Purely Functional GLL Parsing

L. Thomas van Binsbergen[a,b,*], Elizabeth Scott[a], Adrian Johnstone[a]

[a] *Department of Computer Science, Royal Holloway, University of London, TW20 0EX, Egham, United Kingdom*
[b] *Software Analysis and Transformation, Centrum Wiskunde & Informatica, P.O. Box 94079, 1090 GB, Amsterdam, The Netherlands*

Abstract

Generalised parsing has become increasingly important in the context of software language design and several compiler generators and language workbenches have adopted generalised parsing algorithms such as GLR and GLL. The original GLL parsing algorithms are described in low-level pseudo-code as the output of a parser generator. This paper explains GLL parsing differently, defining the FUN-GLL algorithm as a collection of pure, mathematical functions and focussing on the logic of the algorithm by omitting implementation details. In particular, the data structures are modelled by abstract sets and relations rather than specialised implementations. The description is further simplified by omitting lookahead and adopting the binary subtree representation of derivations to avoid the clerical overhead of graph construction.

Conventional parser combinators inherit the drawbacks from the recursive descent algorithms they implement. Based on FUN-GLL, this paper defines generalised parser combinators that overcome these problems. The algorithm is described in the same notation and style as FUN-GLL and uses the same data structures. Both algorithms are explained as a generalisation of basic recursive descent algorithms. The generalised parser combinators of this paper have several advantages over combinator libraries that generate internal grammars. For example, with the generalised parser combinators it is possible to parse larger permutation phrases and to write parsers for languages that are not context-free.

The 'BNF combinator library' is built around the generalised parser combinators. With the library, embedded and executable syntax specifications are written. The specifications contain semantic actions for interpreting programs and constructing syntax trees. The library takes advantage of Haskell's type-system to type-check semantic actions and Haskell's abstraction mechanism enables 'reuse through abstraction'. The practicality of the library is demonstrated by running parsers obtained from the syntax descriptions of several software languages.

Keywords: top-down parsing, generalised parsing, parser combinators, syntax descriptions, functional programming

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 generate parsers for different classes of context-free grammars. Generalised parsing algorithms such as Earley [1], GLR [2] and GLL [3, 4] admit all context-free grammars and compute all possible derivations of a string [5, 6]. Generalised parsing eases the design of programming languages because, compared to traditional techniques, one no longer needs to adjust the concrete syntax of the language to allow a certain parsing technique. In particular, the concrete syntax can be made more abstract if an appropriate disambiguation strategy is available. Several software language development frameworks take advantage of generalised parsing. For example, SPOOFAX [7] and the $\mathbb{K}$ framework [8] are based on SDF [9], a formalism for describing context-free grammars for which GLR parsers are generated. The meta-programming language RASCAL generates GLL parsers from its syntax definitions [10].

The first part of this paper describes FUN-GLL, a purely functional GLL parsing algorithm given as a collection of mathematical functions. In contrast to the original descriptions of GLL parsing [3, 4], the data structures used by the algorithm are modelled by abstract sets and relations rather than specialised implementations. The description thereby focusses on the general logic of the algorithm and it is not interspersed with specific implementation details. The algorithm is simplified further by omitting lookahead and adopting the binary subtree representation (BSR) of [11] to collect derivation information. The binary subtree representation makes it possible to collect all derivations of an input string without the need for maintaining and constructing graphs. To aid understanding, the algorithm is explained as a generalisation of basic, well-known recursive descent parsing algorithms. In [12],

*Corresponding author
Email addresses: ltvanbinsbergen@acm.org (L. Thomas van Binsbergen), elizabeth.scott@rhul.ac.uk (Elizabeth Scott), adrian.johnstone@rhul.ac.uk (Adrian Johnstone)

the FUN-GLL algorithm was given as a Haskell implementation. The mathematical description of FUN-GLL in this paper is intended to appeal to a wider audience. Compared to [12], more detail is provided in relation to the representation of derivations with BSR sets[1].

In languages with higher-order functions, 'parser combinators' can be defined as functions that take parsers as arguments, combining them to form new parsers. The basic parser combinator approach suffers from nontermination due to left-recursion and from inefficiencies due to backtracking. 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 [13], lookahead can reduce backtracking [14], sophisticated memoisation can handle left-recursion [15], and left-recursion can be removed automatically [16]. Grammar combinator libraries extract grammars from combinator expressions before giving the grammars to a stand-alone parsing procedure [17, 18] that may be generalised [19].

Based on FUN-GLL, the second part of the paper defines generalised parser combinators. The combinator expressions formed by applying these combinators represent BNF grammar descriptions explicitly. This makes it possible for the underlying algorithm to compute the grammar information necessary for generalised parsing. The algorithm is described in the same notation and style as FUN-GLL and uses the same data structures. The combinators are defined first as recognition combinators with a simple recursive descent strategy and are subsequently modified to produce BSR sets and finally to employ all the data structures of FUN-GLL. The evaluation section of this paper demonstrates that the resulting generalised parser combinators have several advantages over grammar combinators. For example, with the generalised parser combinators it is possible to parse large permutation phrases [20, 21] and it is also possible to write parsers for languages that are not context-free. The generalised parser combinators are novel and were not included in [12].

The third part of this paper revisits the 'BNF combinator library' of [12]. The library is an embedded implementation of the BNF formalism. The syntax of languages is specified by writing combinator expressions that represent BNF grammars directly. The specifications contain semantic actions for associating semantics with grammar fragments in order to interpret input strings or to construct syntax trees. In [12], an implementation of the BNF combinator library is given that uses a grammar combinator library internally. Compared to other grammar combinator libraries, the extracted grammars are not binarised. The performance benefits of avoiding binarisation are demonstrated in [12].

This paper replaces the internal grammar combinators with the generalised parser combinators, resulting in an alternative implementation of the BNF combinator library. The alternative implementation is compared with the original implementation in the evaluation section of this paper, demonstrating the aforementioned advantages of parser combinators over grammar combinators. The practicality of the BNF combinator library is evaluated with parsers obtained from syntax descriptions of ANSI-C [22], Caml Light [23], and CBS [24].

In summary, this paper makes the following contributions on top of the original contributions in [12]:

- The FUN-GLL algorithm is described in mathematical notation rather than as a concrete implementation in order to appeal to a wide audience, whereas a Haskell implementation of FUN-GLL is given in [12]
- Purely functional, generalised parser combinators are defined based on the FUN-GLL algorithm. The parser combinators are general in the sense that they admit the full class of context-free languages
- The evaluation section of [12] demonstrates the advantages of avoiding binarisation whilst constructing a grammar in the BNF combinator library. An alternative implementation of the BNF combinator library, based on the generalised parser combinators, avoids grammar generation altogether. This paper demonstrates that, by avoiding grammar generation, practical parsers can be obtained for syntax descriptions that would otherwise produce exponentially large or infinitely large grammars

2. Grammars and derivations

This section formalises context-free grammars and introduces most of the related concepts and notations used throughout this paper.

Given distinct sets N of nonterminal symbols and T of terminal symbols, a context-free grammar Γ is a pair $\langle Z, \rho \rangle$ with $Z \in N$ (the starting nonterminal of the grammar), $\rho \subset N \times S^*$ a finite relation (the production rules of the grammar) and $S = N \cup T$ the set of all symbols. Possibly decorated variables X, Y denote nonterminals, Z a starting nonterminal, s denotes a (nonterminal or terminal) symbol, t a terminal, α, β, δ denote sequences of symbols, τ denotes a sequence of terminal symbols (also referred to as a t-string), and ϵ the empty sequence. A sequence of symbols α is an alternate of nonterminal Y if $\langle Y, \alpha \rangle \in \rho$. The set $\Gamma(Y) = \{\alpha \mid \langle Y, \alpha \rangle \in \rho \textbf{ with } \langle Z, \rho \rangle = \Gamma\}$ contains the alternates of Y.

The relation $\rightarrow_\Gamma \subset S^* \times S^*$ captures derivations in Γ:

$$\frac{}{\alpha X \beta \rightarrow_\Gamma \alpha \delta \beta} \ (\delta \in \Gamma(X)) \tag{1}$$

The relations $\rightarrow_\Gamma^+$ and $\rightarrow_\Gamma^*$ are the transitive closure and the reflexive and transitive closure of $\rightarrow_\Gamma$ respectively. A derivation in Γ is a sequence $\alpha_0, \ldots, \alpha_n$ such that for all $1 \leqslant i \leqslant n$ it holds that $\alpha_{i-1} \rightarrow_\Gamma \alpha_i$. A derivation $\alpha_0 \ldots \alpha_n$

[1] BSR sets were referred to as sets of extended packed nodes in [12].

Tuple ::= `"("` **As** `")"`
As ::= ϵ | `"a"` **More**
More ::= ϵ | `","` `"a"` **More**

Figure 1: A BNF syntax description defining the grammar $\Gamma_{\mathbf{Tuple}}$.

is left-most if for all $1 \leqslant i \leqslant n$ it holds that $\alpha_{i-1} = \tau X \beta$ and $\alpha_i = \tau\delta\beta$ (for some τ and β) with $\tau \in T^*$. (If Rule (1) has the additional side condition $\alpha \in T^*$, then all derivations are left-most.) A nonterminal $X \in N$ is said to derive t-string τ in grammar Γ if there is a derivation from X to τ, i.e. $X \rightarrow_\Gamma^* \tau$. The language $\mathcal{L}_\Gamma(X) = \{\tau \mid \tau \in T^*, X \rightarrow_\Gamma^* \tau\}$ is the set of t-strings that can be derived from a nonterminal X in Γ. The language $\mathcal{L}_\Gamma$ generated by the grammar Γ is the language generated by the starting symbol of Γ, i.e. $\mathcal{L}_\Gamma = \mathcal{L}_\Gamma(Z)$ if $\Gamma = \langle Z, \rho\rangle$.

The grammar $\Gamma_{\mathbf{Tuple}}$ is defined in Figure 1 as an example. The example is given in Backus-Naur Form (BNF) [25]. The connection between BNF descriptions and the formal definition of grammars is well-known. Nonterminals are written in boldface and terminals are written as a single character in quotes, e.g. `"a"`. For brevity, terminals adjacent in a t-string may be written together within the same pair of quotes, e.g. `",a"` instead of `","` `"a"`. The nonterminal **Tuple** generates the language {`"()"`, `"(a)"`, `"(a,a)"`, ...} in $\Gamma_{\mathbf{Tuple}}$.

Recognisers and parsers. A recognition procedure is an algorithm that given a grammar Γ, a nonterminal X, and a t-string τ, determines whether τ is in $\mathcal{L}_\Gamma(X)$, i.e. determines whether $X \rightarrow_\Gamma^* \tau$. A parsing procedure is a recognition procedure that provides a proof for $X \rightarrow_\Gamma^* \tau$, if it holds, typically in the form of a parse tree. This paper considers procedures that read t-strings from left to right and produce left-most derivations (LL procedures).

A recognition or parsing procedure is general if it terminates and gives correct results for all grammars. An LL parsing procedure is complete if it is general and provides proof for all possible left-most derivations of the t-string. A nonterminal X is ambiguous in a grammar Γ if there is a t-string τ for which multiple left-most derivations show that X derives τ in Γ.

Complete parsing procedures require sophisticated technology to efficiently represent all possible derivations of a t-string. Grammars exist with t-strings that have exponentially many derivations (in the length of the t-string) and cyclic nonterminals can derive t-strings in infinitely many ways. (A nonterminal X is cyclic in Γ if $X \rightarrow_\Gamma^+ X$.) Several technologies for representing multiple derivations efficiently are discussed at length in [6, 11]. This paper adopts the binary subtree representation approach of [11] in which a set Υ efficiently embeds (potentially infinitely many) derivations.

2.1. Binary subtree representations

The following summarises, with some modifications, the technical material related to binary subtree representations (BSRs) in [11].

The t-string $\tau^{l,r}$ denotes the subsequence of τ ranging from l to $r-1$, i.e. $\tau^{l,r} = \tau_2$ if $\tau = \tau_1\tau_2\tau_3$ and $|\tau_1| = l$ and $|\tau_1\tau_2| = r$. If a sequence of symbols α derives $\tau^{l,r}$ in a grammar Γ, i.e. $\alpha \rightarrow_\Gamma^* \tau^{l,r}$, then it is said that α derives the substring of τ with *left extent* l and *right extent* r.

A binary subtree representation (BSR) is a structure of the form $\langle g, l, k, r\rangle$, where $l \leqslant k \leqslant r$ are natural numbers, l and r are left and right extents respectively, and g is a grammar slot[2]. A grammar slot is a triple of the form $\langle X, \alpha, \beta\rangle$, denoted[3] as $X ::= \alpha \bullet \beta$, such that $\langle X, \alpha\beta\rangle \in \Gamma(X)$.

Potentially many derivations of a t-string τ in a grammar Γ are embedded in a set Υ of BSRs. For Υ to show that nonterminals derive substrings of τ in Γ, the following must hold:

- An element of the form $\langle X ::= \alpha s \bullet \beta, l, k, r\rangle$ can only be in Υ if:
  - when $\alpha \neq \epsilon$, there must be an element of the form $\langle X ::= \alpha \bullet s\beta, l, k', k\rangle$ in Υ. If X is ambiguous in Γ, this may be the case for several choices of k'
  - when s is a terminal, then s is at position k of τ and $r = k + 1$
  - when s is a nonterminal, then it can be shown that $s \rightarrow_\Gamma^* \tau^{k,r}$ holds according to Υ
- To show that $X \rightarrow_\Gamma^* \tau^{l,r}$ holds according to Υ, when X is nonterminal, an element $\langle X ::= \delta \bullet, l, k, r\rangle \in \Upsilon$ is required, for some k and $\delta \in \Gamma(X)$. The existence of such an element may arise from more than one combination of δ and k if X is ambiguous in Γ
- An element of the form $\langle X ::= \bullet \beta, l, k, r\rangle$ is only in Υ if $l = k$ and if $\beta \in \Gamma(X)$

A BSR set Υ for which these properties hold is sound in the sense that it embeds only derivations of substrings of τ that are valid with respect to Γ. Figure 2 gives a sound BSR set for the grammar $\Gamma_{\mathbf{tuple}}$ that embeds the following derivation of the t-string `"(a,a)"`:

$\mathbf{Tuple}^{0,5}$
$\rightarrow$ `"("`^{0,1} $\mathbf{As}^{1,4}$ `")"`^{4,5} {(5), (4), (1)}
$\rightarrow$ `"("` `"a"`^{1,2} $\mathbf{More}^{2,4}$ `")"` {(9), (7)}
$\rightarrow$ `"(a"` `","`^{2,3} `"a"`^{3,4} $\mathbf{More}^{4,4}$ `")"` {(13), (12), (11)}
$\rightarrow$ `"(a,a)"` {(14)}

[2]This simplification to the BSRs of [11] permits less sharing.
[3]Grammar slots are not written with ϵ, i.e. $X ::= \bullet\beta$, $X ::= \alpha\bullet$, and $X ::= \bullet$ denote slots with the empty sequence as the second, third, and second and third component respectively.

$$
\begin{aligned}
\{&\langle \textbf{Tuple} ::= \texttt{"("} \bullet \textbf{As}\ \texttt{")"}, 0, 0, 1\rangle, && (1)\\
&\langle \textbf{Tuple} ::= \texttt{"("}\ \textbf{As} \bullet \texttt{")"}, 0, 1, 1\rangle, && (2)\\
&\langle \textbf{Tuple} ::= \texttt{"("}\ \textbf{As} \bullet \texttt{")"}, 0, 1, 2\rangle, && (3)\\
&\langle \textbf{Tuple} ::= \texttt{"("}\ \textbf{As} \bullet \texttt{")"}, 0, 1, 4\rangle, && (4)\\
&\langle \textbf{Tuple} ::= \texttt{"("}\ \textbf{As}\ \texttt{")"} \bullet, 0, 4, 5\rangle, && (5)\\
&\langle \textbf{As} ::= \bullet, 1, 1, 1\rangle, && (6)\\
&\langle \textbf{As} ::= \texttt{"a"} \bullet \textbf{More}, 1, 1, 2\rangle, && (7)\\
&\langle \textbf{As} ::= \texttt{"a"}\ \textbf{More} \bullet, 1, 2, 2\rangle, && (8)\\
&\langle \textbf{As} ::= \texttt{"a"}\ \textbf{More} \bullet, 1, 2, 4\rangle, && (9)\\
&\langle \textbf{More} ::= \bullet, 2, 2, 2\rangle, && (10)\\
&\langle \textbf{More} ::= \texttt{","} \bullet \texttt{"a"}\ \textbf{More}, 2, 2, 3\rangle, && (11)\\
&\langle \textbf{More} ::= \texttt{","}\ \texttt{"a"} \bullet \textbf{More}, 2, 3, 4\rangle, && (12)\\
&\langle \textbf{More} ::= \texttt{","}\ \texttt{"a"}\ \textbf{More} \bullet, 2, 4, 4\rangle, && (13)\\
&\langle \textbf{More} ::= \bullet, 4, 4, 4\rangle\} && (14)
\end{aligned}
$$

Figure 2: A BSR set for the grammar $\Gamma_{\textbf{tuple}}$ and t-string `"(a,a)"`.

New symbols appear at each step with superscripts that indicate which substring of `"(a,a)"` they derive. This information is extracted from the BSR elements indicated on the right-hand side. For example, the first step replaces **Tuple** with the sequence `"("` **As** `")"` following BSR element (5). The numbers 1 and 4 are extracted from the combination of the BSR elements (5), (4), and (1) and determine that **As** derives the substring `"a,a"`.

Figure 2 contains BSR elements that have not contributed to the derivation and there is no other derivation of the t-string `"(a,a)"` to which they may have contributed. However, such redundant elements are expected in the output of a parsing procedure as no procedure can foresee, in general, which BSR elements will contribute to a derivation (without prior knowledge).

Discussion on BSR sets. The parsing algorithms presented in this paper add an element of the form $\langle X ::= \alpha t \bullet \beta, l, k, r\rangle$ to Υ, with $r = k + 1$, to record that terminal t is at position k in the input t-string τ. Any post-processor therefore no longer needs τ as input to establish that $t \rightarrow_\Gamma^* \tau^{k,k+1}$. The algorithms add elements of the form $\langle X ::= \bullet, l, l, l\rangle$ to record that $\epsilon \in \Gamma(X)$ so that post-processors do not require grammar Γ as input. These are the only elements of the form $\langle X ::= \bullet \beta, l, k, r\rangle$ added to Υ by the procedures.

The BSR representation Υ is binarised in the sense that the grammar slot g of an element $\langle g, l, k, r\rangle$ in Υ splits an alternate of X in two. The number k is referred to as *pivot* k because k splits the range $[l...r]$ in two. Binarisation is required to keep the size of Υ under control, ensuring the worst-case cubic[4] complexity of the data structure.

[4]The number of grammar slots is determined by the number of symbols in the alternates of the grammar and the indices l, k, and r are in the range $[0...n]$, where n is the length of the input string. The number of possible BSR elements is therefore bound by $O(n^3)$.

Besides soundness, it is also possible to consider whether the set Υ computed by a parsing procedure is complete in the sense that it embeds all derivations of a t-string τ. In [26], a theorem is stated and proven for an algorithm similar to the algorithm FUN-GLL given in this paper. The precise connection between BSR sets and parse forests is explained in [11].

3. Recursive descent

This section describes a straightforward recursive descent parsing procedure by giving first a recognition procedure and subsequently extending it to compute BSR sets. The resulting parsing procedure forms the basis of the description of the FUN-GLL algorithm of Section 4.

3.1. The recognition procedure

The recognition procedure is given a grammar Γ, a t-string τ, and a nonterminal X, and determines whether τ is derived by X in Γ. This is done by testing whether $|\tau|$, the length of τ, is in the result of 'descending' nonterminal X with index 0 (an index into τ). That is, the procedure determines whether it holds that (the function $descend_1$ is defined later):

$$|\tau| \in descend_1(\Gamma, \tau, X, 0)$$

The purpose of descending a nonterminal X with index l is to find a set of indices R such that X derives the substrings $t^{l,r}$ of τ for all $r \in R$, i.e. whether $X \rightarrow_\Gamma^* \tau^{l,r}$. (This is why the condition above checks whether $|\tau|$ is in the result of a call to $descend_1$ with 0 as its fourth argument). The index l is thus referred to as the left extent and the elements of R are referred to as right extents.

In recursive descent parsing, 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. The consideration of lookahead is orthogonal to the logic of the recursive descent parser that employs it. In this paper lookahead is ignored to simplify the presentation of its algorithms. Moreover, in general, lookahead is not sufficient to rule out all or all but one alternate, and complete procedures therefore need to accept some form of nondeterminism. However, lookahead does result in considerable efficiency improvements under certain circumstances. The recognition procedure presented in this section processes all alternates of a nonterminal and collects their results in a set (the set R mentioned above) akin to Wadler's 'list of successes' method [27].

$$descend_1(\Gamma, \tau, X, l) = \{r \mid \beta \in \Gamma(X), r \in process_1(\Gamma, \tau, \beta, l)\}$$

Processing an alternate β of nonterminal X with index k – referred to as a pivot – involves 'matching' all symbols in β, one after the other, in the order they appear (the definition of $process_1$ is given later). Matching a terminal t is testing whether t is the next symbol in the t-string, i.e. the symbol found at a given index. Matching a nonterminal symbol is descending the nonterminal.

$$match_1(\Gamma, t_0 \dots t_n, s, k) = \begin{cases} descend_1(\Gamma, t_0 \dots t_n, s, k) & \textbf{if } s \in N \\ \{k+1\} & \textbf{if } s \in T \textbf{ and } t_k = s \\ \emptyset & \textbf{otherwise} \end{cases}$$

Descending a nonterminal and matching a terminal both result in a set of right extents R. Processing an alternate $\beta = s\beta'$ is done by descending or matching symbol s to give R and making a recursive call to process the remainder β' for each $r \in R$.

$$process_1(\Gamma, \tau, \beta, k) = \begin{cases} \{r \mid k' \in match_1(\Gamma, \tau, s, k), \\ \qquad r \in process_1(\Gamma, \tau, \beta', k')\} & \textbf{if } \beta = s\beta' \\ \{k\} & \textbf{if } \beta = \epsilon \end{cases}$$

Initially, pivot k_0 equals the left extent of descending X. Finally, at the base case $\beta = \epsilon$, pivot k_m is discovered as one of the right extents of descending X. The intermediate values $k_1, \dots, k_{m-1}$ obtained for the pivot by m recursive calls split a t-string derived by X into m substrings, each derived by one of the symbols of the alternate.

The following higher-order[5] function returns a recogniser – a function mapping a t-string to a truth value – for the language generated by a given nonterminal in a given grammar:

$$recogniser\_for(\Gamma, X)(\tau) = \begin{cases} \textbf{true} & \textbf{if } |\tau| \in descend(\Gamma, \tau, X, 0) \\ \textbf{false} & \textbf{otherwise} \end{cases}$$

For example, the function $recogniser\_for(\Gamma_{\textbf{Tuple}}, \textbf{Tuple})$ is a recogniser for the language {`"()"`, `"(a)"`, `"(a,a)"`, ...}.

Left-recursion. A nonterminal X is left-recursive in a grammar Γ if X derives, via one or more steps, a sequence of symbols starting with X, i.e. if $X \rightarrow_\Gamma^+ X\delta$ for some δ. If X is left-recursive in Γ, then evaluating $descend_1(\Gamma, \tau, X, k)$, for any τ and k, will require the evaluation of $process_1(\Gamma, \tau, X\delta', k)$, for some δ', which in turn requires $descend_1(\Gamma, \tau, X, k)$. This cyclic dependency results in nontermination if the procedure is directly implemented as described above.

The next subsection extends the recognition procedure to a parsing procedure by computing BSR sets. Section 4 extends the resulting parsing procedure to a complete procedure that works for all grammars including grammars with left-recursive nonterminals.

[5]Higher-order functions are described using several layers of parenthesised parameters. For example, the 'curried' version of a function f with three parameters is defined as $f(a)(b)(c) = ...$ and is applied as $f(1)(2)(3)$. The uncurried version is defined as $f(a, b, c) = ...$ and is applied as $f(1, 2, 3)$.

3.2. The parsing procedure

The parsing procedure has a function $process_2$, replacing $process_1$, that is given a grammar slot $\langle X, \alpha, \beta\rangle$ rather than just β (as in $process_1$). Moreover, $process_2$ also receives an additional integer l besides pivot k. Intuitively, the additional information reflects that the parsing procedure has descended X with left extent l and that processing the alternate $\alpha\beta$ of X led to processing β. This information is sufficient to construct the necessary BSR elements. The result of matching a symbol is no longer just a set of right extents R, but also a BSR set Υ.

$$process_2(\Gamma, \tau, \langle X, \alpha, \beta\rangle, l, k) = \begin{cases} \langle\{k\}, \{\langle X ::= \bullet, l, l, l\rangle\}\rangle & \textbf{if } \beta = \epsilon, \alpha = \epsilon \\ \langle\{k\}, \emptyset\rangle & \textbf{if } \beta = \epsilon, \alpha \neq \epsilon \\ continue(\Gamma, \tau, \langle X, \alpha s, \beta'\rangle, l, k) & \textbf{if } \beta = s\beta' \end{cases}$$

The base case is split in two by checking whether the chosen alternate of X is the empty sequence. If so, a BSR element is added (as discussed at the end of Section 2). Matching the first symbol s of β, if any, is done by *continue*, defined below, making recursive calls to $process_2$ for all right extents discovered by matching s. Binary union over matching results is defined component-wise, i.e. $\langle R_1, \Upsilon_1\rangle \cup \langle R_2, \Upsilon_2\rangle = \langle R_1 \cup R_2, \Upsilon_1 \cup \Upsilon_2\rangle$, and $\bigcup M$ denotes the finitary union over the matching results in M.

$$\begin{aligned} &continue(\Gamma, \tau, \langle X, \alpha s, \beta\rangle, l, k) = \langle R', \Upsilon \cup \Upsilon' \cup \Upsilon''\rangle \\ &\quad \textbf{with } \langle R, \Upsilon\rangle = match_2(\Gamma, \tau, s, k) \\ &\quad \textbf{and } \langle R', \Upsilon'\rangle = \bigcup\{process_2(\Gamma, \tau, \langle X, \alpha s, \beta\rangle, l, r) \mid r \in R\} \\ &\quad \textbf{and } \Upsilon'' = \{\langle X ::= \alpha s \bullet \beta, l, k, r\rangle \mid r \in R\} \end{aligned}$$

The initial call to $process_2$ has $l = k$. Each recursive call propagates left extent l whilst pivot k potentially increases. At the base case $\beta = \epsilon$, it is discovered that k is one of the right extents of descending X. The intermediate values $k_1 \dots k_{m-1}$ obtained for the pivot by the $m = |\alpha\beta|$ recursive calls are stored in the BSR elements constructed by *continue*. In this way, the BSR set reflects the potentially many ways in which $\alpha\beta$ derives substrings of τ.

As before, matching a symbol s is descending s if it is nonterminal, and testing it against the input t-string if it is terminal.

$$match_2(\Gamma, t_0 \dots t_n, s, k) = \begin{cases} descend_2(\Gamma, t_0 \dots t_n, s, k) & \textbf{if } s \in N \\ \langle\{k+1\}, \emptyset\rangle & \textbf{if } s \in T \textbf{ and } t_k = s \\ \langle\emptyset, \emptyset\rangle & \textbf{otherwise} \end{cases}$$

Descending nonterminal X calls $process_2$ with slot $X ::= \bullet\beta$ for each alternate $\beta \in \Gamma(X)$.

$$descend_2(\Gamma, \tau, X, l) = \bigcup\{process_2(\Gamma, \tau, X ::= \bullet\beta, l, l) \mid \beta \in \Gamma(X)\}$$

The function *parser_for* returns a parser – a function mapping a t-string to a BSR set – for the language generated by a given nonterminal in a given grammar.

$$\begin{aligned} &\mathit{parser\_for}(\Gamma, X)(\tau) = \Upsilon \\ &\quad \textbf{with } \langle R, \Upsilon\rangle = \mathit{descend}_2(\Gamma, \tau, X, 0) \end{aligned}$$

For example, the parser $\mathit{parser\_for}(\Gamma_{\textbf{Tuple}}, \textbf{Tuple})$ gives the BSR set of Figure 2 when applied to `"(a,a)"`.

4. FUN-GLL

This section describes the FUN-GLL parsing procedure by generalising the parsing procedure of the previous section. The procedure is complete; it prevents nontermination due to left-recursion, and 'repeated work' in general, through descriptors (similar to the descriptors of [4, 28] and the states of Earley's algorithm [1]), and computes a BSR set that embeds all possible derivations of the input t-string. The complexity is worst-case $O(n^3)$ in both space and runtime, depending on the implementation of the data structures. A discussion on the implementation of data structures for GLL parsing can be found in [29]. This section starts with a high-level summary of the algorithm before presenting its formal definition. The formal definition is explained by referring back to the summary. The example provided by Figures 3, 4, and Table 1 provide insight into how the different steps of the algorithm guarantee termination and completeness.

A descriptor is a triple of a grammar slot and two indices of the input t-string, corresponding exactly to the arguments of $\mathit{process}_2$ (ignoring Γ and τ) and also serving the same purpose. A descriptor is denoted $\langle X ::= \alpha \bullet \beta, l, k\rangle$. A worklist $\mathcal{R}$ contains the descriptors that require processing; its elements are processed one by one by calling the function *process* that replaces $\mathit{process}_2$. A set $\mathcal{U}$ stores all the descriptors that have been added to the worklist previously and is used to ensure that no descriptor is added to the worklist a second time. Nontermination due to left-recursion is simply avoided by never processing the same descriptor twice. However, the flip side of using descriptors to avoid repeated processing is that extra bookkeeping is required to ensure that all derivation information is recorded and thus to ensure the completeness of the algorithm. The main challenge to understanding the FUN-GLL algorithm is understanding how the algorithm ensures it records all the required derivation information.

Consider the situation in which the descriptor $\langle X ::= \alpha \bullet s\beta, l, k\rangle$ has been processed, with s a nonterminal, having resulted in further descriptors $\langle X ::= \alpha s \bullet \beta, l, r_i\rangle$, for all r_i in some set R (the set of right extents discovered by descending s with index k). If the next processed descriptor is of the form $\langle Y ::= \alpha' \bullet s\beta', l', k\rangle$, then no descriptors are added to $\mathcal{R}$ as all descriptors of the form $\langle s ::= \bullet\delta, k, k\rangle$ have already been added previously. However, since s derives the substrings ranging from k to $r_i - 1$, with $r_i \in R$, the descriptors $\langle Y ::= \alpha' s \bullet \beta', l', r_i\rangle$ need to be added to $\mathcal{R}$ (if not already in $\mathcal{U}$) and the BSR elements $\langle Y ::= \alpha' s \bullet \beta', l', k, r_i\rangle$ need to be added to Υ for completeness. To avoid missing these descriptors and BSR elements, the binary relation $\mathcal{P}$ between pairs of *commencements* and right extents is introduced, where a commencement is a pair of a nonterminal and a left extent (i.e. the arguments of $\mathit{descend}_2$). In the example situation, the set R is embedded in $\mathcal{P}$ as specified by the equation $R = \{r \mid \langle\langle s, k\rangle, r\rangle \in \mathcal{P}\}$. The set $\mathcal{P}$ records for all nonterminals the left and right extents that have been discovered so far, i.e. forall X, l, and r, it holds that $\langle\langle X, l\rangle, r\rangle \in \mathcal{P}$ implies $X \rightarrow^*_\Gamma \tau^{l,r}$.

This is not sufficient; some of the descriptors of the form $\langle s ::= \bullet\delta, k, k\rangle$, 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 substrings 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 $\langle Y ::= \alpha' s \bullet \beta', l', r_j\rangle$ and $\langle X ::= \alpha s \bullet \beta, l, r_j\rangle$ to $\mathcal{R}$ (if not in $\mathcal{U}$) and to add the BSR elements $\langle Y ::= \alpha' s \bullet \beta', l', k, r_j\rangle$ and $\langle X ::= \alpha s \bullet \beta, l, k, r_j\rangle$ to Υ. The binary relation $\mathcal{G}$ between commencements and *continuations* is introduced, where a continuation is a pair of a slot and a left extent. Intuitively, a continuation is an incomplete descriptor 'waiting' for a right extent to take up the role of pivot and thereby completing the descriptor. That is, for all X, g, l, k, and r it holds that if $\langle\langle X, k\rangle, \langle g, l\rangle\rangle \in \mathcal{G}$, then continuation $\langle g, l\rangle$ is combined with r to form descriptor $\langle g, l, r\rangle$ (and BSR element $\langle g, l, k, r\rangle$) whenever k and r are discovered as a left and right extent pair of X.

Summary. The FUN-GLL algorithm is summarised as follows. While there are descriptors in the worklist, arbitrarily select the next descriptor $\langle X ::= \alpha \bullet \beta, l, k\rangle$ to be processed and

- if $\beta = s\beta'$ and
  - if s is terminal, **match** the terminal at position k in the input t-string with s. Only if the match is successful, add $\langle X ::= \alpha s \bullet \beta', l, k+1\rangle$ to the worklist and add $\langle X ::= \alpha s \bullet \beta', l, k, k+1\rangle$ to Υ.
  - if s is nonterminal, find $R = \{r \mid \langle\langle s, k\rangle, r\rangle \in \mathcal{P}\}$ and extend $\mathcal{G}$ with $\langle\langle s, k\rangle, \langle X ::= \alpha s \bullet \beta', l\rangle\rangle$. If R is empty, **descend** s by adding $\langle s ::= \bullet\delta, k, k\rangle$, for all alternates δ of s, to the worklist (if not in $\mathcal{U}$). If R is not empty, **skip** s by adding $\langle X ::= \alpha s \bullet \beta', l, r_i\rangle$, for all $r_i \in R$, to the worklist (if not in $\mathcal{U}$) and by adding $\langle X ::= \alpha s \bullet \beta', l, k, r_i\rangle$ to Υ.
- if $\beta = \epsilon$, extend $\mathcal{P}$ with $\langle\langle X, l\rangle, k\rangle$, and **ascend** X by finding $K = \{\langle g, l'\rangle \mid \langle\langle X, l\rangle, \langle g, l'\rangle\rangle \in \mathcal{G}\}$, adding $\langle g, l', k\rangle$ to the worklist for all $\langle g, l'\rangle \in K$ (if not in $\mathcal{U}$), adding $\langle g, l', l, k\rangle$ to Υ and, if $\alpha = \epsilon$, adding $\langle X ::= \bullet, k, k, k\rangle$ to Υ as well.

#	descriptor	action	new descr.	new BSR	$\mathcal{G}$ extension	$\mathcal{P}$ extension
1	$\langle E ::= \bullet EEE, 0, 0\rangle$	**descend**	~~1,2,3~~		$\langle\langle E, 0\rangle, \langle E ::= E \bullet EE, 0\rangle\rangle$	
2	$\langle E ::= \bullet$ `"1"`$, 0, 0\rangle$	**match**	4	5		
3	$\langle E ::= \bullet, 0, 0\rangle$	**ascend**	5	1,2		$\langle\langle E, 0\rangle, 0\rangle$
4	$\langle E ::=$ `"1"`$\bullet, 0, 1\rangle$	**ascend**	6	6		$\langle\langle E, 0\rangle, 1\rangle$
5	$\langle E ::= E \bullet EE, 0, 0\rangle$	**skip**	7,8	3,7	$\langle\langle E, 0\rangle, \langle E ::= EE \bullet E, 0\rangle\rangle$	
6	$\langle E ::= E \bullet EE, 0, 1\rangle$	**descend**	9,10,11		$\langle\langle E, 1\rangle, \langle E ::= EE \bullet E, 0\rangle\rangle$	
7	$\langle E ::= EE \bullet E, 0, 0\rangle$	**skip**	12,13	4,9	$\langle\langle E, 0\rangle, \langle E ::= EEE \bullet, 0\rangle\rangle$	
8	$\langle E ::= EE \bullet E, 0, 1\rangle$	**descend**	~~9,10,11~~		$\langle\langle E, 1\rangle, \langle E ::= EEE \bullet, 0\rangle\rangle$	
9	$\langle E ::= \bullet EEE, 1, 1\rangle$	**descend**	~~9,10,11~~		$\langle\langle E, 1\rangle, \langle E ::= E \bullet EE, 1\rangle\rangle$	
10	$\langle E ::= \bullet$ `"1"`$, 1, 1\rangle$	~~**match**~~				
11	$\langle E ::= \bullet, 1, 1\rangle$	**ascend**	~~8,13~~,14	8,10,11,12		$\langle\langle E, 1\rangle, 1\rangle$
12	$\langle E ::= EEE \bullet, 0, 0\rangle$	**ascend**	~~5,7,12~~	~~2,3,4~~		~~$\langle\langle E, 0\rangle, 0\rangle$~~
13	$\langle E ::= EEE \bullet, 0, 1\rangle$	**ascend**	~~6,8,13~~	~~6,7,9~~		~~$\langle\langle E, 0\rangle, 1\rangle$~~
14	$\langle E ::= E \bullet EE, 1, 1\rangle$	**skip**	15	13	$\langle\langle E, 1\rangle, \langle E ::= EE \bullet E, 1\rangle\rangle$	
15	$\langle E ::= EE \bullet E, 1, 1\rangle$	**skip**	16	14	$\langle\langle E, 1\rangle, \langle E ::= EEE \bullet, 1\rangle\rangle$	
16	$\langle E ::= EEE \bullet, 1, 1\rangle$	**ascend**	~~8,13,14,15,16~~	~~8,10,12,13,14~~		~~$\langle\langle E, 1\rangle, 1\rangle$~~

Table 1: An example execution of FUN-GLL with grammar $\Gamma_{\mathbf{EEE}}$ and t-string `"1"`. The BSR elements are enumerated in Figure 4.

$$\mathbf{E} \quad ::= \quad \mathbf{E}\ \mathbf{E}\ \mathbf{E} \mid \texttt{"1"} \mid \epsilon$$

Figure 3: The ambiguous and cyclic grammar $\Gamma_{\mathbf{EEE}}$ from [19].

As an example, consider grammar $\Gamma_{\mathbf{EEE}}$ in Figure 3 taken from Ridge [19] and the execution of the FUN-GLL algorithm in Table 1 with grammar $\Gamma_{\mathbf{EEE}}$ and t-string `"1"` as inputs. Each row of the table corresponds to the selection of a descriptor (second column) from the worklist. The third column shows the action that has been performed to process the descriptor as determined by the structure of the descriptor and the contents of $\mathcal{G}$ and $\mathcal{P}$ at that step in the execution. The fourth and fifth column show the descriptors and BSR elements discovered by the action and the sixth and seventh column show the extensions to $\mathcal{G}$ and $\mathcal{P}$ made by the action. Struck out elements in these columns are duplicate and are therefore not added to the respective set. The numbers in the "new descriptors" column refer to the numbers in the first column. For example, at sted 2, the descriptor processed at step 4 is added to the worklist. The numbers in the "new BSR" column refer to the indices assigned to the BSR elements in Figure 4. The descriptors of the first three steps are added to the worklist during initialisation (function *descend* in the definition of the algorithm). At steps 1 and 9 non-termination due to left-recursion is avoided.

4.1. The complete parsing procedure

The FUN-GLL algorithm is formalised by a recursive function *loop* that processes the descriptor in the worklist $\mathcal{R}$ by selecting and removing a descriptor in each recursive call 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 the order might influence efficiency. An analysis of the influence of the order on efficiency can be found in [28]. The function *fungll* makes the initial call to *loop* given a nonterminal X.

$$\{\langle \mathbf{E} ::= \bullet, 0, 0, 0\rangle, \quad (1)$$
$$\langle \mathbf{E} ::= \mathbf{E} \bullet \mathbf{E}\ \mathbf{E}, 0, 0, 0\rangle, \quad (2)$$
$$\langle \mathbf{E} ::= \mathbf{E}\ \mathbf{E} \bullet \mathbf{E}, 0, 0, 0\rangle, \quad (3)$$
$$\langle \mathbf{E} ::= \mathbf{E}\ \mathbf{E}\ \mathbf{E} \bullet, 0, 0, 0\rangle, \quad (4)$$
$$\langle \mathbf{E} ::= \texttt{"1"} \bullet, 0, 0, 1\rangle, \quad (5)$$
$$\langle \mathbf{E} ::= \mathbf{E} \bullet \mathbf{E}\ \mathbf{E}, 0, 0, 1\rangle, \quad (6)$$
$$\langle \mathbf{E} ::= \mathbf{E}\ \mathbf{E} \bullet \mathbf{E}, 0, 0, 1\rangle, \quad (7)$$
$$\langle \mathbf{E} ::= \mathbf{E}\ \mathbf{E} \bullet \mathbf{E}, 0, 1, 1\rangle, \quad (8)$$
$$\langle \mathbf{E} ::= \mathbf{E}\ \mathbf{E}\ \mathbf{E} \bullet, 0, 0, 1\rangle, \quad (9)$$
$$\langle \mathbf{E} ::= \mathbf{E}\ \mathbf{E}\ \mathbf{E} \bullet, 0, 1, 1\rangle, \quad (10)$$
$$\langle \mathbf{E} ::= \bullet, 1, 1, 1\rangle, \quad (11)$$
$$\langle \mathbf{E} ::= \mathbf{E} \bullet \mathbf{E}\ \mathbf{E}, 1, 1, 1\rangle, \quad (12)$$
$$\langle \mathbf{E} ::= \mathbf{E}\ \mathbf{E} \bullet \mathbf{E}, 1, 1, 1\rangle, \quad (13)$$
$$\langle \mathbf{E} ::= \mathbf{E}\ \mathbf{E}\ \mathbf{E} \bullet, 1, 1, 1\rangle\} \quad (14)$$

Figure 4: A BSR set for the grammar $\Gamma_{\mathbf{EEE}}$ and t-string `"1"`.

$$\mathit{fungll}(\Gamma, \tau, X) = \mathit{loop}(\Gamma, \tau, \mathit{descend}(\Gamma, X, 0), \emptyset, \emptyset, \emptyset, \emptyset)$$
$$\mathit{descend}(\Gamma, X, l) = \{\langle X ::= \bullet\beta, l, l\rangle \mid \beta \in \Gamma(X)\}$$

The worklist $\mathcal{R}$ initially contains all descriptors of the form $\langle X ::= \bullet\beta, 0, 0\rangle$, for all alternates β of X, as shown by the application of *descend* to X and 0.

Processing a descriptor involves executing one of the **descend**, **ascend**, **skip**, or **match** actions as previously explained in the summary of the algorithm. These actions are formalised by the functions *descend*, *ascend*, *skip*, and *match*, defined later. With the exception of *descend*, these functions return a pair of sets containing the descriptors

and BSR elements discovered by executing the corresponding action (the **descend** action only returns a set of descriptors). Processing a descriptor may involve extending the relations $\mathcal{G}$ and $\mathcal{P}$. The function *process* is called by *loop* to execute one of the actions and to return any extensions to $\mathcal{G}$ and $\mathcal{P}$.

$$
\begin{array}{l}
loop(\Gamma, \tau, \mathcal{R}, \mathcal{U}, \mathcal{G}, \mathcal{P}, \Upsilon) = \\
\quad \begin{cases}
\langle \mathcal{U}, \Upsilon \rangle & \textbf{if } \mathcal{R} = \emptyset \\
loop(\Gamma, \tau, \mathcal{R}'', \mathcal{U} \cup \{d\}, \mathcal{G} \cup \mathcal{G}', \mathcal{P} \cup \mathcal{P}', \Upsilon \cup \Upsilon') & \textbf{if } d \in \mathcal{R} \\
\quad \textbf{with } \langle \langle \mathcal{R}', \Upsilon' \rangle, \mathcal{G}', \mathcal{P}' \rangle = process(\Gamma, \tau, d, \mathcal{G}, \mathcal{P}) & \\
\quad \textbf{and } \ \mathcal{R}'' = (\mathcal{R} \cup \mathcal{R}') \setminus (\mathcal{U} \cup \{d\}) &
\end{cases}
\end{array}
$$

When processed, the descriptor d is added to $\mathcal{U}$ and because no element in $\mathcal{U}$ is ever added to $\mathcal{R}$, it is guaranteed that a descriptor is only processed once. The loop terminates because the number of possible descriptors that can be added to $\mathcal{R}$ is finite. Non-termination due to left-recursion is thus avoided by ensuring that no descriptor enters the worklist $\mathcal{R}$ a second time. The details of the ways in which information is added to and obtained from $\mathcal{G}$ and $\mathcal{P}$ are crucial to ensuring all the BSR elements required for completeness are added to Υ. These details are presented in the definition of *process* that follows.

For clarity, the definition of *process* is split into $process_\epsilon$ and $process_{symbol}$, handling the cases $\beta = \epsilon$ and $\beta = s\beta'$ whenever a descriptor $\langle X ::= \alpha \bullet \beta, l, k \rangle$ is processed.

$$
\begin{array}{l}
process(\Gamma, \tau, \langle X ::= \alpha \bullet \beta, l, k \rangle, \mathcal{G}, \mathcal{P}) = \\
\quad \begin{cases}
process_\epsilon(\langle X ::= \alpha \bullet, l, k \rangle, \mathcal{G}, \mathcal{P}) & \textbf{if } \beta = \epsilon \\
process_{symbol}(\Gamma, \tau, \langle X ::= \alpha \bullet s\beta', l, k \rangle, \mathcal{G}, \mathcal{P}) & \textbf{if } \beta = s\beta'
\end{cases}
\end{array}
$$

In the case $\beta = \epsilon$ (**ascend**), it is discovered that k is a right extent, i.e. that $X \rightarrow_\Gamma^* \tau^{l,k}$. This is 'remembered' by returning the commencement and right extent pair $\langle \langle X, l \rangle, k \rangle$ to extend $\mathcal{P}$. All continuations $K = \{\langle g, l' \rangle \mid \langle \langle X, l \rangle, \langle g, l' \rangle \rangle \in \mathcal{G}\}$ are 'waiting' for the discovery of additional right extents such as k. Ascending X involves combining the continuations in K with l and k to form descriptors and BSR elements, as shown by the definition of *ascend* given later.

$$
\begin{array}{l}
process_\epsilon(\langle X ::= \alpha \bullet, l, k \rangle, \mathcal{G}, \mathcal{P}) = \langle \langle \mathcal{R}, \Upsilon \cup \Upsilon' \rangle, \emptyset, \{\langle \langle X, l \rangle, k \rangle\} \rangle \\
\quad \textbf{with } \langle \mathcal{R}, \Upsilon \rangle = ascend(l, K, k) \\
\quad \textbf{and } \ K = \{\langle g, l' \rangle \mid \langle \langle X, l \rangle, \langle g, l' \rangle \rangle \in \mathcal{G}\} \\
\quad \textbf{and } \ \Upsilon' = \{\langle X ::= \bullet, l, l, l \rangle \mid \alpha = \epsilon\}
\end{array}
$$

If $\beta = \alpha = \epsilon$, an additional BSR element $\langle X ::= \bullet, l, l, l \rangle$ is returned (note that $\alpha = \epsilon$ implies that $l = k$).

In the case $\beta = s\beta'$ with s a nonterminal symbol, the commencement and continuation pair $\langle \langle s, k \rangle, \langle X ::= \alpha s \bullet \beta, l \rangle \rangle$ is returned to extend $\mathcal{G}$. The set $R = \{r \mid \langle \langle s, k \rangle, r \rangle \in \mathcal{P}\}$ contains all the right extents r such that $s \rightarrow_\Gamma^* \tau^{l,r}$. If R is empty (**descend**), then s is descended with left extent k (potentially for a second time). If R is not empty (**skip**), function *skip* combines the continuation $\langle X ::= \alpha s \bullet \beta', l \rangle$ with k and r (for all $r \in R$) to form descriptors and BSR elements.

$$
\begin{array}{l}
process_{symbol}(\Gamma, \tau, \langle X ::= \alpha \bullet s\beta', l, k \rangle, \mathcal{G}, \mathcal{P}) = \\
\quad \begin{cases}
\langle match(\tau, \langle X ::= \alpha \bullet s\beta', l, k \rangle), \emptyset, \emptyset \rangle & \textbf{if } s \in T \\
\langle \langle descend(\Gamma, s, k), \emptyset \rangle, \mathcal{G}', \emptyset \rangle & \textbf{if } s \in N \textbf{ and } R = \emptyset \\
\langle skip(k, \langle X ::= \alpha s \bullet \beta', l \rangle, R), \mathcal{G}', \emptyset \rangle & \textbf{if } s \in N \textbf{ and } R \neq \emptyset
\end{cases} \\
\quad \textbf{with } R = \{r \mid \langle \langle s, k \rangle, r \rangle \in \mathcal{P}\} \\
\quad \textbf{and } \ \mathcal{G}' = \{\langle \langle s, k \rangle, \langle X ::= \alpha s \bullet \beta', l \rangle \rangle\}
\end{array}
$$

In the case $\beta = s\beta'$ with s a terminal symbol (**match**), a descriptor $\langle X ::= \alpha s \bullet \beta', l, k+1 \rangle$ and BSR element $\langle X ::= \alpha s \bullet \beta', l, k, k+1 \rangle$ are returned if s is the terminal at position k in τ.

$$
\begin{array}{l}
match(t_0 \ldots t_n, \langle X ::= \alpha \bullet s\beta', l, k \rangle) = \\
\quad \begin{cases}
\langle \{\langle X ::= \alpha s \bullet \beta', l, k+1 \rangle\}, & \\
\quad \{\langle X ::= \alpha s \bullet \beta', l, k, k+1 \rangle\} \rangle & \textbf{if } t_k = s \\
\langle \emptyset, \emptyset \rangle & \textbf{otherwise}
\end{cases}
\end{array}
$$

Finally, functions *skip* and *ascend* are defined. Function *skip* combines a single continuation c with possibly many right extents R, whereas *ascend* combines a single right extent r with possibly many continuations K, to form descriptors and BSR elements with pivot k.

$$
\begin{array}{rl}
skip(k, c, R) & = nmatch(k, \{c\}, R) \\
ascend(k, K, r) & = nmatch(k, K, \{r\}) \\
nmatch(k, K, R) & = \langle \mathcal{R}, \Upsilon \rangle \\
& \quad \textbf{with } \mathcal{R} = \{\langle g, l, r \rangle \mid \langle g, l \rangle \in K, r \in R\} \\
& \quad \textbf{and } \ \Upsilon = \{\langle g, l, k, r \rangle \mid \langle g, l \rangle \in K, r \in R\}
\end{array}
$$

The function *complete_parser_for* returns a complete parser – a function mapping a t-string to a BSR set that embeds all derivations of the t-string – for the language generated by a given nonterminal in a given grammar.

$$
\begin{array}{l}
complete\_parser\_for(\Gamma, X)(\tau) = \Upsilon \\
\quad \textbf{with } \langle \mathcal{U}, \Upsilon \rangle = fungll(\Gamma, \tau, X)
\end{array}
$$

The generality of FUN-GLL is demonstrated by applying it to the example grammar from Ridge [19] shown in Figure 3. The grammar is highly ambiguous and cyclic ($\mathbf{E}$ can derive itself in at least one step by applying the rule $\mathbf{E} \rightarrow \mathbf{E}\ \mathbf{E}\ \mathbf{E}$ and then applying $\mathbf{E} \rightarrow \epsilon$ twice). Given the t-string `"1"`, the parser $complete\_parser\_for(\Gamma_{\mathbf{EEE}}, \mathbf{E})$ returns the BSR set of Figure 4. An example execution of FUN-GLL with these inputs is given in Table 1. The resulting BSR set embeds the following derivation:

$$
\begin{array}{rr}
\mathbf{E}^{0,1} \rightarrow \mathbf{E}^{0,0}\ \mathbf{E}^{0,1}\ \mathbf{E}^{1,1} & \{(10), (7), (2)\} \\
\rightarrow \mathbf{E}^{0,1}\ \mathbf{E}^{1,1} & \{(1)\} \\
\rightarrow \mathbf{E}^{0,0}\ \mathbf{E}^{0,0}\ \mathbf{E}^{0,1}\ \mathbf{E}^{1,1} & \{(9), (3), (2)\} \\
\rightarrow \mathbf{E}^{0,0}\ \mathbf{E}^{0,1}\ \mathbf{E}^{1,1} & \{(1)\} \\
\rightarrow \mathbf{E}^{0,1}\ \mathbf{E}^{1,1} & \{(1)\} \\
\rightarrow \texttt{"1"}^{0,1}\ \mathbf{E}^{1,1} & \{(5)\} \\
\rightarrow \texttt{"1"} & \{(11)\}
\end{array}
$$

Note that the same sequence of symbols is obtained after the second and the fifth step. The third, fourth, and fifth step can be repeated arbitrarily many times to produce infinitely many derivations. This example shows how infinitely many derivations can be embedded in a BSR set for a cyclic grammar. BSR post-processors can be used to realise ambiguity reduction, filtering out the derivations that are not preferred. For example, Ridge presents a post-processor in which spurious repeated steps are avoided, resulting in 'minimal' derivations. Instead of the above, the following derivation would be selected (among others):

$$\begin{aligned}
\mathbf{E}^{0,1} &\rightarrow \mathbf{E}^{0,0}\ \mathbf{E}^{0,1}\ \mathbf{E}^{1,1} && \{(10),\ (7),\ (2)\}\\
&\rightarrow \mathbf{E}^{0,1}\ \mathbf{E}^{1,1} && \{(1)\}\\
&\rightarrow \texttt{"1"}^{0,1}\ \mathbf{E}^{1,1} && \{(5)\}\\
&\rightarrow \texttt{"1"} && \{(11)\}
\end{aligned}$$

The BNF combinator library discussed in Section 7 implements Ridge's method to avoid enumerating infinitely many derivations and enables its users to choose several additional ambiguity reduction strategies. A principled discussion on the development of ambiguity reduction strategies for filtering BSR sets is warranted but is out of the scope of this paper. Principled approaches to disambiguation have been developed in the Ph.D. theses of Visser, Vinju and Afroozeh and Izmaylova [30, 31, 32].

5. Combinator parsing

The goal of the second half of this paper is to define generalised parser combinators based on FUN-GLL and to demonstrate some of the advantages of using these combinators. This section introduces combinator parsing. The generalised parser combinators based on FUN-GLL are defined in Section 6 and are evaluated in Section 8.

5.1. Conventional parser combinators

Parser combinators are a popular alternative to parser generators as a method for obtaining parsers, especially in functional programming communities. Rather than obtaining a parser from a grammar directly, parsers are composed to form more complex parsers, starting with simple elementary parsers. The composition of parsers is expressed through parser combinators. 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 executes the parsers out of which it is composed typically in a recursive descent, top-down fashion. Parser combinators have been defined in many ways [27, 33, 15, 34, 14].

Figure 5 defines a collection of elementary recognisers and basic recogniser combinators. The recognisers are functions that, given a t-string and an index of the t-string, return a set of indices of the t-string. The indices are similar to the left and right extents of the recognition procedure of Section 3 in the sense that a recogniser

$$\begin{aligned}
term_0(t)(t_0 \ldots t_m, k) &= \begin{cases} \{k+1\} & \textbf{if } t_k = t \\ \emptyset & \textbf{otherwise} \end{cases}\\
seq(p,q)(\tau, l) &= \{r \mid k \in p(\tau, l), r \in q(\tau, k)\}\\
alt(p,q)(\tau, k) &= p(\tau, k) \cup q(\tau, k)\\
succeeds(\tau, k) &= \{k\}\\
fails(\tau, k) &= \emptyset\\
recognise_0(p)(\tau) &= \begin{cases} \textbf{true} & \textbf{if } |\tau| \in p(\tau, 0) \\ \textbf{false} & \textbf{otherwise} \end{cases}
\end{aligned}$$

Figure 5: A collection of basic combinators and elementary parsers.

that is given a t-string τ and an index l returns the indices $r \in R$ if it recognises the substring ranging from l to $r-1$. Infix operators are often associated with combinators. For example, associating $\otimes$ with *seq* and $\oplus$ with *alt* (and granting $\otimes$ a higher precedence than $\oplus$), a recogniser for the grammar $\Gamma_{\mathbf{Tuple}}$ is obtained by applying $recognise_0$ to *pTuple*, defined as follows:

$$\begin{aligned}
pTuple &= term(\texttt{"("}) \otimes pAs \otimes term(\texttt{")"})\\
pAs &= succeeds \oplus term(\texttt{"a"}) \otimes pMore\\
pMore &= succeeds \oplus term(\texttt{","}) \otimes term(\texttt{"a"}) \otimes pMore
\end{aligned}$$

The combinators defined in Figure 5 are for recognition only. Some combinators for parsing produce derivation trees [34]. Most combinator libraries, however, integrate 'semantic actions' that perform a form of evaluation on the fly in the spirit of syntax-directed translation [14, 33, 19]. The combinators presented in Section 6 produce BSR sets. To enable semantic actions, the combinators in Section 6 can be paired with the semantic combinators of [12], as outlined in Section 7.2.

Advantages of parser combinators. 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 actions. Monadic parser combinators even extend the capabilities beyond parsing context-free languages [33]. Secondly, parsers and parser combinators are reusable. Borrowing the abstraction mechanism of the host language, it is possible to abstract over sub-parsers and replace them with a parameter. For example, *pTupleAbs*, defined below, is applied to a recogniser p to obtain a recogniser for the language of tuples whose elements are recognised by p.

$$\begin{aligned}
pTupleAbs(p) &= term(\texttt{"("}) \otimes pAsAbs(p) \otimes term(\texttt{")"})\\
pAsAbs(p) &= succeeds \oplus p \otimes pMoreAbs(p)\\
pMoreAbs &= succeeds \oplus term(\texttt{","}) \otimes p \otimes pMoreAbs(p)
\end{aligned}$$

A third advantage is that additional parser combinators and elementary parsers are defined easily, if the underlying

parser algorithm is sufficiently simple. For example, the function *pred* generalises *term* and returns a recogniser for the terminals that satisfy the given predicate f:

$$pred(f)(t_0 \dots t_m, k) = \begin{cases} \{k+1\} & \textbf{if } f(t_k) \\ \emptyset & \textbf{otherwise} \end{cases}$$

Overcoming the drawbacks of basic parser combinators. There is a direct correspondence between the definition of *pTuple* above and the rules of the BNF description of $\Gamma_{\textbf{Tuple}}$ in Figure 1. This observation suggests that it is possible to write combinator expressions for arbitrary BNF descriptions of grammars. Although it is possible to write the combinator expressions, the resulting recognisers fail to terminate if the described grammar is left-recursive. Moreover, the recognisers exhibit exponential running times for certain grammars which is caused by considering both alternatives in applications of *alt*. In general, parsers constructed with parser combinators based on standard recursive descent parsing inherit the drawbacks of recursive descent parsing.

A common solution to the problem of left-recursion is to refactor the grammar to remove left-recursion, either manually or automatically [16]. Johnson showed that recognition combinators written in continuation-passing style can be extended with memoisation to solve the left-recursion problem [15]. Afroozeh, Izmaylova and Van der Storm, build on Johnson's recognition combinators to develop general parser combinators [35]. Frost, Hafiz, and Callaghan [34] handle left-recursion with a 'curtailment' procedure, making at most as many recursive calls as there are characters remaining in the input t-string.

To avoid parsing inefficiencies in some cases, less naïve variations of *alt* can be defined, for example to avoid considering both alternatives by default [33] or to choose an alternative using lookahead [14]. Efficiency can be improved with memoisation as well [13, 36].

The resulting algorithms are more complicated, and may depend on impure methods to detect recursion [37, 18, 38]. As a result, it can be more difficult to extend the combinator libraries with functions such as *pred*. In general, it is difficult to reason formally about parsers developed with parser combinators. In particular, determining the language recognised by a combinator parser involves knowledge of the underlying operational details.

5.2. Grammar combinators

The approaches suggested by Devriese and Piessens [17], Ljunglöf [18], and Ridge [19] have in common that higher-order functions combine grammar fragments. The resulting grammars are passed to a stand-alone parsing procedure which need not be restricted to recursive descent and can indeed be a generalised parsing procedure. A library of such *grammar combinators*[6] can be seen as an embedded implementation of BNF with parameterisable nonterminals – similar to PRECC or HAPPY grammars [39, 40] and macro-grammers [41, 42] – by taking advantage of the abstraction mechanism of the host language. Like parser combinators, grammar combinators inherit other features of the host language that can be leveraged. However, the user is limited in using these features by the constraint that combinator expressions must yield context-free grammars. 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 and automatic left-factoring [28] can be realised without interfering with the combinator definitions as the underlying parsing procedure is simply replaced.

The next section defines generalised parser combinators based on the FUN-GLL algorithm. Section 7 explains how these combinators are used in practice in a combinator library that has the advantages of a grammar combinator library. Section 8 demonstrates that this library is not as restrictive as a grammar combinator library however.

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

6. FUN-GLL parser combinators

Parsers written with conventional parser combinators represent grammar descriptions only superficially. Although a combinator parser may have been written with a grammar in mind, this grammar is not available to the underlying parsing algorithm. Grammar slots are required, however, to implement parser combinators according to a generalised parsing algorithm such as GLR [2], Earley [1], or GLL [4]. This section demonstrates that it is possible to define generalised parsing combinators, based on FUN-GLL of Section 4, when grammar slots are available.

As mentioned in the previous section, combinator libraries that compute grammars exist. Computing a grammar requires the introduction of nonterminals that are otherwise not present. For example, nonterminals do not occur in the expressions formed by applying the combinators of Figure 5. The nonterminals are typically introduced at recursion points and advanced programming techniques are required that enable the underlying algorithm to observe recursion [18, 16, 17].

Rather than using such advanced techniques, this section adopts the approach taken by Ridge [19] and forces programmers to insert nonterminals. The combinator *nterm* is introduced for this purpose. The disadvantages of forcing the programmer to insert nonterminals are discussed in Section 9. The advantages of generalised parser combinators over grammar combinators are demonstrated in Section 8.

The core BNF combinators. This section makes two significant changes to the combinators of Figure 5 to enable the underlying algorithm to compute grammar slots. Firstly, the combinator *nterm* is introduced to inject nonterminals into combinator expressions. Secondly, the combinators are restricted in their applicability so that combinator expressions have a structure that resembles a BNF grammar

$$
\begin{aligned}
\mathit{nterm} &: N \times \mathit{Choice} \to \mathit{Symb} \\
\mathit{term} &: T \to \mathit{Symb} \\
\mathit{seqOp} &: \mathit{Sequence} \times \mathit{Symb} \to \mathit{Sequence} \\
\mathit{seqStart} &: \mathit{Sequence} \\
\mathit{altOp} &: \mathit{Choice} \times \mathit{Sequence} \to \mathit{Choice} \\
\mathit{alt} &: \mathit{Choice}
\end{aligned}
$$

Figure 6: The signatures of the BNF combinators.

description more directly. The richer structure of combinator expressions makes it possible to extract grammars (and thus grammar slots) from combinator expressions without the need to binarise the grammar [12]. A third change is superficial, renaming *seq* to *seqOp*, *succeeds* to *seqStart*, *alt* to *altOp*, and *fails* to *altStart*, so that their names are suggestive of their roles in a BNF description.

The combinators are applied to form one of three different types of combinator expression: symbol expressions, sequence expressions and choice expressions. Symbol expressions represent symbols in BNF, a sequence expression represents a sequences of symbols (an alternate, or part of an alternate), whereas a choice expression represents the choice between several alternates. Figure 6 gives signatures to the combinators, showing for each combinator which type of combinator expression is formed when it is applied and restricting its operands to specific types of combinator expressions or symbols. As before, the distinct sets T and N contain terminal symbols and nonterminal symbols respectively. The combinators are referred to as 'BNF combinators' because their expressions represent BNF grammar descriptions explicitly.

A sequence expression constructed by *seqStart* represents the empty sequence of symbols. Each application of *seqOp* adds an additional symbol (second argument) to the end of a sequence (first argument). The combinator *seqOp* therefore relates to juxtaposition in a BNF rule. Similarly, a choice expression constructed by *altStart* represents the empty sequence of alternates. Each application of *altOp* adds an additional alternate (second argument) to an other sequence of alternates (first argument). The combinator *altOp* therefore relates to the | operator in a BNF rule. An application of *nterm* groups zero or more alternates (second argument) under a single nonterminal (first argument). The combinator *nterm* hence relates to the ::= operator of a BNF rule. An application of *term* lifts a terminal symbol to a symbol expressions. The combinator *term* relates to terminals as they appear in the right-hand sides of BNF rules. Figure 7 shows the symbol expressions that represent the nonterminals of $\Gamma_{\mathbf{Tuple}}$ defined in Figure 1. Figure 8 shows the symbol expression that represents the **More** nonterminal as a tree

The expressions of Figure 7 are verbose and difficult to read. As shown in [12] (and by the examples of Section 7), the user-experience of conventional combinator libraries is recovered by introducing infix operators and automatic conversions between the different types of combinator expressions. This material is not repeated in this paper. Instead, Section 7 discusses an alternative implementation of the combinator library of [12]. The alternative implementation made possible by the generalised parser combinators introduced in this section.

The remainder of this section gives three alternative definitions of the BNF combinators: first as recognition combinators, then as parser combinators, and finally as generalised parser combinators based on the FUN-GLL algorithm. The definitions of the combinators in this section show a similar progression to the definitions of the recognition and parsing procedures in Sections 3 and 4.

6.1. The recognition combinators

GLL algorithms generalise recursive descent parsing by managing continuations themselves, rather than relying on the call-stack of a host language [4, 28]. In FUN-GLL, the relation $\mathcal{G}$ is used for this purpose. The parsing algorithm implemented in the generalised parser combinators of this section manages continuations in the same way as FUN-GLL. This is made possible by giving the definitions in 'continuation-passing style'. Examples of combinators definitions in continuation-passing style can be found in [15, 18, 11]. The reader is referred to [43] for a general overview on designing and implementing combinator libraries. The recognition combinators that are defined next demonstrate how continuation-passing style is used in this section.

Figure 9 defines the BNF combinators as recognition combinators. The definitions differ from those of Figure 5 in two important ways due to the application of continuation-passing style. Firstly, the recognition functions receive a continuation function (for which the placeholder c is used) as an additional argument. A continuation function takes an index of the input t-string and returns a Boolean. Secondly, the result of evaluating a combinator expression is a Boolean rather than a set of indices. When recognition fails, the result is **false** instead of the empty set. When recognition succeeds, the result of applying the continuation function to $k+1$ is returned rather than $\{k+1\}$. Note that the same continuation can be applied several times when multiple alternates are successful. This follows from the definition of *altOp*, in which c is given to both operands p and q.

The recognition combinators do not take advantage of the additional information provided by nonterminals injected with *nterm* nor by the richer structure of combinator expressions. The parser combinators that are defined next demonstrate how this additional information can be used to compute the grammar slots of BSRs.

6.2. The parser combinators

This paper has so far considered a BSR set to be computed with respect to a grammar that is known before-

$$bnfTuple = nterm(\textbf{Tuple}, altOp(altStart, seqOp(seqOp(seqOp(seqStart, term(\texttt{"("})), bnfAs), term(\texttt{")"}))))$$
$$bnfAs = nterm(\textbf{As}, altOp(altOp(altStart, seqStart), seqOp(seqOp(seqStart, term(\texttt{"a"})), bnfMore)))$$
$$bnfMore = nterm(\textbf{More}, altOp(altOp(altStart, seqStart), seqOp(seqOp(seqOp(seqStart, term(\texttt{","})), term(\texttt{"a"})), bnfMore)))$$

Figure 7: Symbol expressions for $\Gamma_{\textbf{Tuple}}$ without infix operators and automatic conversions.

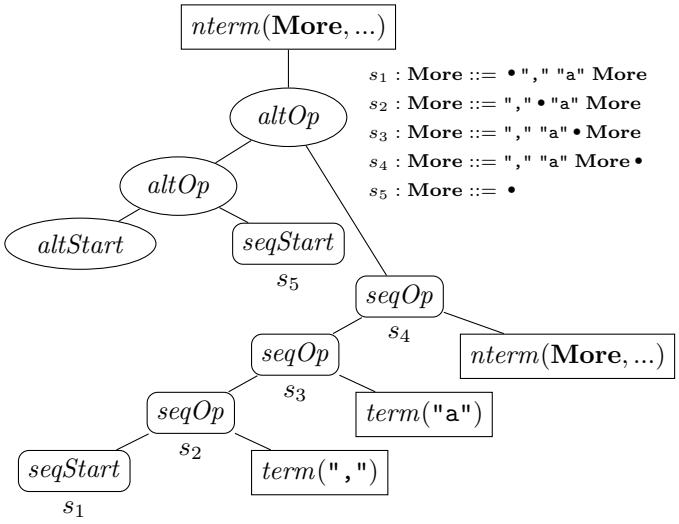

Figure 8: The symbol expression that represents the nonterminal **More** as defined in Figure 1. The sequence expressions within the symbol expression are labelled with the grammar slots $s_1, \ldots, s_5$.

$$nterm_1(X, p)(\tau, k, c) = p(\tau, k, c)$$
$$term_1(t)(t_0 \ldots t_n, k, c) = \begin{cases} c(k+1) & \textbf{if } t_k = t \\ \textbf{false} & \textbf{otherwise} \end{cases}$$
$$seqStart_1(\tau, k, c) = c(k)$$
$$seqOp_1(p, q)(\tau, k, c) = p(\tau, k, c') \quad \textbf{with } c'(r) = q(\tau, r, c)$$
$$altStart_1(\tau, k, c) = \textbf{false}$$
$$altOp_1(p, q)(\tau, k, c) = p(\tau, k, c) \vee q(\tau, k, c)$$
$$recognise_1(p)(\tau) = p(\tau, 0, c)$$
$$\textbf{with } c(r) = \begin{cases} \textbf{true} & \textbf{if } r = |\tau| \\ \textbf{false} & \textbf{otherwise} \end{cases}$$

Figure 9: The BNF combinators defined as recognition combinators.

hand. However, a BSR set can be computed from a BNF combinator expression so that it is valid with respect to the grammar represented by the combinator expression, without actually constructing the grammar. To do so, a grammar slot is computed for each sequence expression. Consider the symbol expression visualised in Figure 8. The sequence expressions within the expression are labelled with the grammar slots that need to be computed for them.

To compute grammar slots for all occurrences of sequence expression, some information is propagated 'downwards', while other information is sent 'upwards'. Sending information upwards means giving the information as input to a continuation function, whereas sending information downwards means passing it as input to a subexpression. A sequence expression sends upwards the sequence of symbols it represents. This is the sequence `"a,"` in the case of the sequence expression s_3 of the example. A sequence expression receives the nonterminal symbol represented by the closest ancestor that is a symbol expression. This is the nonterminal **More** for all sequence expressions in the example. A sequence expression also receives the symbols represented by the 'rest' of the sequence in which it appears. This is the sequence `"a"` **More** in the case of s_2. The definitions of *seqStart* and *seqOp* given later show in detail how slots are computed.

In Section 3, a left extent and grammar slot were added to the input list of the function $process_1$ to extend the recognition procedure to a parsing procedure. In this section, the same modification is made to the input list of continuation functions. Continuation functions are also modified to return BSR sets. The function $continue_1$ is given a BSR $\langle X ::= \alpha \bullet \beta, l, k, r\rangle$ and a continuation function c. It applies c to the descriptor[7] $\langle X ::= \alpha \bullet \beta, l, r\rangle$ and adds the BSR to the result of applying c.

$$continue_1(\langle X ::= \alpha \bullet \beta, l, k, r\rangle, c) =$$
$$c(\langle X ::= \alpha \bullet \beta, l, r\rangle) \cup \Upsilon$$
$$\textbf{with } \Upsilon = \{\langle X ::= \alpha \bullet \beta, l, k, r\rangle \mid \alpha \neq \epsilon \vee \beta = \epsilon\}$$

The definitions of *seqStart* and *seqOp* are as follows:

$$seqStart_1(\tau, X, \beta, l, c_0) = continue_1(\langle X ::= \bullet \beta, l, l, l\rangle, c_0)$$
$$seqOp_1(p, \langle s, q\rangle)(\tau, X, \beta, l, c_0) = p(\tau, X, s\beta, l, c_1)$$
$$\textbf{with } c_1(\langle X ::= \alpha \bullet s\beta, l, k\rangle) = q(\tau, X ::= \alpha s \bullet \beta, l, k, c_2)$$
$$\textbf{with } c_2(\langle X ::= \alpha s \bullet \beta, l, r\rangle) =$$
$$continue_1(\langle X ::= \alpha s \bullet \beta, l, k, r\rangle, c_0)$$

Besides the t-string τ, a sequence expression receives the left extent l, the continuation function c_0, and nonterminal X, and symbol sequence β for constructing a slot. An occurrence of *seqStart* computes the slot $X ::= \bullet \beta$ and calls $continue_1$ with the BSR $\langle X ::= \bullet \beta, l, l, l\rangle$ and continuation function c (which will add the BSR to the BSR set

[7] Descriptors are used as a convenience here. The generalised parser combinators given later in this section actually use descriptors for the same purpose as for which they were introduced in Section 4.

only if $\beta = \epsilon$). The second operand of an occurrence of *seqOp* represents a symbol s that is added to the sequence β when given as input to the first operand p. The continuation of p may be applied to the slot computed for p, providing the α such that $X ::= \alpha s \bullet \beta$ is the slot computed for this occurrence of *seqOp*. The application of the second operand q shows that a symbol expression receives the slot computed for its parent. The continuation of q is applied when a right extent r is found for the symbol s, which is used together with the computed slot, left extent l, and pivot k, to form the BSR $\langle X ::= \alpha s \bullet \beta, l, k, r\rangle$. The BSR is used in a call to $continue_1$.

As mentioned above, a symbol expression representing some s receives the slot $X ::= \alpha s \bullet \beta$ computed for its parent. An occurrence of $nterm_1$ defers evaluation to its second operand p, a choice expression, providing s as input (and pivot k takes up the role of left extent). The input s is required for computing slots for the sequence expressions that form the choice expression p.

$$nterm_1(s,p) = \langle s, nterm\_parser_1(p)\rangle$$
$$nterm\_parser_1(p)(\tau, X ::= \alpha s \bullet \beta, l, k, c) = p(\tau, s, k, c')$$
$$\textbf{with } c'(\langle s ::= \delta \bullet, k, r\rangle) = c(\langle X ::= \alpha s \bullet \beta, l, r\rangle)$$

The continuation c' of p receives a right extent r, discovered for s by p, and applies the continuation c to the descriptor $\langle X ::= \alpha s \bullet \beta, l, r\rangle$, indicating that αs derives $\tau^{l,r}$.

An occurrence of *term* matches the terminal symbol s it represents against the terminal at position k in the input t-string and, if the match was successful, applies its continuation function to the descriptor $\langle X ::= \alpha s \bullet \beta, l, k+1\rangle$.

$$term_1(s) = \langle s, term\_parser_1\rangle$$
$$term\_parser_1(t_0 \ldots t_n, X ::= \alpha s \bullet \beta, l, k, c) = \begin{cases} c(\langle X ::= \alpha s \bullet \beta, l, k+1\rangle) & \textbf{if } t_k = s \\ \emptyset & \textbf{otherwise} \end{cases}$$

If the match fails, the continuation function is not applied (and no BSRs are added). Evaluating an occurrence of *altStart* has the same effect as failing to **match** a terminal (recall that *altStart* replaced *fails* of Figure 5):

$$altStart_1(\tau, s, k, c) = \emptyset$$
$$altOp_1(p, q)(\tau, s, k, c) = p(\tau, s, k, c) \cup q(\tau, s, \epsilon, k, c)$$

An occurrence of *altOp* is evaluated by applying both its operands and uniting the BSR sets produced by each application. The second operand q is a sequence expression and receives nonterminal s and the empty sequence of symbols ϵ. The given sequence is empty because q represents the end of an alternate (consider s_4 and s_5 in Figure 8).

Function $parse_1$ generates a parser for a symbol expression p based on a nonterminal X that should not appear as the first operand of any occurrence of $nterm_1$ in p:

$$parse_1(\langle s, f\rangle)(\tau) = f(\tau, X ::= s \bullet, 0, 0, \hat{c})$$
$$\textbf{with } \hat{c}(\langle X ::= s \bullet, 0, r\rangle) = \emptyset$$
$$\textbf{and} \quad X \text{ a fresh nonterminal}$$

If at any point during a parse the continuation $\hat{c}$ is given a right extent r such that $r = |\tau|$, then the t-string τ is recognised by the parser and the BSR set computed by the application of f embeds its derivations.

The parsers generated by $parse_1$ exhibit the same drawbacks as those generated by *parser_for* in Section 3. The generalised parser combinators that are defined next overcome these drawbacks in the same manner as the FUN-GLL algorithm, using the set $\mathcal{U}$ of descriptors to avoid repeated work and using the relations $\mathcal{G}$ and $\mathcal{P}$ to ensure that all possible derivations are discovered.

6.3. The generalised parser combinators

The definitions of the generalised parser combinators are explained as a modification of $nterm_1$, $term_1$, $seqStart_1$, $seqOp_1$, $altStart_1$, and $altOp_1$ given earlier. The main difference is that a continuation function no longer returns just a BSR set. Instead, a continuation function returns a function that given a tuple of the form $\langle \mathcal{U}, \mathcal{G}, \mathcal{P}, \mathcal{C}, \Upsilon\rangle$ returns a tuple of the form $\langle \mathcal{U}', \mathcal{G}', \mathcal{P}', \mathcal{C}', \Upsilon'\rangle$ for which it holds that $\mathcal{U} \subseteq \mathcal{U}'$, $\mathcal{G} \subseteq \mathcal{G}'$, $\mathcal{P} \subseteq \mathcal{P}'$, $\mathcal{C} \subseteq \mathcal{C}'$, and $\Upsilon \subseteq \Upsilon'$. A function of this type is referred to as a *command*. A continuation function is a function that returns a command given a descriptor. Continuation functions return commands to perform the bookkeeping tasks that are essential to the FUN-GLL algorithm: inserting the required descriptors, BSR elements, and elements of $\mathcal{G}$ and $\mathcal{P}$. The descriptor argument informs the continuation that its behaviour is that of processing the given descriptor and is thus a way to determine whether the continuation has been executed already. Non-termination due to left-recusion is avoided by ensuring no continuation is executed with the same descriptor a second time. As shown by the definition of $continue_2$ given below, a continuation only exhibits its effects if its descriptor argument is not already in $\mathcal{U}$.

The sets $\mathcal{U}$, $\mathcal{G}$, $\mathcal{P}$, and Υ serve the same purpose as in Section 4. Unlike in FUN-GLL, there is no worklist $\mathcal{R}$. The additional relation $\mathcal{C}$ maps continuations to continuation functions. An invariant that is to be maintained by commands is that if a commencement $\langle X, l\rangle$ is mapped to continuation $\langle g, l'\rangle$ in $\mathcal{G}$, i.e. $\langle\langle X, l\rangle, \langle g, l'\rangle\rangle \in \mathcal{G}$, then there exists exactly one continuation function c such that $\langle\langle g, l'\rangle, c\rangle \in \mathcal{C}$. Intuitively, if $\langle\langle g, l'\rangle, c\rangle \in \mathcal{C}$, then whenever continuation $\langle g, l'\rangle$ is combined with a right extent r to form a descriptor $\langle g, l', r\rangle$, c is the code for processing that descriptor, i.e. c is applied to $\langle g, l', r\rangle$ where *process* would be applied to $\langle g, l', r\rangle$ in the FUN-GLL algorithm. The relation $\mathcal{C}$ is introduced so that usages of $\mathcal{G}$ are identical to those of Section 4. Implementations can 'merge' $\mathcal{G}$ and $\mathcal{C}$ so that the continuation c is stored alongside the continuation $\langle g, l'\rangle$.

The definitions of the combinators $altStart_2$ and $altOp_2$ are modified slightly:

$$altStart_2(\tau, s, k, c)(\sigma) = \sigma$$
$$altOp_2(p, q)(\tau, s, k, c) = p(\tau, s, k, c) \circ q(\tau, s, \epsilon, k, c)$$

The command returned by an application of $altStart_2$ makes no changes to the data structures. The command returned by $altOp_2$ is the composition of the commands for its two operands p and q so that both operands exhibit their effects.

The definition of $term_2$ shows that failing to match the next terminal in the input t-string means that the data structures remain unchanged:

$$
\begin{aligned}
&term_2(s) = \langle s, term\_parser_2 \rangle \\
&term\_parser_2(t_0 \dots t_n, X ::= \alpha s \bullet \beta, l, k, c)(\sigma) = \\
&\quad \begin{cases} c(\langle X ::= \alpha s \bullet \beta, l, k+1 \rangle)(\sigma) & \textbf{if } t_k = s \\ \sigma & \textbf{otherwise} \end{cases}
\end{aligned}
$$

The continuation is applied if the terminal at position k of the input t-string matches s. Note that as in the previous subsection, symbol expressions evaluate to a pair of which the first component is a symbol and that this symbol is required for computing the grammar slots.

The new definitions of *seqStart* and *seqOp* differ in that they apply $continue_2$ (given later) rather than $continue_1$:

$$
\begin{aligned}
&seqStart_2(\tau, X, \beta, l, c_0) = continue_2(\langle X ::= \bullet \beta, l, l, l \rangle, c_0) \\
&seqOp_2(p, \langle s, q \rangle)(\tau, X, \beta, l, c_0) = p(\tau, X, s\beta, l, c_1) \\
&\quad \textbf{with } c_1(\langle X ::= \alpha \bullet s\beta, l, k \rangle) = q(\tau, X ::= \alpha s \bullet \beta, l, k, c_2) \\
&\qquad \textbf{with } c_2(\langle X ::= \alpha s \bullet \beta, l, r \rangle) = \\
&\qquad\quad continue_2(\langle X ::= \alpha s \bullet \beta, l, k, r \rangle, c_0)
\end{aligned}
$$

As $continue_1$, the function $continue_2$ receives a BSR together with a continuation function c, and it applies c to a descriptor. However, c is only applied if the descriptor is not in $\mathcal{U}$. The descriptor is added to $\mathcal{U}$ as part of the application of c. The BSR element is added to Υ independent of whether the descriptor is in $\mathcal{U}$.

$$
\begin{aligned}
&continue_2(\langle X ::= \alpha \bullet \beta, l, k, r \rangle, c)(\langle \mathcal{U}, \mathcal{G}, \mathcal{P}, \mathcal{C}, \Upsilon \rangle) = \\
&\quad \begin{cases} c(\langle X ::= \alpha \bullet \beta, l, r \rangle)(\langle \mathcal{U} \cup \mathcal{U}', \mathcal{G}, \mathcal{P}, \mathcal{C}, \Upsilon \cup \Upsilon' \rangle) \\ \qquad\qquad \textbf{if } \langle X ::= \alpha \bullet \beta, l, r \rangle \notin \mathcal{U} \\ \langle \mathcal{U}, \mathcal{G}, \mathcal{P}, \mathcal{C}, \Upsilon \cup \Upsilon' \rangle \quad \textbf{otherwise} \end{cases} \\
&\quad \textbf{with } \Upsilon' = \{ \langle X ::= \alpha \bullet \beta, l, k, r \rangle \mid \alpha \neq \epsilon \vee \beta = \epsilon \} \\
&\quad \textbf{and } \ \mathcal{U}' = \{ \langle X ::= \alpha \bullet \beta, l, r \rangle \}
\end{aligned}
$$

Recall the definition of $nterm_1$. The subexpression p of $nterm_1(s, p)$ is evaluated with a continuation function that may be applied to a descriptor of the form $\langle s ::= \delta \bullet, k, r \rangle$, indicating that the alternate δ of s derives $\tau^{k,r}$. In GLL, a descriptor of this form is processed by $process_\epsilon$ and involves executing the continuations found for the commencement $\langle s, k \rangle$ in $\mathcal{G}$ (corresponding to the **ascend** action). The function *cont_for* defined below creates a continuation function for a given nonterminal s and choice expression p that executes the continuations retrieved from $\mathcal{G}$ (directly rather than via a worklist).

$$
\begin{aligned}
&cont\_for(s, p)(\langle s ::= \delta \bullet, k, r \rangle)(\langle \mathcal{U}, \mathcal{G}, \mathcal{P}, \mathcal{C}, \Upsilon \rangle) = \\
&\quad \circ \{ c(\langle g, l', r \rangle) \mid \langle \langle s, k \rangle, \langle g, l' \rangle \rangle \in \mathcal{G}, \langle \langle g, l' \rangle, c \rangle \in \mathcal{C} \}(\sigma') \\
&\quad \textbf{with } \sigma' = \langle \mathcal{U}, \mathcal{G}, \mathcal{P} \cup \{ \langle \langle s, k \rangle, r \rangle \}, \mathcal{C}, \Upsilon \rangle
\end{aligned}
$$

For each continuation $\langle g, l' \rangle$ found in $\mathcal{G}$, the corresponding continuation function c is found in $\mathcal{C}$ and c is applied to the descriptor $\langle g, l', r \rangle$, resulting in a command for each continuation. The commands are composed; the notation $\circ\{x_1, \dots, x_n\}$ denotes the composition of the functions $x_1, \dots, x_n$ in an arbitrary order. The right extent r is recorded by adding $\langle \langle s, k \rangle, r \rangle$ to $\mathcal{P}$.

The continuation function $cont\_for(s, p)$ is given to p when $nterm\_parser_2(p)$ – the second component of $nterm_2(s, p)$ – is evaluated with some index k, but only if there is no right extent r such that $\langle \langle s, k \rangle, r \rangle \in \mathcal{P}$ (**descend**).

$$
\begin{aligned}
&nterm_2(s, p) = \langle s, nterm\_parser_2(p) \rangle \\
&nterm\_parser_2(p)(\tau, X ::= \alpha s \bullet \beta, l, k, c)(\langle \mathcal{U}, \mathcal{G}, \mathcal{P}, \mathcal{C}, \Upsilon \rangle) = \\
&\quad \begin{cases} p(\tau, s, k, cont\_for(s, p))(\sigma') & \textbf{if } R = \emptyset \\ \circ\{ c(\langle X ::= \alpha s \bullet \beta, l, r \rangle) \mid r \in R \}(\sigma') & \textbf{otherwise} \end{cases} \\
&\quad \textbf{with } R = \{ r \mid \langle \langle s, k \rangle, r \rangle \in \mathcal{P} \} \\
&\quad \textbf{and } \ \sigma' = \langle \mathcal{U}, \mathcal{G} \cup \mathcal{G}', \mathcal{P}, \mathcal{C} \cup \mathcal{C}', \Upsilon \rangle \\
&\quad \textbf{and } \ \mathcal{G}' = \{ \langle \langle s, k \rangle, \langle X ::= \alpha s \bullet \beta, l \rangle \rangle \} \\
&\quad \textbf{and } \ \mathcal{C}' = \{ \langle \langle X ::= \alpha s \bullet \beta, l \rangle, c \rangle \}
\end{aligned}
$$

If there are such right extents (**skip**), i.e. $R \neq \emptyset$, then continuation function c, given to $nterm\_parser_2(p)$, is applied to each descriptor $\langle X ::= \alpha s \bullet \beta, l, r \rangle$ with $r \in R$. The slot $X ::= \alpha s \bullet \beta$ is the slot computed for the parent of $nterm_2(s, p)$. The slot forms a continuation together with the given left extent l. The continuation is stored in $\mathcal{G}$ and is mapped to c in $\mathcal{C}$. The definition of $nterm_2$ shows a striking resemblance with the memoisation combinator of Johnson's recognition combinators [15], as is briefly discussed in Section 10.

Function $parse_2$ is similar to $parse_1$:

$$
\begin{aligned}
&parse_2(\langle s, f \rangle)(\tau) = \Upsilon \\
&\quad \textbf{with } \langle \mathcal{U}, \mathcal{G}, \mathcal{P}, \mathcal{C}, \Upsilon \rangle = f(\tau, X ::= s \bullet, 0, 0, \hat{c})(\hat{\sigma}) \\
&\quad \textbf{and } \ \hat{c}(\langle X ::= s \bullet, 0, r \rangle)(\sigma) = \sigma \\
&\quad \textbf{and } \ \hat{\sigma} = \langle \emptyset, \emptyset, \emptyset, \emptyset, \emptyset \rangle \\
&\quad \textbf{and } \ X \text{ a fresh nonterminal}
\end{aligned}
$$

The command produced by evaluating the second component of the given symbol expression is applied to a tuple of empty sets, initialising the data structures used by the algorithm. Only the final BSR set Υ is returned by the parser. The initial continuation $\hat{c}$ returns the identity command given a descriptor $\langle X ::= s \bullet, 0, r \rangle$. The t-string τ is recognised if there is an application of $\hat{c}$ with $r = |\tau|$.

This section defined the BNF combinators *nterm*, *term*, *seqStart*, *seqOp*, *altStart*, and *altOp* as general parser combinators. The combinators are general in the sense that

for any t-string τ and symbol expression p that represents a grammar Γ it holds that $parse_2(p)(\tau)$ evaluates to a finite BSR set that embeds all derivations of τ in Γ. Non-termination due to left-recursion is avoided by using descriptors to avoid executing the same continuation a second time (see the definition of $continue_2$). It is still possible, however, to write symbol expressions for which the generated parser fails to terminate for some input t-strings. This is the case when the symbol expression does not actually represent a context-free grammar. However, there are symbol expressions that do not represent a context-free grammar for which the parser still terminates and computes a meaningful and correct result. This is an advantage of using parser combinators over grammar combinators and is considered in more detail in Section 8. The examples in Section 8 show what sort of symbol expressions do not represent context-free grammars and how meaningful results can still be produced for some of these.

7. The BNF combinator library

This section summarises the second half of [12] in which the 'BNF combinator library' is introduced as a literate Haskell program. The library is discussed in this paper to demonstrate one way in which the generalised parser combinators of the previous section can be used in practice and to show that they can be used with semantic actions. The library is also the vehicle with which the experiments in the evaluation section have been performed.

The BNF combinator library is similar in functionality and architecture to the P3 grammar combinator library presented by Ridge at SLE in 2014 [19]. The novelty of the BNF combinator library is that it implements the BNF formalism more directly compared to conventional parser combinator libraries or grammar combinator libraries. The underlying grammar-extraction algorithm is simpler compared to that of P3 and the other grammar combinator libraries mentioned in Section 5. Most importantly, the algorithm avoids binarisation of the extracted grammar. The evaluation section of [12] demonstrates the negative effects of binarisation on the runtime of the underlying parsing procedure.

This section introduces the BNF combinator library by giving example syntax descriptions in §7.1 and giving an overview of the library's architecture in §7.2. In §7.2 it is also explained how an alternative implementation of the BNF combinator library is enabled by the generalised parser combinators of the previous section. The two versions of the library are compared in Section 8.

7.1. Examples

The BNF combinator library is presented in [12] without considering lookahead, error reporting and disambiguation. An implementation with these features is available as the GLL package on Haskell's package manager Hackage [44]. The examples written in this section are written with the functions and infix operators exported by the *GLL.GrammarCombinators* module of the package.

The following Haskell code fragment shows how $\Gamma_{\mathbf{tuple}}$ of Figure 1 is described with BNF combinators.

```
bnfTuple = "Tuple"
  ⟨::=⟩ char '(' **⟩ bnfAs ⟨** char ')'
bnfAs    = "As"
  ⟨::=⟩ satisfy 0
  ⟨|||⟩ (1+) ⟨$$ char 'a' ⟨**⟩ bnfMore
bnfMore  = "More"
  ⟨::=⟩ satisfy 0
  ⟨|||⟩ (1+) ⟨$$ char ',' ⟨** char 'a' ⟨**⟩ bnfMore
```

The ⟨::=⟩ operator is the external infix version of *nterm*, ⟨|||⟩ of *altOp*, and ⟨**⟩ of *seqOp*. Function *satisfy* is the external version of *seqStart* and *char* is one of several external versions of *term*. The operator ⟨::=⟩ has the lowest precedence, the precedence of ⟨|||⟩ is higher than that of ⟨::=⟩ but lower than that of ⟨**⟩. The argument of *satisfy* is the semantic value that gives an interpretation to the alternate it constructs. For example, the nonterminal `"As"` describes the language of zero or more occurrences of the character `'a'` separated by commas. The associated semantic actions count the occurrences of `'a'`. The first alternate of `"As"` corresponds to zero occurrences, the number to which *satisfy* is applied. The operators ⟨** and **⟩ are variants of ⟨**⟩ that suppress the semantic value of their right and left operand respectively. The ⟨$$⟩ operator occurs at the start of an alternate and inserts a semantic function[8] that is applied to the semantic values produced for the symbols of the alternate, ignoring the semantic values that are suppressed. The ⟨$$ operator is a variant of ⟨$$⟩ that suppresses the semantic value of the first symbol of the alternate. The operators ⟨$$⟩, ⟨**⟩, and their variants have the same precedence and are left-associative.

The main difference compared to popular Haskell parser combinator libraries such as PARSEC and UU-LIB is the requirement to use ⟨::=⟩ to insert a nonterminal name into recursive combinator expressions. The inserted name is used internally for detecting recursion and computing grammar slots. Drawbacks and alternatives to this form of nonterminal insertion are discussed in Section 9.

An interesting advantage of combinator parsing is the ability to use the abstraction mechanism of the host language. The BNF combinator library takes advantage by defining several functions that closely correspond to popular extensions of BNF, capturing common patterns such as optionality and repetition. The definition of a function for optionality is given as an example:

```
optional p = mkNt p "optional" ⟨::=⟩ satisfy Nothing
                               ⟨|||⟩ Just ⟨$$⟩ p
```

Haskell's *Maybe* type is used to wrap the semantic value of p with *Just* if p is chosen, or to give *Nothing* otherwise. The function *mkNt* is given a BNF description and

[8] The word 'semantic function' is used instead of 'semantic action' to emphasise that it is a pure function.

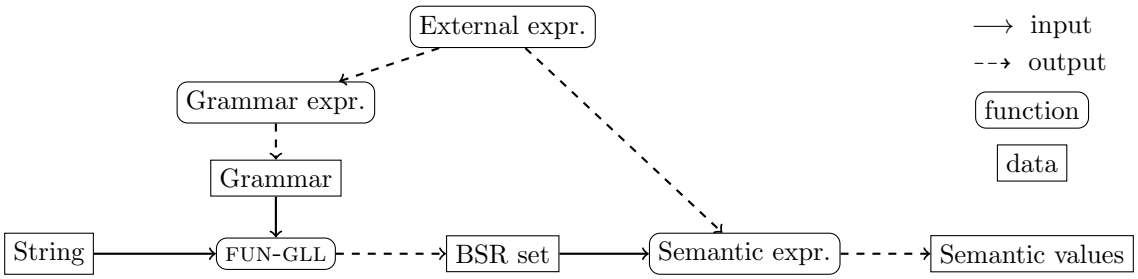

Figure 10: The architecture of the BNF combinator library with internal grammar combinators.

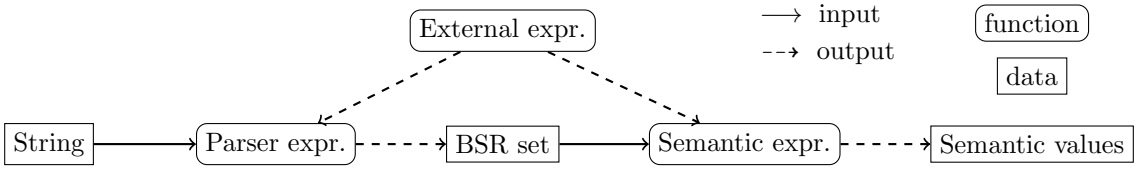

Figure 11: The architecture of the BNF combinator library with internal parser combinators.

a string and generates a unique nonterminal name based on its inputs. The function assumes that all the nonterminals in the BNF description have unique names and that the string is not used as a nonterminal name anywhere else. The application of *mkNt* is necessary to ensure that, for example, *optional* (*char* `'a'`) and *optional* (*char* `'b'`) describe different nonterminals. In general, if a BNF description is parameterised, then the parameters need to be taken into account when injecting a nonterminal name.

The following example demonstrates how Haskell's standard library function *map* can be used to effectively generate a BNF description.

$$bnfKeywords\ xs = chooses\ \texttt{"keywords"}\ (map\ keyword\ xs)$$

The function *bnfKeywords* is given a list of strings and generates a BNF description that defines the nonterminal `"keywords"` with an alternate for each string. Each alternate is formed by applying *keyword* to one of the strings. The function *keyword* is similar to *char* except that it matches a particular string rather than a single character. Function *chooses* is a variant of ⟨::=⟩ that expects a list of alternates as its second argument, enabling nonterminals with a number of alternates that is unknown statically.

7.2. Architecture of the BNF combinator library

The BNF combinator library described in [12] is implemented as a collection of 'external' combinators (available to the programmer) that generate expressions by applying the combinators of two 'internal' libraries (hidden from the programmer). Each of the three libraries implements the BNF combinators of Figure 6 in different ways, providing different instantiations for the types *Symb*, *Sequence*, and *Choice*. The interconnection between the external and internal libraries is summarised as follows:

- An expression formed by applying the internal *grammar combinators* is evaluated to yield the grammar represented by the combinator expression
- An expression formed by applying the internal *semantic combinators* has semantic actions embedded within it and computes grammar slots for its sequence expressions in the way described for the parser combinators in Section 6. The expression is evaluated given a BSR set and uses the slots to find pivots in the BSR set in order to enumerate derivations[9] and execute the semantic actions that interpret the derivations
- An expression formed by applying the external combinators generates expressions of the same structure in the two internal libraries. The generated grammar combinator expression is evaluated to produce a grammar. The grammar is given to an implementation of FUN-GLL to yield the BSR set for a given t-string. The generated semantic combinator expression is evaluated to yield the interpretations of the derivations embedded in the BSR set
- The functions and operators such as ⟨::=⟩, ⟨$$⟩, ⟨||⟩ and *satisfy*, used in the examples of §7.1, are defined as syntactic sugar on top of the external combinators

The architecture of the library is visualised in Figure 10.

The parser combinators of the previous section enable an alternative implementation of the BNF combinator library. The parser combinators can be used to replace FUN-GLL and the internal grammar combinators because

[9] Cyclic grammars that may admit infinitely many derivations of an input string are dealt with in the way described at the end of Section 4. Disambiguation strategies can be used to filter out certain pivots, for example by applying the 'longest match' rule.

Table 2: Parsing ANSI-C files (in seconds).

Tokens	1515	8411	15589	26551	36827
Grammar	0.45	2.48	4.46	8.01	10.52
No Grammar	0.48	2.62	4.89	8.36	11.41
Factor	1.07	1.06	1.10	1.04	1.08

Table 3: Parsing Caml Light files (in seconds).

Tokens	1097	4534	8846	15910	28703
Grammar	1.25	4.02	7.92	13.47	27.28
No Grammar	1.38	4.07	8.68	14.81	28.55
Factor	1.10	1.01	1.10	1.10	1.05

they compute BSR sets directly. The architecture of the alternative implementation of the library is visualised in Figure 11. Replacing the internal grammar combinators with the generalised parser combinators has proven to be straightforward. The alternative library is available as the FUNGLL-COMBINATORS package on Hackage [45]. The two implementations are compared in Section 8.

8. Evaluation

The previous section gives an overview of the 'BNF combinator library', initially introduced in [12], for writing syntax descriptions that admit generalised parsing. Examples of using the library are given in §7.1. An alternative architecture for the library is suggested at the end of §7.2. In the alternative, the generalised parser combinators of Section 6 replace the internal grammar combinators of the library. This section compares the two implementations of the library based on the alternative architectures. The libraries export their functionality in the *GLL.GrammarCombinators* and *GLL.ParserCombinators* modules. The modules can be used interchangeably as their interfaces are identical and switching between libraries for the experiments of this section is as simple as changing an import statement. The only difference is that *GLL.GrammarCombinators* uses an internal grammar. Some of the advantages of avoiding an internal grammar are discussed in §8.2. In particular, §8.2 shows that parsers can be obtained for a larger class of syntax descriptions, including descriptions that do not specify a context-free grammar. In §8.1, the running times for parsing large inputs for real-world languages are compared.

All tests in this section have been executed without lookahead tests under Ubuntu 14.04 on a laptop with quad-core 2.4GHz processors and 8GiB of RAM. Lookahead has been switched off because the implementation of the BNF combinator library with internal parser combinators does not currently support a form of lookahead. Possible approaches to extending the generalised parser combinators with lookahead are discussed in Section 9.

8.1. Performance

The running times of *GLL.GrammarCombinators* and *GLL.ParserCombinators* are compared by running parsers for the syntax descriptions of three software languages: two programming languages – ANSI-C and Caml Light – and the semantic specification language CBS [24]. The running times include a lexicalisation phase which produces a sequence of 'tokens', given as input to the parsing phase. In the case of CBS, the running times include semantic evaluation and disambiguation, producing an abstract syntax tree that is printed to a file. For each language, considerable software-language engineering projects[10] have been selected: an in-house 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 in order to create files of several sizes. The running times are shown in Tables 2, 3, and 4. The rows starting with 'Grammar' and 'No Grammar' show the running times of the two alternative implementations of the BNF combinator library, with and without generating an internal grammar. The rows starting with 'Factor' shows the relative difference in running times between the two implementations.

Table 4: Parsing and pretty-printing CBS files (in seconds).

Tokens	2653	14824	17593	21162	26016
Grammar	1.30	11.68	15.07	19.63	29.54
No Grammar	1.10	9.18	11.28	15.16	21.80
Factor	0.85	0.79	0.75	0.77	0.75

The syntax description of ANSI-C is a direct transcription of the grammar listed by Kernighan and Ritchie [22]. This grammar is written in BNF (without extensions). The grammar is nondeterministic and left-recursive. The grammar generated by the internal grammar combinators of *GLL.GrammarCombinators* has 229 alternates and 71 nonterminals. The syntax description of Caml Light is taken from the Caml Light reference manual by Leroy [23]. The grammar is highly nondeterministic 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. The grammar generated by the internal grammar combinators of *GLL.GrammarCombinators* has 285 alternates and 134 nonterminals. The syntax description for CBS is nondeterministic and has several sources of ambiguity. The grammar generated by the internal grammar combinators of *GLL.GrammarCombinators* has 257 alternates and 126 nonterminals.

The experiments show that for ANSI-C and Caml Light there is a decrease in performance of between 4% and 12% using the internal parser combinators. However, for CBS there is a performance increase of 15-25%. The main conclusion is that both implementations of the BNF combi-

[10] The three case studies are taken from [12]. The rows labelled 'Grammar' correspond to the rows labelled 'Flexible' in [12]. The numbers differ slightly because the experiments were run again.

nator library are practicall to use for real-world languages, even without specialised performance engineering, thus also demonstrating that practical implementations of the generalised parser combinators are possible. The experiment is in part a repeat of the experiment in [12], which did not include the generalised parser combinator alternative ('No Grammar'), but instead demonstrated the advantage of the internal grammar combinators ('Grammar') over a variant that generates binarised grammars (both with and without lookahead). More information about the languages and test strings can be found in [12].

8.2. The benefits of avoiding grammar generation

Both implementations of the BNF combinator library are general in the sense that terminating parsers are available for all syntax descriptions that describe context-free grammars. However, syntax descriptions that do not describe a context-free grammar can be written, in which case the grammar extraction algorithm of the internal grammar combinators fails to terminate. Moreover, syntax descriptions that produce very large grammars for which parsing is not practical can also be written. An example of the latter is given by permutation phrases. An example of the former is given at the end of this section.

Permutation phrases. A permutation phrase is a sequence of permutable elements in which each element occurs exactly once and in any order [20]. The following experiment involves a variation of permutation phrases that allows elements to be included optionally, i.e. elements occur *at most* once. Real-world examples of permutation phrases are provided by the 'modifiers' associated with fields and methods in Java [42] and the 'declaration specifiers' of C [20]. For example, the optional modifiers `'static'`, `'final'`, and `'public'` can be written in any order in the signature of a Java method.

In BNF, the syntax of permutation phrases can be captured by a nonterminal with an alternate for each of the possible permutations of the elements. Such a syntax description is not desirable for several reasons, e.g. the description is error-prone and tedious to maintain. Moreover, the number of alternates grows exponentially in the number of permutable elements. The syntax of permutation phrases can be capture more conveniently with parser combinators [21]. With the BNF combinator library, the syntax of permutation phrases of length three can be described as follows:

```
bnfPermP3 m1 m2 m3 =
  foldr mkNt "Perm3" [m1, m2, m3]
    ⟨::=⟩ satisfy []
      ⟨||⟩ (:) ⟨$$⟩ m1 ⟨**⟩ bnfPermP3 fails m2 m3
      ⟨||⟩ (:) ⟨$$⟩ m2 ⟨**⟩ bnfPermP3 m1 fails m3
      ⟨||⟩ (:) ⟨$$⟩ m3 ⟨**⟩ bnfPermP3 m1 m2 fails
```

This example is an adapted version of the `PermP3` example of [42]. Function *fails* is the external version of *altStart*. The function *bnfPermP3* receives the BNF descriptions of three permutable elements *m1*, *m2*, and *m3*. The second alternate selects *m1* and the recursive call disables *m1* for further selection by replacing it with *fails* (and similarly for the third and fourth alternate). The first alternate selects no element. The semantics of a permutation phrase is a list of semantic values for the elements in the order the elements occurred in the phrase. The syntax of Java modifiers is expressed by specialising *bnfPermP3*:

```
bnfModifiers = bnfPermP3 (keyword "static")
                         (keyword "final") (keyword "public")
```

The usage of *mkNt* in the definition of *bnfPermP3* guarantees that each recursive application of *bnfPermP3* to different arguments results in different nonterminals being injected with ⟨::=⟩. In variants of *bnfPermP3* with larger number of permutable elements, the number of recursive applications with different arguments grows exponentially. As a result, the size of the grammars generated by the internal grammar combinators grows exponentially. To demonstrate this, a function has been defined that generates the BNF syntax description of permutation phrases of n elements, with n as input to the function. The function has been called with several values for n. For each value of n the number of nonterminals of the internal grammar has been recorded, together with the number of seconds it takes to parse a particular permutation of n elements. The results are given in Table 5 and show how grammar size and running times grow exponentially. The running times are dominated by grammar generation. Running the same experiments with the internal parser combinators shows the possibility to parse much larger permutations. These results are given in Table 6. The running times in Table 6 are dominated by executing the semantics.

Table 5: Parsing permutations with internal grammar combinators.

#Elements	10	11	12	13	14
Time (sec)	0.16	0.5	1.05	2.17	5.24
#Nonterminals	1028	2052	4100	8196	16388

Table 6: Parsing permutations with internal parser combinators.

#Elements	50	100	150	200	250
Time (sec)	0.11	0.75	2.36	5.55	10.75

Beyond context-free languages. The example of permutation phrases shows that using recursion it is possible to write syntax descriptions that produce exponentially large internal grammars. The following example shows that using recursion it is also possible to give syntax descriptions for grammars that are not context-free. Such descriptions cause nontermination when executed with internal grammar combinators. However, parsing is still possible in some cases with the internal parser combinators. Consider the following syntax description.

```
scales_a = scales (char 'a')
scales p = mkNt p "scales"
  ⟨::=⟩ 1     ⟨$$ p
  ⟨||⟩  (1+) ⟨$$ p ⟨**⟩ scales (parens p)
parens q =
  mkNt q "parens" ⟨::=⟩ char '(' **⟩ q ⟨** char ')'
```

The argument of *scales* changes in every recursive call so that *mkNt* produces a new nonterminal name for every recursive call. As a result, attempting to generate a grammar for *scales_a* results in nontermination. The fact that no grammar can be generated for *scales_a* is unsurprising given that it describes the language {"a", "a(a)", "a(a)((a))", ...} which is not context-free. However, with the internal parser combinators, this example does give a terminating parser that recognises the language. The parser terminates because the index of the input t-string is increased before a recursive call is made. Eventually, the end of the t-string is reached and no alternate of *scales* is applicable. The syntax description *scales_a′*, given below, for the language {"a", "(a)a", "((a))(a)a", ...} does not admit a terminating parser.

```
scales_a' = scales' (char 'a')
scales' p = mkNt p "scales'"
  ⟨::=⟩ 1     ⟨$$ p
  ⟨||⟩  (1+) ⟨$$⟩ scales' (parens p) ⟨** p
```

In this case, a recursive call is made without getting closer to the end of the input t-string. This problem is similar to that of left-recursion. The problem is different, however, as evidenced by the observation that GLL's solution to left-recursion does not apply. A recursive call for a left-recursive non-terminal results in a descriptor being created which is already in the set $\mathcal{U}$ (and therefore not processed). In this case, a recursive call results in a new nonterminal name and thus in a descriptor not present in $\mathcal{U}$.

This section has demonstrated that parsers can use the binary subtree representation of derivations even if no (context-free) grammar is known beforehand or possible. The BNF combinator library with internal parser combinators has been shown to exhibit several benefits over the alternative with internal grammar combinators. Firstly, by using parser combinators internally, exponential running times are avoided for syntax descriptions that generate exponentially large internal grammars. Permutation phrases have been given as an example. Secondly, parsers can be obtained for syntax descriptions that do not describe context-free languages. Monadic parser combinators also extend parsing beyond context-free languages [33] (see Section 10 for a brief comparison).

9. Conclusions and future work

FUN-GLL. The first part of this paper gives a purely functional description of FUN-GLL, a generalised top-down (GLL) parsing procedure. In contrast to the original descriptions of GLL parsing [3, 4], the data structures used by the algorithm are modelled by abstract sets and relations rather than specialised implementations. The binary subtree representation (BSR) of [11] makes it possible to collect all derivations of an input t-string without the need for maintaining and constructing graphs.

The FUN-GLL algorithm is defined as a collection of functions that are written in a notation that can be seen as a functional pseudo-code. Implementations of FUN-GLL in functional languages can be derived more or less directly from these definitions, of which an example is provided by the Haskell implementation in [12]. The main challenge for the programmer is to develop efficient implementations of the algorithm's data structures. The reader is referred to [29] for a discussion on GLL's data structures. Implementations in procedural languages should be equally straightforward and might benefit from iteration and mutable data. An important motivation for the work presented in this paper has been to make GLL parsing accessible to a wider audience.

FUN-GLL parser combinators. The second part of this paper shows that parser combinators that employ a generalised top-down parsing algorithm very similar to FUN-GLL can be defined. Conventional parser combinators, which only represent grammar descriptions superficially, cannot implement generalised parsing algorithms such as GLR, GLL, and Earley because these algorithms require grammar slots. The generalised parser combinators of this paper force the programmer to inject nonterminal names so that grammar slots can be computed. Computing grammar slots is simplified by restricting the applicability of the combinators and thereby giving combinator expressions a richer structure. Several other approaches to generalising the combinator approach to parsing have been discussed in Section 5.

The algorithm underlying the generalised parser combinators uses descriptors to prevent duplicate work and to prevent non-termination in the face of left-recursion, exactly as the FUN-GLL algorithm. The algorithm guarantees that all derivation information is discovered by using the data structures of FUN-GLL in the same way as the FUN-GLL algorithm. The main difference between the two algorithms is that the algorithm underlying the parser combinators does not use a worklist to process descriptors. This design decision preserves a strong connection to conventional parser combinator definitions. The algorithm has been presented by the gradual extension of simple recognition combinators to parser combinators that compute BSR sets to the generalised parser combinators.

Although the generalised parser combinators compute grammar slots, they crucially do not compute an actual grammar. As demonstrated in Section 8, this makes it possible to write parsers for permutation phrases that would otherwise induce exponentially large grammars. Parsers can also be written for languages that are not context-free.

These observations demonstrate that the BSR representation of derivations can be used in situations in which no context-free grammar is available (or possible).

The benefits of the generalised parser combinators are not limited to applications in combinator parsing. The combinator definitions may provide inspiration for developing the backend of a parser generator. A straightforward implementation of such a backend produces a combinator expression that represents the input grammar by applying the combinators directly. More efficient implementations can generate variants of the combinators that are specialised with respect to particular grammar fragments and then combine the specialised variants to form a parser.

Adding a form of lookahead to the generalised parser combinators is more complicated than it is for grammar combinators because there is no separate grammar object. Multiple approaches to adding lookahead are possible and have been considered. For example, lookahead sets can be pre-computed alongside the grammar slots (§6.2). Alternatively, lookahead sets can be computed on the fly in a form of caching. The technical presentation and evaluation of these approaches are left as future work.

The BNF combinator library. In [12], the BNF combinator library is introduced for developing syntax descriptions that admit generalised parsing. 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 for combining syntax descriptions, enabling 'reuse through abstraction'.

Experience has shown that the BNF combinator library is practical and easy to use. The library has been used to describe the syntax of several software languages (including ANSI-C and Caml Light). The syntax descriptions are easy to develop, verify and debug. Without specialised engineering, the parsers for these syntax descriptions show acceptable running times. The benefits of developing syntax without having to consider left-factoring or left-recursion removal are worth the price of generalised parsing. Moreover, if parsing speed is essential, the descriptions can be refactored for efficiency. Earlier work by two of the authors of this paper has demonstrated that GLL algorithms can be extended to perform left-factoring automatically [28]. Adjusting this technique to work with BSR representations, bringing it into the scope of the algorithms of this paper, is an interesting line of future work.

There are two main caveats concerning the usability of the BNF combinators. Firstly, as a language evolves, it is hard to keep track of 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 [46]. Impure alternatives typically involve the automatic generation of references for nonterminals [38, 37, 18].

Secondly, disambiguation is required for ambiguous syntax, before or during semantic evaluation. The current ambiguity reduction strategies are low-level, defined directly on BSR 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 BSR sets.

10. Related work

Section 5 discussed the work of several authors on parser combinators and on approaches to generalising parser combinators. In [12], Ridge's P3 library is compared with the BNF combinator library.

GLL parsing. Since generalised top-down (GLL) parsing emerged [3, 4], several GLL algorithms have been published [28, 47]. These algorithms are described in low-level pseudo-code as the output of a parser generator. The FUN-GLL has been described as a collection of pure functions and at a higher level of abstraction, abstracting over the grammar and modelling the data structures as sets.

Spiewak has also adapted GLL to a functional setting by defining 'GLL combinators' in SCALA [48]. 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. However, to ensure that at least one derivation is preserved by disambiguation it is necessary, in general, to use a data structure, such as a BSR set, as an intermediary between parsing and semantic evaluation.

FUN-GLL can be seen as a high-level, functional description of RGLL [28], bearing also a striking resemblance to Johnson's recognition combinators. The connection between Johnson's combinators and GLL has also been observed by Afroozeh and Izmaylova [47]. 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. The 'continuations' recorded by Johnson's combinators are the continuation functions of the continuation-passing style in which the combinators are defined. The right extents and continuation functions are recorded together in a memoisation table for the pair $\langle X, l\rangle$, referred to as a 'commencement' in the description of FUN-GLL. In FUN-GLL, continuations are modelled by a pair of a grammar slot and a left extent, and are recorded in relation $\mathcal{G}$ that serves the same purpose as the GSS of GLL. The FUN-GLL-based parser combinators are also defined in continuation-passing style, and record continuation functions alongside continuations in the relation $\mathcal{C}$. Johnson's recognition combinators can be modified to return BSR sets, thus extending the combinators to generalised parser combinators (demonstrated in [11] and [26]).

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 [49], 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. Devriese and Piessens show how grammar combinators are defined in 'finally tagless' style [46], which is one of several techniques developed to overcome these differences [50, 51].

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 [52]. These techniques can also be applied to the BNF combinator library, 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 combinators at compile-time [17].

Monadic parser combinators. As demonstrated in Section 8, the generalised parser combinators of this paper make it possible to obtain parsers for languages that are not context-free. Monadic parser combinators also extend parsing beyond context-free languages by enabling the construction of parsers based on semantic values produced by earlier parse results [33]. A theoretical or practical comparison has not been made between the two approaches. For example, it is interesting to compare the classes of languages for which parsers can be written. Note that monadic parsing and generalised parsing are fundamentally at odds because, in order to support principled disambiguation strategies, generalised parsers need to produce intermediate representations of derivations (such as parse forests or BSR sets) and should not perform evaluation on the fly to avoid spurious, premature results.

Macro-grammars. This paper has given examples that show the abstraction mechanism of Haskell can be leveraged by the BNF combinator library to abstract over grammar fragments. As shown in §8.2, the ability to abstract over grammar fragments makes it possible to describe the syntax of languages that are not context-free.

In 1968, Fischer introduced the inside-out (IO) and outside-in (OI) macro-grammars as strict super-classes of the context-free grammars [41]. Macro-grammars obtain their power by allowing production rules to abstract over sequences of symbols. The BNF combinator library can be used to describe OI macro-grammars. However, as explained in §8.2, a terminating parser cannot be obtained for all of these descriptions.

In [42], Thiemann and Neubauer analyse the conditions under which macro-grammars can be transformed into equivalent context-free grammars. Thiemann and Neubauer present an algorithm that determines whether a macro-grammar satisfies these conditions. It may be possible to incorporate this algorithm in the BNF combinator library in order to prevent nontermination when generating internal grammars. An extension to the algorithm may be possible as well, enabling the algorithm to determine exactly whether syntax descriptions can be handled by the internal parser combinators.

Acknowledgements

We thank the anonymous reviewers of SLE for helpful comments and suggestions on earlier versions of this paper.

References

[1] J. Earley, An efficient context-free parsing algorithm, Communications of the ACM 13 (2) (1970) 94–102.

[2] M. Tomita, Efficient Parsing for Natural Language: A Fast Algorithm for Practical Systems, Kluwer Academic Publishers, 1985.

[3] E. Scott, A. Johnstone, 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).

[4] E. Scott, A. Johnstone, GLL parse-tree generation, Science of Computer Programming 78 (10) (2013) 1828 – 1844.

[5] E. Scott, A. Johnstone, R. Economopoulos, BRNGLR: a cubic tomita-style GLR parsing algorithm, Acta Informatica 44 (6) (2007) 427–461.

[6] E. Scott, A. Johnstone, Recognition is not parsing - SPPF-style parsing from cubic recognisers, Science of Computer Programming 75 (1-2) (2010) 55–70.

[7] E. Visser, G. Wachsmuth, A. Tolmach, P. Neron, V. Vergu, A. Passalaqua, G. Konat, A language designer's workbench: A one-stop-shop for implementation and verification of language designs, in: Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software, Onward! 2014, ACM, 2014, pp. 95–111. doi:10.1145/2661136.2661149.

[8] G. Roşu, T. F. Şerbănuţă, K overview and simple case study, Electronic Notes in Theoretical Computer Science 304 (2014) 3 – 56.

[9] M. G. J. van den Brand, J. Heering, P. Klint, P. A. Olivier, Compiling language definitions: The ASF+SDF compiler, ACM Trans. Program. Lang. Syst. 24 (4) (2002) 334–368.

[10] B. Basten, J. van den Bos, M. Hills, P. Klint, A. Lankamp, B. Lisser, A. van der Ploeg, T. van der Storm, J. J. Vinju, Modular language implementation in Rascal - experience report, Sci. Comput. Program. 114 (C) (2015) 7–19. doi:10.1016/j.scico.2015.11.003.
URL `https://doi.org/10.1016/j.scico.2015.11.003`

[11] E. Scott, A. Johnstone, L. T. van Binsbergen, Derivation representation using binary subtree sets, Science of Computer Programming 175 (2019) 63 – 84. doi:10.1016/j.scico.2019.01.008.

[12] L. T. van Binsbergen, E. Scott, A. Johnstone, GLL parsing with flexible combinators, in: Proceedings of the 11th ACM SIGPLAN International Conference on Software Language Engineering, SLE 2018, 2018.

[13] P. Norvig, Techniques for automatic memoization with applications to context-free parsing, Computational Linguistics 17 (1) (1991) 91–98.

[14] S. D. Swierstra, Combinator parsing: A short tutorial, in: Language Engineering and Rigorous Software Development, LNCS 5520, Springer Berlin Heidelberg, 2009, pp. 252–300.

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

[16] A. I. Baars, S. D. Swierstra, Type-safe, self inspecting code, in: Proceedings of the 2004 ACM SIGPLAN Workshop on Haskell, Haskell '04, ACM, 2004, pp. 69–79.

[17] D. Devriese, F. Piessens, Explicitly recursive grammar combinators, in: Proceedings of the 13th International Symposium on Practical Aspects of Declarative Languages, PADL 2011, Springer Berlin Heidelberg, 2011, pp. 84–98.

[18] P. Ljunglöf, Pure functional parsing, Ph.D. thesis, Chalmers University of Technology and Göteborg University (March 2002).

[19] T. Ridge, Simple, efficient, sound and complete combinator parsing for all context-free grammars, using an oracle, in: Software Language Engineering, LNCS 8706, Springer International Publishing, 2014, pp. 261–281.

[20] R. D. Cameron, Extending context-free grammars with permutation phrases, ACM Letters on Programming Languages and Systems 2 (1-4) (1993) 85–94. doi:10.1145/176454.176490.

[21] A. I. Baars, A. Löh, S. D. Swierstra, Parsing permutation phrases, Journal of Functional Programming 14 (6) (2004) 635–646. doi:10.1017/S0956796804005143.

[22] B. W. Kernighan, D. M. Ritchie, The C programming language, Prentice Hall, 1988, p. Appendix 13.

[23] X. Leroy, Caml Light manual, `http://caml.inria.fr/pub/docs/manual-caml-light` (1997).

[24] L. T. van Binsbergen, P. D. Mosses, N. Sculthorpe, Executable component-based semantics, Journal of Logical and Algebraic Methods in Programming 103 (2019) 184–212. doi:10.1016/j.jlamp.2018.12.004.

[25] D. E. Knuth, Backus normal form vs. Backus Naur form, Communications of the ACM 7 (12) (1964) 735–736. doi:10.1145/355588.365140.

[26] L. T. van Binsbergen, Executable formal specification of programming languages with reusable components, Ph.D. thesis, Royal Holloway, University of London (2019).

[27] P. Wadler, 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., 1985, pp. 113–128.

[28] E. Scott, A. Johnstone, Structuring the GLL parsing algorithm for performance, Science of Computer Programming 125 (2016) 1 – 22.

[29] A. Johnstone, E. Scott, Modelling GLL parser implementations, in: Software Language Engineering, LNCS 6563, Springer Berlin Heidelberg, 2011, pp. 42–61.

[30] E. Visser, Syntax definition for language prototyping, Ph.D. thesis, University of Amsterdam (1997).

[31] J. J. Vinju, Analysis and transformation of source code by parsing and rewriting, Ph.D. thesis, University of Amsterdam (2005).

[32] A. Afroozeh, A. Izmaylova, Practice general top-down parsers, Ph.D. thesis, University of Amsterdam (2019).

[33] D. Leijen, E. Meijer, Parsec: Direct style monadic parser combinators for the real world, Tech. Rep. UU-CS-2001-35, Department of Computer Science, Universiteit Utrecht (2001).

[34] R. A. Frost, R. Hafiz, P. Callaghan, Parser combinators for ambiguous left-recursive grammars, in: Practical Aspects of Declarative Languages, LNCS 4902, Springer Berlin Heidelberg, 2008, pp. 167–181.

[35] A. Izmaylova, A. Afroozeh, T. v. d. Storm, Practical, general parser combinators, in: Proceedings of the 2016 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation, PEPM 2016, ACM, 2016, pp. 1–12.

[36] R. A. Frost, B. Szydlowski, Memoizing purely functional top-down backtracking language processors, Science of Computer Programming 27 (3) (1996) 263–288.

[37] K. Claessen, D. Sands, Observable sharing for functional circuit description, in: In Asian Computing Science Conference, Springer Verlag, 1999, pp. 62–73.

[38] A. Gill, Type-safe observable sharing in Haskell, in: Proceedings of the 2Nd ACM SIGPLAN Symposium on Haskell, Haskell '09, ACM, 2009, pp. 117–128.

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

[40] S. Marlow, A. Gill, Happy - the parser generator for Haskell, `https://www.haskell.org/happy/`, [Online, accessed 10-July-2018] (2001).

[41] M. J. Fischer, Grammars with macro-like productions, in: 9th Annual Symposium on Switching and Automata Theory, SWAT 1968, 1968, pp. 131–142. doi:10.1109/SWAT.1968.12.

[42] P. Thiemann, M. Neubauer, Macros for context-free grammars, in: Proceedings of the 10th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming, PPDP 2008, ACM, 2008, pp. 120–130. doi:10.1145/1389449.1389465.

[43] S. D. Swierstra, P. R. A. Alcocer, J. a. Saraiva, Designing and implementing combinator languages, in: Advanced Functional Programming, Springer Berlin Heidelberg, 1999, pp. 150–206.

[44] L. T. van Binsbergen, Hackage Repository `gll`, `https://hackage.haskell.org/package/gll`, [Online, accessed 29-March-2019] (2015).

[45] L. T. van Binsbergen, Hackage repository `fungll-combinators`, `https://hackage.haskell.org/package/fungll-combinators`, [Online, accessed 29-March-2019] (2019).

[46] D. Devriese, F. Piessens, Finally tagless observable recursion for an abstract grammar model, Journal of Functional Programming 22 (6) (2012) 757–796.

[47] A. Afroozeh, A. Izmaylova, Faster, practical GLL parsing, in: Compiler Construction, Springer Berlin Heidelberg, 2015, pp. 89–108.

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

[49] J. Gibbons, N. Wu, Folding domain-specific languages: Deep and shallow embeddings (functional pearl), in: Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming, ICFP '14, 2014, pp. 339–347.

[50] J. Svenningsson, E. Axelsson, Combining deep and shallow embedding for EDSL, in: Trends in Functional Programming, 2013, pp. 21–36.

[51] J. Carette, O. Kiselyov, C.-C. Shan, 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, 2007, pp. 222–238.

[52] J. Duregård, P. Jansson, Embedded parser generators, in: Proceedings of the 4th ACM Symposium on Haskell, Haskell '11, ACM, 2011.