp (x, [ ], β) l l
```

As for $process_1$, a call to $process_2$ with $\beta = w : \beta'$ attempts to find right extents $[r_1, \ldots, r_n]$ and makes a recursive call for each r_i. The results of the recursive calls, which include EPN sets, are concatenated with $concat_2$:

```
concat2 = foldr combine ([ ], emptyPNs)
  where combine (rs, ns) (rs', ns') = (rs ⧺ rs', unionPNs ns ns')
```

The base case $\beta = [\,]$ of $process_2$ returns the empty set of EPNs, or, if $\alpha = \beta = [\,]$, a single EPN $((x, [\,], [\,]), l, l, l)$.

```
process2 gram inp (x, α, β) l k = case β of
  [ ] | α ≡ [ ]    → ([k], singlePN ((x, [ ], [ ]), l, l, l))
      | otherwise → ([k], emptyPNs)
  (w : β') → continue (x, α ⧺ [w], β') $ case w of
               Nt x    → descend2 gram inp x k
               Term t | match1 inp t k → ([k + 1], emptyPNs)
                      | otherwise      → ([ ], emptyPNs)
where continue slot (rs, ns) = (rs', unionsPNs [ns, ns', ns''])
        where (rs', ns') = concat2 (map call rs)
              call r     = process2 gram inp slot l r
              ns''       = fromListPNs [(slot, l, k, r) | r ← rs]
```

Function $descend_2$ differs from $descend_1$ in that it applies $concat_2 \circ map$ rather than *concatMap* and applies $process_{init}$:

```
descend2 gram inp x l =
  concat2 (map (processinit gram inp x l) alts)
  where alts = maybe [ ] S.toList (M.lookup x gram)
funrdp gram x inp = snd (descend2 gram inp x 0)
```

As an example of running FUN-RDP (implemented by *fundrp* above), consider the following output, in which the EPNs 1-5, 7-9, and 11-14 prove that "tuple" derives "(a,a)" (function *printEPNs* prints a set of EPNs in a readable fashion):

```
> printEPNs (funrdp tupleRR "tuple" (listArray (0,4) "(a,a)"))
1  : (("tuple",['('],["as",')']),0,0,1)
2  : (("tuple",['(',"as"],[')']),0,1,1)
3  : (("tuple",['(',"as"],[')']),0,1,2)
4  : (("tuple",['(',"as"],[')']),0,1,4)
5  : (("tuple",['(',"as",')'],[]),0,4,5)
6  : (("as",[],[]),1,1,1)
7  : (("as",['a'],["more"]),1,1,2)
8  : (("as",['a',"more"],[]),1,2,2)
9  : (("as",['a',"more"],[]),1,2,4)
10 : (("more",[],[]),2,2,2)
11 : (("more",[',',],['a',"more"]),2,2,3)
12 : (("more",[',','a'],["more"]),2,3,4)
13 : (("more",[',','a',"more"],[]),2,4,4)
14 : (("more",[],[]),4,4,4)
```

4 FUN-GLL

This section describes and implements the FUN-GLL parsing procedure by generalising FUN-RDP of the previous section. Nontermination due to left-recursion, and other kinds of 'repeated work', are prevented by introducing descriptors (similar to the descriptors in [31, 32] and Earley items in [9]). The complexity is potentially $O(n^3)$ in both space and runtime, depending on the implementation of the data structures.

A descriptor is a triple containing the arguments of a call to $process_2$, a slot, a left extent and another index:

```
type Descr n t = (Slot n t, Int, Int)
```

A worklist $\mathcal{R}$ contains descriptors that require processing; its elements are processed one by one by calling *process* (replacing $process_2$). A set $\mathcal{U}$ of descriptors remembers the descriptors added to $\mathcal{R}$, and is used to ensure that no descriptor is added to the worklist a second time. Preventing repeated

[4] For a formal definition of left-recursion we refer to [14].

processing avoids nontermination due to left-recursion, but may result in the omission of derivation information.

Consider the situation in which the descriptor $((x, \alpha, s : \beta), l, k)$ has been processed, with s a nonterminal, having resulted in further descriptors $((x, \alpha \# [s], \beta), l, r_i)$, for all r_i in some set R. If the next processed descriptor is of the form $((y, \delta, s : v), l', k)$, then no further descriptors are added to $\mathcal{R}$, as all descriptors of the form $((s, [\,], \mu), k, k)$ have already been encountered. However, since s derives the subsentences ranging from k to $r_i - 1$, with $r_i \in R$, we should still add the descriptors $((y, \delta \# [s], v), l', r_i)$ and EPNs $((y, \delta \# [s], v), l', k, r_i)$. To avoid missing these descriptors and EPNs, we introduce the binary relation[5] $\mathcal{P}$ between pairs of *commencements* and right extents, where a commencement is a pair of a nonterminal and a left extent (i.e. the arguments of $descend_2$). In the example situation, the set R is embedded in $\mathcal{P}$ as specified by the equation $R = \{r \mid ((s, k), r) \in \mathcal{P}\}$.

This is not sufficient; some of the descriptors of the form $((s, [\,], \mu), k, k)$, or descriptors that follow from these, may not have been processed yet. This means that there may be right extents R', with $R' \cap R = \emptyset$, for which it holds that s derives the subsentences ranging from k to $r_j - 1$, with $r_j \in R'$. When the right extents in R' are 'discovered', it is necessary to add the descriptors $((y, \delta \# [s], v), l', r_j)$ and $((x, \alpha \# [s], \beta), l, r_j)$ as well as the EPNs $((y, \delta \# [s], v), l', k, r_j)$ and $((x, \alpha \# [s], \beta), l, k, r_j)$. We introduce the binary relation[6] $\mathcal{G}$ between commencements and *continuations*, where a continuation is a pair of a slot and a left extent (i.e. a descriptor missing an index).

type *Comm n t* = (n, Int)
type *Cont n t* = $(Slot\ n\ t, Int)$

Summary The FUN-GLL algorithm is summarised as follows. While there are descriptors in the worklist, arbitrarily select the next descriptor $((x, \alpha, \beta), l, k)$ to be processed. If $\beta = w{:}\beta'$ and w is terminal, **match** the terminal at position k in the input sentence with w. Only if the match is successful, add the descriptor $((x, \alpha \# [w], \beta'), l, k + 1)$ to the worklist and add EPN $((x, \alpha \# [w], \beta'), l, k, k + 1)$ to the EPN set. If w is nonterminal, find $R = \{r \mid ((w, k), r) \in \mathcal{P}\}$ and extend $\mathcal{G}$ with $((w, k), ((x, \alpha \# [w], \beta'), l))$. If R is empty, **descend** w by adding $((w, [\,], \delta), k, k)$, for all alternates δ of w, to the worklist (if not in $\mathcal{U}$). If R is not empty, **skip** w by adding the descriptors $((x, \alpha \# [w], \beta'), l, r_i)$, for all $r_i \in R$ (if not in $\mathcal{U}$) and adding the EPNs $((x, \alpha \# [w], \beta'), l, k, r_i)$ to the EPN set. If $\beta = [\,]$, extend $\mathcal{P}$ with $((x, l), k)$, and **ascend** x by finding all continuations $K = \{(slot, l') \mid ((x, l), (slot, l')) \in \mathcal{G}\}$, adding $(slot, l', k)$ to the worklist for all $(slot, l') \in K$ (if not in $\mathcal{U}$), adding EPNs $(slot, l', l, k)$ to the EPN set and, if $\alpha = \beta = [\,]$, add EPN $((x, [\,], [\,]), k, k, k)$ as well.

[5]Referred to as the pop-set by Scott and Johnstone [29, 31].
[6]Modelling a simplification of the GSS used by the RGLL algorithm [1, 32].

```
type RList n t ≡ S.Set (Descr n t)
popRList       :: RList n t → (Descr n t, RList n t)
emptyRList     :: RList n t
singletonRList :: Descr n t    → RList n t
unionRList     :: RList n t → RList n t → RList n t
fromListRList  :: [Descr n t] → USet n t → RList n t
fromListRList ds U = foldr op emptyRList ds
  where op d R | hasDescr d U = R
               | otherwise    = unionRList (singletonRList d) R

type USet n t ≡ S.Set (Descr n t)
emptyUSet      :: USet n t
addDescr       :: Descr n t → USet n t → USet n t
hasDescr       :: Descr n t → USet n t → Bool

type GRel n t ≡ S.Set (Comm n t, Cont n t)
emptyG         :: GRel n t
addCont        :: Comm n t → Cont n t → GRel n t → GRel n t
conts          :: Comm n t → GRel n t → [Cont n t]

type PRel n t ≡ S.Set (Comm n t, Int)
emptyP         :: PRel n t
addExtent      :: Comm n t → Int → PRel n t → PRel n t
extents        :: Comm n t → PRel n t → [Int]
```

Figure 1. Types of FUN-GLL data structures and operations.

Data structures The types of the data structures and their operations are given in Figure 1. The efficiency and worst-case complexity of FUN-GLL is strongly influenced by the implementation of the data structures. We described the essential data structures — $\mathcal{R}$, $\mathcal{U}$, $\mathcal{G}$ and $\mathcal{P}$ — as sets, but a direct implementation as HASKELL sets (from *Data.Set*) is inefficient. For example, we need to be able to determine quickly whether a descriptor has already been encountered by inspecting $\mathcal{U}$. This operation needs to be performed in constant time for FUN-GLL to have a worst-case complexity of $O(n^3)$ [18]. We do not discuss actual implementations of these data structures and operations as we focus on the logic of the algorithm instead. We use ≡ in the type definitions to suggest a simple implementation, as we did for *EPNs*.

The operation *popRList* arbitrarily removes an element from worklist $\mathcal{R}$ :: *RList*. The worklist is constructed by applying *fromListRList* to a list of descriptors, guaranteeing that the resulting worklist contains no elements already in the given $\mathcal{U}$ :: *USet*. The relation $\mathcal{G}$ :: *GRel*, between commencements and continuations, is extended by applying *addCont*. Operation *conts* returns all the continuations paired with a given commencement in $\mathcal{G}$. The relation $\mathcal{P}$ :: *PRel*, between commencements and right extents, is extended by applying *addExtent*. Similar to *conts*, *extents* returns all the right extents paired with a particular commencement in $\mathcal{P}$.

The generalised parsing procedure Function *fungll* implements FUN-GLL. It is given a grammar, a nonterminal, and

an input sentence and returns the processed set of descriptors $\mathcal{U}$ and the set of discovered EPNs, recursively applying function *loop* to $\mathcal{R}$ to process one descriptor at a time.

```
fungll gram x inp =
  loop gram inp R emptyUSet emptyG emptyP emptyPNs
where R = fromListRList (descend gram x 0) emptyUSet
```

The initial $\mathcal{R}$ contains the descriptors resulting from 'descending' the given symbol x with left extent 0. Descending a nonterminal x with index k requires the descriptors $((x,[\,],\beta),k,k)$ to be processed, for each alternate β of x.

```
descend gram x k = [((x,[ ],β),k,k) | β ← alts]
  where alts = maybe [ ] S.toList (M.lookup x gram)
```

Function *loop* recurses over $\mathcal{R}$, removing a descriptor d (using *popRList*) for processing, until $\mathcal{R}$ is empty. The order in which descriptors are selected is irrelevant to the correctness (and worst-case complexity) of the algorithm, but might influence efficiency. An analysis of the influence of the order on efficiency can be found in [32].

Processing a descriptor may result in descriptors, which are added to $\mathcal{R}$ if not in $\mathcal{U}$, as well as some EPNs. Processing a descriptor may also result in an extension to $\mathcal{G}$ or $\mathcal{P}$. Thus, *process* returns (a list of) descriptors, a (set of) EPNs, an optional commencement and continuation pair, and an optional commencement and right extent pair. The list of descriptors returned by *process* is converted into an *RList* using *fromListRList*.

```
loop gram inp R U G P ns
  | null R     = (U, ns)    -- base case: R is empty
  | otherwise = loop gram inp R'' U' G' P' (unionPNs ns ns')
  where ((rlist, ns'), mcont, mpop) = process gram inp d G P
        (d, R')  = popRList R
        R''      = unionRList R' (fromListRList rlist U')
        U'       = addDescr d U
        G'       | Just (k, v) ← mcont = addCont k v G
                 | otherwise           = G
        P'       | Just (k, v) ← mpop  = addExtent k v P
                 | otherwise           = P
```

There are two cases to distinguish when a descriptor $((x,\alpha,\beta),l,k)$ is processed: $\beta = w:\beta'$ and $\beta = [\,]$.

```
process gram inp ((x,α,[ ]),l,k) G P =
  ((rlist, unionPNs ns ns'), Nothing, Just ((x,l),k))
  where (rlist, ns) = ascend l K k
        ns' | α ≡ [ ]    = singlePN ((x,[ ],[ ]),l,k,k)
            | otherwise = emptyPNs
        K = [(slot, l') | (slot, l') ← conts (x,l) G]
```

The code above implements the latter case, in which it is discovered that x derives the subsentence ranging from l to $k-1$ and thus that k is a right extent for (x,l). This is 'remembered' by returning the commencement and right extent pair $((x,l),k)$ to extend $\mathcal{P}$. All continuations K, 'waiting' for the discovery of additional right extents such as k, are obtained by applying *conts* to (x,l) and $\mathcal{G}$. The descriptors and EPNs obtained by combining the continuations in K with l and k are returned by *ascend* (given later). The former case, $\beta = w:\beta'$ is implemented by the code below.

```
process gram inp ((x,α,w:β'),l,k) G P = case w of
  Term t →   (match inp ((x,α,w:β'),l,k), Nothing, Nothing)
  Nt y
    | R ≡ [ ] → ((descend gram y k, emptyPNs), Just cc, Nothing)
    | R ≢ [ ] → (skip k ((x,α ⧺ [w],β'),l) R, Just cc, Nothing)
      where R  = extents (y,k) P
            cc = ((y,k),((x,α ⧺ [w],β'),l))
```

When w is a nonterminal symbol, the commencement and continuation pair $((w,k),((x,\alpha \mathbin{+\!\!+} [w],\beta'),l))$ is returned for extending $\mathcal{G}$. Operation *extents* is used to find any right extents $r \in R$, providing the information that w derives the subsentence ranging from l to $r-1$. If $R \equiv [\,]$, w is descended with left extent k (potentially for a second time). Otherwise, if $R \not\equiv [\,]$, function *skip* computes the descriptors and EPNs that follow from the earlier discovery that w derives the subsentence ranging from k to $r-1$. Function *skip* combines a single continuation with perhaps many right extents, whereas *ascend* combines a single right extent with potentially many continuations.

```
skip k d R      = nmatch k [d] R
ascend k K r    = nmatch k K [r]
nmatch k K R = (rlist, fromListPNs elist)
  where rlist = [(slot,l,r)    | (slot,l) ← K, r ← R]
        elist = [(slot,l,k,r) | (slot,l) ← K, r ← R]
```

If $\beta = w:\beta'$, and w is terminal, $match_1$ from Section 3 is applied to check whether w matches the terminal at position k in the input sentence. If so, the descriptor $((x,\alpha \mathbin{+\!\!+} [w],\beta'),l,k+1)$ and the EPN $((x,\alpha \mathbin{+\!\!+} [w],\beta'),l,k,k+1)$ are returned.

```
match inp (slot@(x,α,Term t:β),l,k)
  | match₁ inp t k = ([((x,α ⧺ [Term t],β),l,k+1)]
                      ,singlePN ((x,α ⧺ [Term t],β),l,k,k+1))
  | otherwise      = ([ ], emptyPNs)
```

To show the generality of *fungll* we apply it to one of Ridge's example grammars [28]:

```
tripleE = M.fromListWith S.union
            [("E", S.singleton [Nt "E", Nt "E", Nt "E"])
            ,("E", S.singleton [Term '1'])
            ,("E", S.singleton [])]
```

Grammar *tripleE* is left-recursive and cyclic[7], admitting infinitely many derivations of the sentences in the language it generates. Applied to *tripleE*, nonterminal "E", and the input sentence "1", *fungll* returns the following EPNs:

[7] A cyclic nonterminal can derive itself via one or more steps.

```
type Parser t a = Input t → Int → [(Int, a)]

term1 :: Eq t ⇒ t → Parser t t
term1 t inp k | match1 inp t k = [(k + 1, inp ! k)]
              | otherwise      = [ ]

succeeds :: a → Parser t a
succeeds v inp k = [(k, v)]

fails :: Parser t a
fails inp k = [ ]

infixl 3 ⟨|⟩
( ⟨|⟩ ) :: Parser t a → Parser t a → Parser t a
(p ⟨|⟩ q) inp k = p inp k ++ q inp k

infixl 4 ⟨∗⟩
( ⟨∗⟩ ) :: Parser t (a → b) → Parser t a → Parser t b
(p ⟨∗⟩ q) inp l = [(r, f a) | (k, f) ← p inp l, (r, a) ← q inp k]

run_parser :: Parser t a → Input t → [a]
run_parser p inp = [a | (r, a) ← p inp 0, r ≡ ub + 1]
  where (_, ub) = bounds inp
```

Figure 2. A parser combinator library with semantic actions.

```
> printEPNs (snd (fungll tripleE "E" (listArray (0,0) "1")))
1  : (("E",[ ],[ ]),0,0,0)
2  : (("E",["E"],["E","E"]),0,0,0)
3  : (("E",["E","E"],["E"]),0,0,0)
4  : (("E",["E","E","E","E"],[ ]),0,0,0)
5  : (("E",['1'],[ ]),0,0,1)
6  : (("E",["E"],["E","E"]),0,0,1)
7  : (("E",["E","E"],["E"]),0,0,1)
8  : (("E",["E","E"],["E"]),0,1,1)
9  : (("E",["E","E","E","E"],[ ]),0,0,1)
10 : (("E",["E","E","E","E"],[ ]),0,1,1)
11 : (("E",[ ],[ ]),1,1,1)
12 : (("E",["E"],["E","E"]),1,1,1)
13 : (("E",["E","E"],["E"]),1,1,1)
14 : (("E",["E","E","E","E"],[ ]),1,1,1)
```

In the next sections we develop the BNF combinator library which uses FUN-GLL internally, beginning with a general introduction to parser combinators.

5 Parser Combinators

In the next sections we assume familiarity with parser combinators and embedded DSLs generally. For introductions to these topics we refer the reader to [36] and [37].

Parser combinators have been defined in many ways [10, 17, 21, 36, 42]. In general, a parser combinator is a higher-order function combining one or more parsers, or a higher-order function constructing a parser based on some (non-parser) arguments. A combinator parser is the composition of smaller 'subparsers', and runs the parsers out of which it is composed. In Figure 2, we define a basic parser combinator library as an example.

Applied to a sentence *inp* and a left extent k, a parser $p :: Parser\ t\ a$ returns a list of pairs containing (r, v) when p has recognised the subsentence ranging from k to $r - 1$, and $v :: a$ is the 'semantic interpretation' given to the subsentence by the semantic functions embedded in p. As an example of a parser developed with parser combinators, consider the parser *pTupleRR* defined below, recognising the language generated by `"tuple"` in grammar *tupleRR* (see Sections 2 and 3):

```
pTupleRR :: Parser Char Int
pTupleRR = term1 '(' ∗⟩ pAs ⟨∗ term1 ')'
pAs = succeeds 0 ⟨|⟩ (+1) ⟨$ term1 'a' ⟨∗⟩ pMore
pMore = succeeds 0 ⟨|⟩ (+1) ⟨$ term1 ',' ⟨∗ term1 'a' ⟨∗⟩ pMore
```

The semantic functions compute the length of the tuple. The combinators ⟨∗ and ∗⟩ are variants of ⟨∗⟩ that ignore the semantic value of their right and left argument respectively, $p\ \langle\$\rangle\ q = succeeds\ p\ \langle*\rangle\ q$, and $p\ \langle\$\ q = succeeds\ p\ \langle*\ q$.

Positives There are several advantages to writing parsers with parser combinators. Firstly, a parser combinator library inherits features from the host language in which it is implemented. For example, parsers can be defined within different name spaces, or within different modules, and the host language's type-system type-checks the application of the semantic functions. Secondly, combinator parsers and parser combinators are reusable. Borrowing the abstraction mechanism of the host language, we can abstract over subparsers and replace them with a parameter. For example, *comSep* recognises comma-separated sequences of elements determined by a given parser:

```
comSep :: Parser Char a → Parser Char [a]
comSep p = succeeds [ ]
           ⟨|⟩ (:[ ]) ⟨$⟩ p
           ⟨|⟩ (:)    ⟨$⟩ p ⟨∗ term1 ',' ⟨∗⟩ comSep p
```

A third advantage is that additional elementary parsers are defined easily, if the underlying parser algorithm is simple.

Negatives Straightforward implementations like the one in Figure 2 suffer from the same weaknesses as standard recursive descent: left-recursion causes nontermination and backtracking may result in exponential running times. The first problem is often avoided by refactoring parsers to remove left-recursion. Parser combinators can also be defined to remove left-recursion automatically [2]. Johnson showed that recognition combinators written in continuation-passing style can be extended with memoisation to solve the left-recursion problem [17]. Afroozeh, Izmaylova and Van der Storm, build on Johnson's recognition combinators to develop general parser combinators [16]. Frost, Hafiz, and Callaghan [10] handle left-recursion with a 'curtailment' procedure, making at most as many recursive calls as there are characters remaining in the input sentence. To avoid the

second problem, less naive variations of ⟨|⟩ can be implemented, for example to avoid backtracking by default [21] or to choose an alternative using lookahead [36]. Efficiency can be improved with memoisation as well [11, 26].

The resulting algorithms are more complicated, and may depend on impure methods to detect recursion [5, 13, 23]. As a result, it is more difficult to extend the combinator libraries. In general, it is difficult to reason formally about parsers developed with parser combinators, e.g. determining the language recognised by a combinator parser involves knowledge of the underlying operational details.

Grammar combinators The approaches suggested by Devriese and Piessens [6], Ljunglöf [23], and Ridge [28] have in common that higher-order functions combine grammar fragments. The resulting grammars are given to a standalone parsing procedure which need not be restricted to recursive descent and can indeed by a generalised parsing procedure. A library of such *grammar combinators*[8] can be seen as an embedded implementation of BNF with parameterisable nonterminals [24, 38], piggybacking on the abstraction mechanism of the host language. Like parser combinators, grammar combinators inherit other features of the host language. By using a grammar as a level of indirection, it is possible to reason formally about the language defined by combinator expressions. Moreover, optimisations such as lookahead computation and automatic left-factoring [32] can be realised without interfering with the combinator definitions; the underlying parsing procedure is replaceable.

6 BNF Combinators

In this section we give a preview of the BNF combinator library developed in Sections 7, 8, and 9. Superficially, grammar descriptions developed with BNF combinators are like parsers developed with parser combinators. For example[9]:

```
brackets :: Symb_EX t a → Symb_EX t a
brackets p = term '[' **> p <** term ']'

either :: Symb_EX t a → Symb_EX t b → Symb_EX t (Either a b)
either p q = Left <$$> p <||> Right <$$> q
```

Figure 3 contains simplified signatures of the BNF combinators. The signatures reveal a crucial difference with parser combinators. Combinator parsers consist of only one type of expressions — parsers — whereas the signatures of the BNF combinators refer to three different types of expressions: : symbol expressions ($Symb_{EX}$), choice expressions ($Choice_{EX}$), and sequence expressions (Seq_{EX}).

Symbol expressions represent symbols in BNF, a sequence expression represents a sequences of symbols (an alternate), whereas a choice expressions represents the choice between several alternates. A sequence expressions constructed by

[8]The term used by Ljunglöf and Devriese and Piessens.

[9]We use different infix operators for the combinators to avoid confusion and because Applicative laws [25] cannot be proven generally.

```
term     :: Eq t ⇒ t → Symb_EX t t
infixl 2 <::=>
(<::=>)  :: String → Choice_EX t a → Symb_EX t a
infixl 3 <**>
(<**>)   :: Seq_EX t (a → b) → Symb_EX a → Seq_EX b
seqStart :: a → Seq_EX t a
infixl 4 <||>
(<||>)   :: Choice_EX t a → Seq_EX t a → Choice_EX t a
altStart :: Choice_EX t a
```

Figure 3. Simplified signatures of the BNF combinators.

seqStart represents the empty sequence of symbols. Each application of ⟨**⟩ adds an additional symbol to the end of an already existing sequence. Combinator ⟨**⟩ therefore relates to juxtaposition in a BNF rule. Similarly, a choice expression constructed by *altStart* represents the empty sequence of alternates. Each application of ⟨||⟩ adds an additional alternate to an already existing sequence of alternates. Combinator ⟨||⟩ therefore relates to the | operator in a BNF rule. An application of ⟨::=⟩ groups a sequence of alternates (second argument) under a single nonterminal (first argument). Combinator ⟨::=⟩ therefore relates to the ::= operator of a BNF rule. For example, the BNF rule *opta*::=ϵ | 'a' is represented by the symbol expression (with semantic values *Nothing* and *Just* 'a'):

```
"opta" <::=> altStart <||> seqStart Nothing
                      <||> seqStart Just <**> term 'a'
```

The rich structure of BNF combinator expressions makes it possible to generate grammars without binarisation. The actual signatures and the implementation of the BNF combinators are given in Section 9. The implementation is based on the combinator libraries developed in Section 7 and 8. These combinator libraries are *internal*: their expressions have the same structure as the BNF combinator expressions and are generated from BNF combinator expressions. An internal expression of the first kind evaluates to a grammar, which is given to FUN-GLL to produce a parser. The parser is applied to a input sentence to yield a set of EPNs. An internal expression of the second kind builds a function that applies the semantic functions embedded in the expression by extracting derivation information from the EPN set.

7 Grammar Combinators

In this section we implements a library of internal grammar combinators. The type signatures of the grammar combinators are given in Figure 4. The rich structure of the grammar combinator expressions makes it possible to generate grammars without binarisation.

Grammar generation In P3, a unique nonterminal is generated for each combinator expression by combining the

```
term_gr     :: t → Symb_G n t
nterm_gr    :: n → Choice_G n t → Symb_G n t
altStart_gr :: Choice_G n t
alt_gr      :: Choice_G n t → Seq_G n t → Choice_G n t
seqStart_gr :: Seq_G n t
seq_gr      :: Seq_G n t → Symb_G n t → Seq_G n t
```

Figure 4. Signatures of the grammar combinators.

nonterminals generated for its subexpressions. The grammars generated this way are binarised in the sense that each nonterminal has one alternate with at most two symbols or two alternates with at most one symbol. When grammar descriptions are cyclic, nonterminals need to be inserted to side-step nonterminal generation, thus avoiding nontermination[10]. We also rely on the insertion of nonterminals, via $\langle ::= \rangle$ and $nterm_{gr}$ internally, to avoid nontermination.

The types $Symb_G$, Seq_G, and $Choice_G$ are as follows:

```
type Symb_G l t   = (Symbol l t, GrammarGen l t)
type Seq_G l t    = (Rhs l t, GrammarGen l t)
type Choice_G l t = ([Rhs l t], GrammarGen l t)
```

The first component of each combinator expression is the grammar fragment represented by it, e.g. the first component of a choice expression is the sequence of alternates it represents. The second component of each combinator expression is a 'grammar generator': a function that (possibly) extends a given grammar and a given set of nonterminals. The set of nonterminals is used to detect recursion and to avoid nontermination.

```
type GrammarGen n t =
      (S.Set n, Grammar n t) → (S.Set n, Grammar n t)
```

The definitions of the functions $term_{gr}$, $nterm_{gr}$, $altStart_{gr}$, alt_{gr}, $seqStart_{gr}$ and seq_{gr} are given in Figure 5. Only $nterm_{gr}$ extends the grammar with additional productions, one for each alternate represented by subexpression p. If the nonterminal n occurs in the set of nonterminals encountered so far (nts), no productions are added and the choice expression p is ignored. This is valid because the contribution of each nonterminal to the generated grammar is context-independent[11].

Grammars are obtained from symbol expressions by applying the function *grammarOf*. The generated grammars are augmented with a start symbol.

```
grammarOf :: n → Symb_G n t → Grammar n t
grammarOf start (n, pgen) = snd (pgen (S.empty, gram))
  where gram = M.singleton start (S.singleton [n])
```

[10]This is a crude solution to the well-studied problem of making sharing observable in the implementation of embedded domain-specific languages [5, 13, 23]. We consider it a pragmatic choice for our purposes.
[11]This assumes that nonterminals are inserted uniquely by the user.

```
term_gr t = (Term t, id)
nterm_gr n p = (Nt n, gen)
  where (alts, pgen) = p
        gen (nts, gram) | S.member n nts = (nts, gram)
                        | otherwise      = pgen (nts', gram')
          where nts' = S.insert n nts
                gram' = M.insertWith S.union n
                          (S.fromList alts) gram
altStart_gr = ([], id)
alt_gr (as, pgen) (α, qgen) = (as ++ [α], qgen ∘ pgen)
seqStart_gr = ([], id)
seq_gr (α, pgen) (s, qgen) = (α ++ [s], pgen ∘ qgen)
```

Figure 5. Definitions of the grammar combinators.

```
type Symb_S n t a   = (Symbol n t, OracleParser n t a)
type Seq_S n t a    = (Rhs n t, n → Rhs n t → OracleParser n t a)
type Choice_S n t a = (n → OracleParser n t a)

term_sem     :: t → Symb_S n t t
nterm_sem    :: (Ord n) ⇒ n → Choice_S n t a → Symb_S n t a
altStart_sem :: Choice_S n t a
alt_sem      :: Choice_S n t a → Seq_S n t a → Choice_S n t a
seqStart_sem :: a → Seq_S n t a
seq_sem      :: Seq_S n t (a → b) → Symb_S n t a → Seq_S n t b
```

Figure 6. The signatures of the semantic combinators.

Function *parserFor* applies *fungll* to the grammar generated for a symbol expression and produces a function from an input sentence to set of EPNs.

```
parserFor :: n → Symb_G n t → Input n t → EPNs n t
parserFor start p inp = snd (fungll (grammarOf start p) start inp)
```

8 Semantic Combinators

In this section we describe and implement semantic combinators, based on Ridge's 'semantic phase' [28], modifying it to our setting of non-binarised grammars (we omit memoisation). Ridge explains his combinators as parser combinators 'guided by an oracle', where the oracle — a set of EPNs — provides a pre-computed set of pivots.

```
type OracleParser n t a = EPNs n t → Int → Int → S.Set n → [a]
```

The signatures of the semantic combinators are given in Figure 6. Besides type variables n and t for nonterminals and terminals, the signatures also include the types of semantic values, similar to the parser combinators in Section 5. Besides a set of EPNs, an oracle-guided parser receives a set of nonterminals, a left extent l and a right extent r. The set of nonterminals is used to detect recursion and avoid non-termination, as in Section 7. The left extent and right

```
term_sem t = (Term t, gen)
  where gen ns l r nts | l + 1 ≡ r  = [t]
                       | otherwise = []
nterm_sem x p = (Nt x, gen)
  where gen ns l r nts
            | S.member x nts = []
            | otherwise      = p x ns l r (S.insert x nts)
alt_sem p (α, q) = gen
  where gen x ns l r nts = p x ns l r nts ⧺
                           q x [] ns l r nts
altStart_sem = gen
  where gen x ns l r nts = []
seqStart_sem a = ([], gen)
  where gen x β ns l r nts | l ≡ r = [a]
                           | otherwise = []
seq_sem (α, p) (s, q) = (α ⧺ [s], gen)
  where gen x β ns l r nts =
          [f a | k ← pivots ((x, α ⧺ [s], β), l, r) ns
               , f ← p x (s : β) ns l k (leftLabels nts l k r)
               , a ← q ns k r (rightLabels nts l k r)]
        leftLabels nts l k r  | k < r  = S.empty
                              | otherwise = nts
        rightLabels nts l k r | k > l  = S.empty
                              | otherwise = nts
```

Figure 7. Definitions of the semantic combinators.

extent are used to select pivots from the EPN set. The pivots are obtained by applying the operation *pivots*:

$pivots :: (Slot\ n\ t, Int, Int) \to EPNs\ n\ t \to [Int]$

The semantic combinators compute grammar slots in order to apply *pivots*. The first component of a symbol expression is the symbol represented by the expression. Similarly, the first component of a sequence expression is the sequence of symbols represented by the expression. These are used to compute the required grammar slots. The argument of a choice expression, and the arguments of the second component of a sequence expression, are used for the same purpose as explained later.

The semantics combinators are implemented in Figure 7. The semantic value of $term_{sem}\ t$ is t, just like $term_1$ of Section 5. The second component of a sequence expression receives nonterminal x and a list of symbols β as arguments. The arguments of seq_{sem} provide a list of symbols α and a symbol s (first component). This information is used to extract all the pivots $[k_1, \ldots, k_n]$ from the EPN set such that $((x, \alpha, \beta), l, k_i, r)$ is an EPN in the set, for all $1 \leqslant i \leqslant n$. For each k_i, p is applied with left extent l and right extent k_i to give semantic function $[f_1, \ldots, f_m]$ and q is applied with left extent k_i and right extent r to give semantic values $[a_1, \ldots, a_o]$. The semantic values of the sequence are the result of applying all f_i to all a_j, with $1 \leqslant i \leqslant m$ and

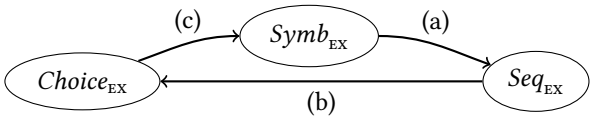

Figure 8. Conversions between types of expressions.

$1 \leqslant j \leqslant o$. Ambiguity reduction is required to keep the number of combinations under control (see also Section 10). The set *nts* is emptied whenever a call is made to an oracle parser with a larger left extent, or a smaller right extent. This implies that only 'bad' derivations of cyclic grammars are ignored (following Ridge's definition of good and bad derivations [28]).

An evaluator is obtained by applying *evaluatorFor* to a symbol expression, initial left and right extents, and EPNs.

$evaluatorFor :: Symb_S\ n\ t\ a \to Int \to Int \to EPNs\ n\ t \to [a]$
$evaluatorFor\ (\_, p)\ l\ r\ ns = p\ ns\ l\ r\ S.empty$

9 Flexible BNF Combinators

This section implements the BNF combinators by applying the combinators discussed in the previous two sections. A brief introduction to the BNF combinators was given in Section 6, but the signatures of the combinators shown there were simplified. The actual signatures present a more general and flexible interface. The flexibility is achieved by automatic conversions between different types of combinator expressions. Using Haskell's type-classes [15], these conversions are realised as implicit coercions.

A BNF combinator expression is a pair of a grammar combinator expression (Section 7) and a semantic combinator expression (Section 8). The types $Symb_{EX}$, $Choice_{EX}$, and Seq_{EX} are defined as follows:

$\textbf{newtype}\ Symb_{EX}\ t\ a = Symb\ (Symb_G\ String\ t, Symb_S\ String\ t\ a)$
$\textbf{newtype}\ Seq_{EX}\ t\ a = Seq\ (Seq_G\ String\ t, Seq_S\ String\ t\ a)$
$\textbf{newtype}\ Choice_{EX}\ t\ a = Ch\ (Choice_G\ String\ t, Choice_S\ String\ t\ a)$

The type of nonterminals is no longer left abstract. Strings are chosen to support nonterminal generation.

Consider the diagram in Figure 8. The edge labelled (a) represents a function $Symb_{EX}\ t\ a \to Seq_{EX}\ t\ a$, converting symbol expressions into sequence expressions. Similarly, edge (b) represents a function $Seq_{EX}\ t\ a \to Choice_{EX}\ t\ a$ and (c) represents a function $Choice_{EX}\ t\ a \to Symb_{EX}\ t\ a$. The core idea of this section is to implement these conversions as methods in a type-class and apply them in combinator definitions. For example, if p and q are symbol expressions, we can then write $p \langle||\rangle q$, which is automatically converted to $altStart \langle||\rangle p \langle||\rangle q$.

The type-classes and instances of Figure 9 implement the conversions (a), (b), and (c) of Figure 8, as well as their compositions. To implement conversion (c), from a choice expression to a symbol expression, we use Ridge's technique,

```
class IsSeq seq where
  toSeq :: Show t ⇒ seq t a → Seq_EX t a
instance IsSeq Seq_EX where
  toSeq = id
instance IsSeq Symb_EX where
  toSeq p = seqStart id ⟨**⟩ p    -- (a)
instance IsSeq Choice_EX where
  toSeq = toSeq ∘ toSymb          -- (c) then (a)
class IsCh ch where
  toCh :: Show t ⇒ ch t a → Choice_EX t a
instance IsCh Choice_EX where
  toCh = id
instance IsCh Seq_EX where
  toCh p = altStart ⟨||⟩ p        -- (b)
instance IsCh Symb_EX where
  toCh = toCh ∘ toSeq             -- (a) then (b)
class IsSymb symb where
  toSymb :: (Show t) ⇒ symb t a → Symb_EX t a
instance IsSymb Symb_EX where
  toSymb = id
instance IsSymb Choice_EX where
  toSymb p = genNt p ⟨::=⟩ p      -- (c)
instance IsSymb Seq_EX where
  toSymb = toSymb ∘ toCh          -- (b) then (c)
```

Figure 9. Implementation of the conversions of Figure 8.

generating a string based on the structure of the choice expression. As shown by the following definitions, symbols are combined with "*" and alternates with "|":

```
showRhs :: (Show n, Show t) ⇒ Rhs n t → String
showRhs [ ]      = "__()"
showRhs (x : xs) = "__(" ++ show x ++ foldr comb "" xs ++ ")"
  where comb s acc = "*" ++ show s ++ acc

showRhss :: (Show n, Show t) ⇒ [Rhs n t] → String
showRhss [ ]      = "__()"
showRhss (x : xs) = "__(" ++ showRhs x ++ foldr comb "" xs ++ ")"
  where comb s acc = "|" ++ showRhs x ++ acc

genNt :: (Show t) ⇒ Choice_EX t a → String
genNt (Ch ((alts, _), _)) = showRhss alts
```

The BNF combinators are defined in Figure 10. Each combinator is defined as a straightforward application of the corresponding grammar combinator and semantic combinator to the subexpressions obtained by converting the combinator's arguments. The interface provided by the BNF combinators is flexible in the sense that the combinators ⟨::=⟩, ⟨||⟩, and ⟨**⟩ can be applied to arbitrary BNF combinator expressions, as conversions between all combinator expressions are available. Perhaps the combinators are too flexible, as recursive combinator expressions can be written without the use of ⟨::=⟩, thus causing nontermination. As a decision in

```
term :: (Show t) ⇒ t → Symb_EX t t
term t = Symb (term_gr t, term_sem t)

infixl 2 ⟨::=⟩
( ⟨::=⟩ ) :: (IsCh ch, Show t) ⇒ String → ch t a → Symb_EX t a
l ⟨::=⟩ p = Symb (nterm_gr l pgram, nterm_sem l psem)
  where Ch (pgram, psem) = toCh p

infixl 3 ⟨||⟩
( ⟨||⟩ ) :: (IsCh ch, IsSeq seq, Show t) ⇒
            ch t a → seq t a → Choice_EX t a
p ⟨||⟩ q = Ch (alt_gr pgram qgram, alt_sem psem qsem)
  where Ch (pgram, psem) = toCh p
        Seq (qgram, qsem) = toSeq q

altStart :: Choice_EX t a
altStart = Ch (altStart_gr, altStart_sem)

infixl 4 ⟨**⟩
( ⟨**⟩ ) :: (IsSeq seq, IsSymb symb, Show t) ⇒
            seq t (a → b) → symb t a → Seq_EX t b
p ⟨**⟩ q = Seq (seq_gr pgram qgram, seq_sem psem qsem)
  where Seq  (pgram, psem) = toSeq p
        Symb (qgram, qsem) = toSymb q

seqStart :: a → Seq_EX t a
seqStart a = Seq (seqStart_gr, seqStart_sem a)
```

Figure 10. BNF combinator definitions with coercions.

the design of the library, we could have ignored conversion (c) — by omitting type-class *IsSymb* and the instances that involve *toSymb*, thereby forcing the user to insert nonterminal names manually with ⟨::=⟩. Instead, we decided to offer the most flexible interface. Being aware of this risk, a user can avoid writing expressions that involve conversion (c).

Function *execute* is given a start nonterminal, a symbol expression, and a sentence, and applies functions *parserFor* and *evaluatorFor* to yield all the interpretations of the sentence:

```
execute :: (Show t, IsSymb s) ⇒ String → s t a → Input t → [a]
execute start s inp =
  evaluatorFor psem 0 (ub + 1) (parserFor start pgram inp)
  where (Symb (pgram, psem)) = toSymb s
        (_ub)                = bounds inp
```

10 Evaluation

The code fragments in this paper are simplified extracts taken from the GLL package available on Hackage [40]. The GLL package is evaluated in this section. In the code fragments, we have omitted several features that improve usability. For example, the actual implementation of FUN-GLL can produce BSPPFs — not just EPNs, and throws error messages to aid debugging.

Most significant are the omission of lookahead and ambiguity reduction. Ambiguity reduction strategies, such as longest match, are implemented in variations of ⟨::=⟩, ⟨||⟩,

$$(\ \langle ::= \rangle_{\text{BIN}}\ ) :: String \to Symb_{\text{EX}}\ t\ a \to Symb_{\text{EX}}\ t\ a$$
$$(\ \langle ::= \rangle_{\text{BIN}}\ ) = (\ \langle ::= \rangle\ )$$
$$(\ \langle || \rangle_{\text{BIN}}\ ) :: Symb_{\text{EX}}\ t\ a \to Symb_{\text{EX}}\ t\ a \to Symb_{\text{EX}}\ t\ a$$
$$p\ \langle || \rangle_{\text{BIN}}\ q = toSymb\ (p\ \langle || \rangle\ q)$$
$$(\ \langle ** \rangle_{\text{BIN}}\ ) :: Symb_{\text{EX}}\ t\ (a \to b) \to Symb_{\text{EX}}\ t\ a \to Symb_{\text{EX}}\ t\ b$$
$$p\ \langle ** \rangle_{\text{BIN}}\ q = toSymb\ (p\ \langle ** \rangle\ q)$$
$$succeeds_{\text{BIN}} :: a \to Symb_{\text{EX}}\ t\ a$$
$$succeeds_{\text{BIN}} = toSymb \circ seqStart$$

Figure 11. Binarising BNF combinators.

and $\langle ** \rangle$ by filtering the pivots extracted from an EPN set during the semantic phase. Configuration options enable ambiguity reduction strategies globally.

Lookahead is performed in the way described by Scott and Johnstone [31]. Functions *descend* and *ascend* yield only the descriptor $((x, \alpha, \beta), l, k)$ if the k-th terminal of the input is in the lookahead-set pre-computed for (x, α, β).

The GLL package exports two modules — *Interface* and *BinaryInterface* — defining the same BNF combinators. The combinators of *Interface* are defined as in Section 9. The combinators of *BinaryInterface* specialise the combinators of *Interface*, converting all expressions to symbol expressions, as shown by Figure 11. Because *toSymb* is used in the definitions of $\langle || \rangle_{\text{BIN}}$ and $\langle ** \rangle_{\text{BIN}}$, the grammars generated by the underlying grammar combinator expressions are binarised in the same sense as P3's internal grammars. If a syntax description is originally written with the combinators of *BinaryInterface*, one can change the import of *BinaryInterface* to *Interface* and the description is still valid. In this section we take advantage and evaluate these two modules of the GLL package, demonstrating the effects of grammar binarisation and lookahead on running FUN-GLL.

Parser evaluation We evaluate the syntax descriptions of three software languages written with the BNF combinators: two programming languages — ANSI-C and Caml Light — and the semantic specification language CBS [3]. The running times include a lexicalisation phase which produces a sequence of 'tokens', given as input to the parsing phase. For each language we selected considerable software-language engineering projects: a parser generator in ANSI C, a Caml Light compiler in Caml Light, and a complete semantic specification in CBS. The test files are the result of composing varying selections of source files taken from these projects. The tests have been executed on a laptop with quad-core 2.4GHz processors and 8GiB of RAM, under Ubuntu 14.04.

The data for ANSI-C is shown in Table 1. The syntax description is a direct transcription of the grammar listed by Kernighan and Ritchie [20]. This grammar is written in BNF (without extensions). The grammar is nondeterministic and left-recursive. The internal grammar given to FUN-GLL has 229 alternates and 71 nonterminals. When written with *BinaryInterface*, a large number of alternates and nonterminals is generated: the internal grammar has 848 alternates and 690 nonterminals. The running times of FUN-GLL are strongly affected by the binarisation of the grammar, differing with a factor between 2.4 and 2.6 with lookahead and between 2.5 and 3.1 without lookahead. Lookahead has a significant effect on the largest files when the grammar is binarised. A visualisation of the data in Table 1 is given in Figure 12 of Appendix A.

Tokens	1515	8411	15589	26551	36827
Flexible	0.44	2.46	4.83	7.86	10.40
+lookahead	0.50	2.77	4.75	7.23	10.27
Binarised	1.12	7.17	13.47	22.47	32.41
+lookahead	1.31	6.67	12.02	18.9	25.30

Table 1. Parsing ANSI-C files (in seconds).

Tokens	1097	2808	4531	8832	15900	28674
Flexible	1.28	2.69	4.17	8.13	13.90	28.36
+lookahead	1.41	2.75	4.28	9.12	15.42	28.15
Binarised	1.90	4.58	7.33	14.40	25.00	–
+lookahead	1.80	3.56	6.10	11.86	19.90	38.79

Table 2. Parsing Caml Light files (in seconds).

Tokens	2653	14824	17593	21162	26016
Flexible	1.36	12.01	15.24	20.17	29.54
+lookahead	1.10	5.56	6.54	7.86	9.57
Binarised	2.94	41.83	–	–	–
+lookahead	2.31	9.51	11.38	13.48	16.62

Table 3. Parsing and pretty-printing CBS files (in seconds).

The data in Table 2 (Figure 13) shows the running times for Caml Light. The grammar is taken from the Caml Light reference manual by Leroy [22]. The grammar is highly nondeterminstic and has many sources of ambiguity. In particular, the grammar contains a large and highly nondeterministic nonterminal for expressions. The combinator description of the grammar makes heavy use of abstraction to capture EBNF notations, and some coercions are used when written with *Interface*. The difference in size between the grammars given to FUN-GLL are therefore smaller: 285 versus 731 alternates and 134 versus 580 nonterminals. The negative effect of grammar binarisation is also smaller, between 1.3 and 1.4 with lookahead and between 1.4 and 1.8 without lookahead. The running time on the largest file is missing for *BinaryInterface* without lookahead, due to memory-overflow.

Evaluating the semantic phase The data in Tables 1 and 2 exclude the semantic phase. The data in Table 3 (visualised in Figure 14) shows the running times of parsing, applying the

semantics phase with disambiguation, and pretty-printing the resulting abstract syntax tree. The effect of binarising the grammar is not as big as in the case of ANSI-C: between 1.7 and 2.1 with lookahead (without lookahead omitted). The grammar given to FUN-GLL by *Interface* has 257 alternates and 126 nonterminals, versus 771 alternates and 640 nonterminals by *BinaryInterface*. The effect of lookahead is dramatic and lookahead is required to keep the running times under control as the input grows.

11 Conclusion

This paper presents a library of 'BNF combinators' with which general context-free grammars can be described. Internal grammars are generated from the description and are given to FUN-GLL, a purely functional, general, and top-down parsing procedure. The BNF combinators take advantage of HASKELL's strong type-system to type-check semantic actions, of type-classes to present a flexible user-interface, and of its abstraction mechanism to define functions combining grammar fragments, enabling 'reuse through abstraction'.

Experience has shown that the BNF combinator library is practical and easy to use. We have used the library to describe the syntax of several software languages. The syntax descriptions are easy to develop, verify and debug. Without specialised engineering, FUN-GLL shows acceptable running times on these grammars. For us, the benefits of developing syntax without having to consider left-factoring or left-recursion removal are certainly worth the price of generalised parsing. Moreover, if parsing speed is essential, the descriptions can be refactored for efficiency.

There are two main caveats concerning the usability of the BNF combinators. Firstly, as a language evolves, it is hard to keep track which nonterminals have already been used across its syntax description. When defining a recursive function that combines grammar fragments, care must be taken to ensure that the inserted nonterminal reflects the parameters. As a pure alternative to nonterminal insertion, Devriese and Piessens suggest primitive recursion constructs defined with datatype generic programming [7]. Impure alternatives typically involve automatic generation of references for nonterminals [5, 13, 23].

Secondly, disambiguation is required for ambiguous grammars before or during semantic evaluation. The current ambiguity reductions strategies are low-level, defined directly on EPN sets, and may not comfortably deal with certain ambiguities. Further research is required to determine which high-level strategies are necessary, and to discover how these strategies are realised by filtering EPN sets.

12 Related Work

Throughout this paper we made comparisons with Ridge's P3 library. In Section 5 we have discussed the work of several authors on parser combinators.

GLL parsing Since generalised top-down (GLL) parsing emerged [29, 31], several GLL algorithms have been published [1, 32]. These algorithms are described in a low-level pseudo-language as the output of a parser generator. We describe FUN-GLL in a functional setting at a higher level of abstractions by abstracting over the grammar and by modelling the data structures as sets with basic operations.

Spiewak has also adapted GLL to a functional setting by defining 'GLL combinators' in SCALA [34]. Spiewak's GLL combinators and FUN-GLL both use a 'trampoline' to loop through descriptors for processing. The GLL combinators apply semantic actions on the fly, without collecting derivation information. We expect that it is necessary to use a data structure, such as an EPN set, as an intermediary between parsing and semantic evaluation to ensure at least one derivation is preserved by disambiguation.

We view FUN-GLL as a high-level, functional description of RGLL [32], bearing a striking resemblance to Johnson's recognition combinators. The connection between Johnson's combinators and GLL has also been observed by Afroozeh and Izmaylova [1]. The algorithms have in common that for each call to the parse function of a nonterminal x with left extent l (descending x with l) right extents and continuations are recorded to ensure all derivations are discovered.

Grammar combinators A grammar combinator library can be seen as an embedded DSL for describing syntax, generating parsers at run-time. In this light, a parser combinator library provides a shallow embedding, whereas a grammar combinator library provides a deep embedding. Theoretically, shallow and deep embeddings are closely related [12], but in practice implementations differ significantly. A shallow embedding is usually easier to extend, and its implementation more succinct. In a deep embedding it is easier to perform program transformations and pre-computation. Techniques have been developed to overcome these differences [4, 35]. Devriese and Piessens show how grammar combinators are defined in 'finally tagless' style [4, 7].

Duregård and Jansson have developed an 'embedded parser generator' library with meta-programming, in which Template Haskell code defines a grammar for which a parser is generated at compile-time [8]. We expect these techniques are applicable to the BNF combinators, removing the constant run-time overhead of generating a grammar and computing lookahead sets. Devriese and Piessens have used Template Haskell to perform grammar transformation on the grammars generated by their combinators at compile-time [6].

Acknowledgments

The authors would like to thank the anonymous reviewers who gave detailed, insightful and helpful comments on earlier versions of this paper.

References

[1] Ali Afroozeh and Anastasia Izmaylova. 2015. Faster, Practical GLL Parsing. In *Compiler Construction*. Springer Berlin Heidelberg, 89–108.

[2] Arthur I. Baars and S. Doaitse Swierstra. 2004. Type-safe, Self Inspecting Code. In *Proceedings of the 2004 ACM SIGPLAN Workshop on Haskell (Haskell '04)*. ACM, 69–79.

[3] L. Thomas van Binsbergen, Neil Sculthorpe, and Peter D. Mosses. 2016. Tool Support for Component-based Semantics. In *Companion Proceedings of the 15th International Conference on Modularity (MODULARITY Companion 2016)*. ACM, 8–11.

[4] Jacques Carette, Oleg Kiselyov, and Chung-Chieh Shan. 2007. Finally Tagless, Partially Evaluated: Tagless Staged Interpreters for Simpler Typed Languages. In *Proceedings of the 5th Asian Conference on Programming Languages and Systems (APLAS'07)*. 222–238.

[5] Koen Claessen and David Sands. 1999. Observable sharing for functional circuit description. In *In Asian Computing Science Conference*. Springer Verlag, 62–73.

[6] Dominique Devriese and Frank Piessens. 2011. Explicitly Recursive Grammar Combinators. In *Proceedings of the 13th International Symposium on Practical Aspects of Declarative Languages*. Springer Berlin Heidelberg, 84–98.

[7] Dominique Devriese and Frank Piessens. 2012. Finally Tagless Observable Recursion for an Abstract Grammar Model. *Journal of Functional Programming* 22, 6 (2012), 757–796.

[8] Jonas Duregård and Patrik Jansson. 2011. Embedded Parser Generators. In *Proceedings of the 4th ACM Symposium on Haskell (Haskell '11)*. ACM.

[9] Jay Earley. 1970. An Efficient Context-free Parsing Algorithm. *Communcations of the ACM* 13, 2 (Feb. 1970), 94–102.

[10] Richard A. Frost, Rahmatullah Hafiz, and Paul Callaghan. 2008. Parser Combinators for Ambiguous Left-Recursive Grammars. In *Practical Aspects of Declarative Languages*. Springer Berlin Heidelberg, 167–181.

[11] Richard A. Frost and Barbara Szydlowski. 1996. Memoizing Purely Functional Top-down Backtracking Language Processors. *Science of Computer Programming* 27, 3 (1996), 263–288.

[12] Jeremy Gibbons and Nicolas Wu. 2014. Folding Domain-specific Languages: Deep and Shallow Embeddings (Functional Pearl). In *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming (ICFP '14)*. 339–347.

[13] Andy Gill. 2009. Type-safe Observable Sharing in Haskell. In *Proceedings of the 2Nd ACM SIGPLAN Symposium on Haskell (Haskell '09)*. ACM, 117–128.

[14] Dick Grune and Ceriel Jacobs. 2010. *Parsing Techniques: A Practical Guide* (2nd ed.). Springer Publishing Company, Incorporated.

[15] Cordelia V. Hall, Kevin Hammond, Simon L. Peyton Jones, and Philip L. Wadler. 1994. Type classes in Haskell. In *European Symposium on Programming (LNCS 788)*. Springer Verlag, 241–256.

[16] Anastasia Izmaylova, Ali Afroozeh, and Tijs van der Storm. 2016. Practical, General Parser Combinators. In *Proceedings of the 2016 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation (PEPM 2016)*. ACM, 1–12.

[17] Mark Johnson. 1995. Memoization in Top-down Parsing. *Computational Linguistics* 21, 3 (1995), 405–417.

[18] Adrian Johnstone and Elizabeth Scott. 2011. Modelling GLL Parser Implementations. In *Software Language Engineering*. Springer Berlin Heidelberg, 42–61.

[19] Lennart C. L. Kats and Eelco Visser. 2010. The Spoofax Language Workbench: Rules for Declarative Specification of Languages and IDEs. In *International Conference on Object Oriented Programming Systems Languages and Applications*. ACM, 444–463.

[20] Brian W. Kernighan and Dennis M. Ritchie. 1988. The C Programming Language. Prentice Hall, Appendix 13.

[21] Daan Leijen and Erik Meijer. 1999. Domain Specific Embedded Compilers. In *Proceedings of the 2Nd Conference on Domain-specific Languages (DSL '99)*. ACM, 109–122.

[22] Xavier Leroy. 1997. Caml Light Manual. (1997). http://caml.inria.fr/pub/docs/manual-caml-light.

[23] Peter Ljunglöf. 2002. *Pure Functional Parsing*. Ph.D. Dissertation. Chalmers University of Technology and Göteborg University.

[24] Simon Marlow and Andy Gill. 2001. Happy - The Parser Generator for Haskell. https://www.haskell.org/happy/. (2001). [Online, accessed 10-July-2018].

[25] Conor McBride and Ross Paterson. 2008. Applicative Programming with Effects. *Journal of Functional Programming* 18, 1 (2008), 1–13.

[26] Peter Norvig. 1991. Techniques for Automatic Memoization with Applications to Context-free Parsing. *Computational Linguistics* 17, 1 (1991), 91–98.

[27] Chris Okasaki and Andrew Gill. 1998. Fast Mergeable Integer Maps. In *In Workshop on ML*. 77–86.

[28] Tom Ridge. 2014. Simple, Efficient, Sound and Complete Combinator Parsing for All Context-Free Grammars, Using an Oracle. In *Software Language Engineering*. Springer International Publishing, 261–281.

[29] Elizabeth Scott and Adrian Johnstone. 2010. GLL Parsing. *Electronic Notes in Theoretical Computer Science* 253, 7 (2010), 177 – 189. Proceedings of the Ninth Workshop on Language Descriptions Tools and Applications (LDTA 2009).

[30] Elizabeth Scott and Adrian Johnstone. 2010. Recognition is Not Parsing - SPPF-style Parsing from Cubic Recognisers. *Science of Computer Programming* 75, 1-2 (2010), 55–70.

[31] Elizabeth Scott and Adrian Johnstone. 2013. GLL parse-tree generation. *Science of Computer Programming* 78, 10 (2013), 1828 – 1844.

[32] Elizabeth Scott and Adrian Johnstone. 2016. Structuring the {GLL} parsing algorithm for performance. *Science of Computer Programming* 125 (2016), 1 – 22.

[33] Elizabeth Scott, Adrian Johnstone, and Rob Economopoulos. 2007. BRNGLR: a cubic Tomita-style GLR parsing algorithm. *Acta Informatica* 44, 6 (2007), 427–461.

[34] Daniel Spiewak. 2012. Scala GLL combinators GitHub Repository. https://github.com/djspiewak/gll-combinators. (2012). [Online; accessed 05-July-2018].

[35] Josef Svenningsson and Emil Axelsson. 2013. Combining Deep and Shallow Embedding for EDSL. In *Trends in Functional Programming*. 21–36.

[36] S.Doaitse Swierstra. 2009. Combinator Parsing: A Short Tutorial. In *Language Engineering and Rigorous Software Development*. Springer Berlin Heidelberg, 252–300.

[37] S. Doaitse Swierstra, Pablo R. Azero Alcocer, and João Saraiva. 1999. Designing and implementing combinator languages. In *Advanced Functional Programming*. Springer Berlin Heidelberg, 150–206.

[38] Breuer Peter T. and Bowen Jonathan P. 1995. A prettier compiler-compiler: Generating higher-order parsers in C. *Software: Practice and Experience* 25, 11 (1995), 1263–1297.

[39] Masaru Tomita. 1985. *Efficient Parsing for Natural Language: A Fast Algorithm for Practical Systems*. Kluwer Academic Publishers.

[40] L. Thomas van Binsbergen. 2015. GLL Hackage Repository. https://hackage.haskell.org/package/gll. (2015). [Online, accessed 10-July-2018].

[41] Mark G. J. van den Brand, J. Heering, P. Klint, and P. A. Olivier. 2002. Compiling Language Definitions: The ASF+SDF Compiler. *ACM Trans. Program. Lang. Syst.* 24, 4 (2002), 334–368.

[42] Philip Wadler. 1985. How to Replace Failure by a List of Successes. In *Proc. Of a Conference on Functional Programming Languages and Computer Architecture*. Springer-Verlag New York, Inc., 113–128.

A Appendix – Evaluation

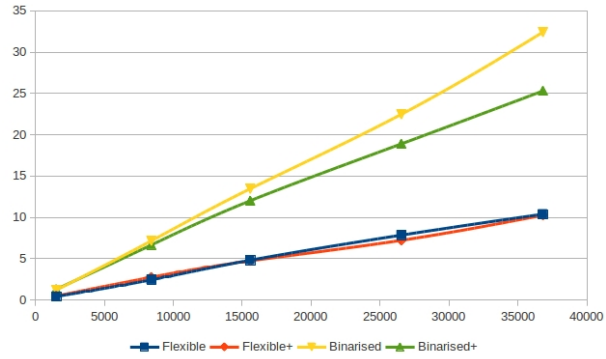

Figure 12. Visualisation of the data in Table 1.

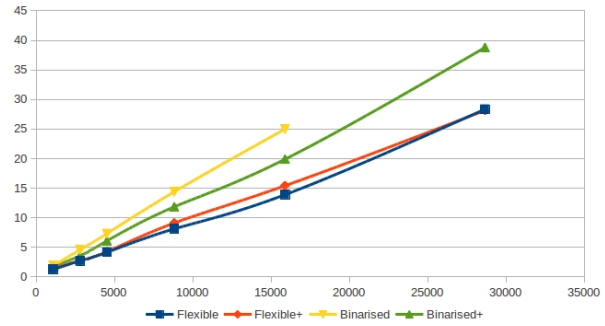

Figure 13. Visualisation of the data in Table 2.

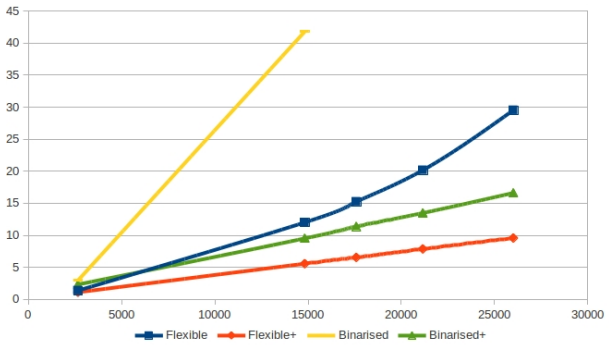

Figure 14. Visualisation of the data in Table 3.