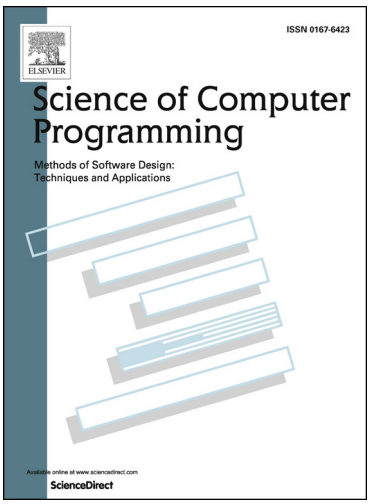
Accepted Manuscript



Derivation representation using binary subtree sets

Elizabeth Scott, Adrian Johnstone, L. Thomas van Binsbergen

PII: S0167-6423(18)30230-2
DOI: https://doi.org/10.1016/j.scico.2019.01.008
Reference: SCICO 2268

To appear in: *Science of Computer Programming*

Received date: 12 June 2018
Revised date: 31 October 2018
Accepted date: 15 January 2019

Please cite this article in press as: E. Scott et al., Derivation representation using binary subtree sets, *Sci. Comput. Program.* (2019), https://doi.org/10.1016/j.scico.2019.01.008

Highlights

- Introduces BSR sets, an alternative to the SPPF representation of all derivations of a string.
- General BSR combinator parsers and Earley-style parsers are presented.
- A new BSR GLL-style parser (CNP) is presented which has a more efficient combined call. stack
- Data structure size comparisons for GLL and CNP are given for C, C^{++} and Java grammars.

Derivation representation using binary subtree sets

Elizabeth Scott, Adrian Johnstone and L. Thomas van Binsbergen
Royal Holloway, University of London

Abstract

This paper introduces sets of binary subtree representations as an alternative to shared packed parse forests as the output of a generalised parser, and shows how these may be generated by Earley's algorithm, by a new GLL-style parser and by Johnson's continuation passing combinator style parsers. The set based output removes the clerical overhead associated with graph constructions, making the parsers simpler.

A recogniser for a language, $L(\Gamma)$, specified by a grammar, Γ, is an algorithm that takes as input a string, u say, and returns *success* if $u \in L(\Gamma)$ and *failure* otherwise. A parser for $L(\Gamma)$ returns a representation of the structure of u with respect to Γ. It is not always easy to extend a recogniser to a parser [29]. The primary purpose of this paper is to introduce BSR sets and to advocate BSR set generation as a simple approach for extending language recognisers to parsers. The concept of a BSR set is not radical; BSR sets are closely related both to shared packed parse forests [33], which are commonly used to represent derivations, and to the sets of items constructed by an Earley recogniser [8]. However, our BSR set definition is deliberately independent of any particular parsing technique, allowing BSR set generation to be easily added to various different recogniser techniques. We support the latter claim by demonstrating it for three quite different techniques: Earley recognisers, continuation passing style combinator parsers, and GLL style parsers.

It is perhaps not immediately clear that recognition and derivation construction is still a problem for computer languages. For cases where the sentence structure is uniquely defined, well known derivation generation techniques exist; but they impose constraints even on non-ambiguous grammars. Generalised techniques can be applied to all grammars, both those that are nondeterministic in the Knuth sense [16] and those for which the structure of some $u \in L(\Gamma)$ is ambiguous (not uniquely defined). Generalised language analysis techniques free language designers from algorithm specific constraints, and facilitate semantic specification. The desirability of generalised parsers is reflected in the development of general extensions to the classical, restricted, LR and LL techniques, and the incorporation of these extensions in tools such as ASF+SDF [36], Elkhound [20], Stratego/Spoofax [37], Rascal [15] and the K-tool [25]. Most of these tools use a generalised LR technique, but the attraction of the simpler and easier to understand LL technique is evidenced by the continued use of extended, but not fully general, versions for example in the combinator approach [22] and tools such as ANTLR [23]. A practical, fully general, LL technique was given in [30] and a general combinator parser which deploys an 'oracle' was given in [24].

Generalised parsers need to produce representations of all the possible structurings (derivations) of an input string (subject, potentially, to specified disambiguation rules). However, there are infinitely many derivations in the case of grammars with cycles, and even for cycle-free grammars there can be an unbounded polynomial number of derivations. In 1985 Tomita [33] introduced an efficient derivation representation, a shared packed parse forest (SPPF).

SPPFs were studied in detail in [5], and Johnson [13] pointed out that in the worst case an SPPF can be of unbounded polynomial size. However, it is reasonably straightforward to modify the structure into a binarised form which is worst case cubic size [30]. One problem, however, is that the machinery needed to construct SPPFs adds considerably to the complexity of a parser over a corresponding recogniser. This is not simply because of the specific nature of the SPPFs. As we will discuss further below, it turned out to be surprisingly difficult to turn Earley's classic general recogniser into a parser, and Tomita's original generalised LR parser also contains an error.

In this paper we introduce an alternative derivation representation which is a set rather than a graph. This allows the specification of the parsers to be simplified because there is no need for graph construction machinery, and the information required to build the graph does not need to be carried around. No edges are computed or stored. The elements of the set represent fragments of an SPPF, and we refer to the set as a binary subtree representation (BSR) of the input string. It is straightforward to reconstruct the SPPF, if required, via a separate function which is reusable with any BSR set generating parser. In fact, BSR sets are as convenient as SPPFs for post-parse analysis, in particular for ambiguity resolution, derivation selection, abstract syntax tree construction, and post-parse semantic action evaluation, so it is likely that the SPPF extraction can be dispensed with entirely in many applications. In summary we contend that:

- Parsers that generate BSR sets are simpler to construct and require less runtime space than those that generate SPPFs, because there is no need for the additional graph construction machinery or to 'carry around' the left child which will be required when its parent is built.
- Disambiguation rules can be applied directly to a BSR set.
- A BSR set can be used as an oracle in the sense of Lee [18] during semantic evaluation, see Section 6.
- If an SPPF is required then a general 'library' SPPF extraction algorithm can be applied to the BSR set post-parse, see Section 3.2.

We begin with some initial definitions; a detailed, accessible introduction to generalised parsing can be found in [10]. Then to motivate the work we review Earley's recogniser and show that a minor addition is all that is required to turn it into a BSR set generating parser. We then discuss the extraction of derivation trees from BSR sets, before developing the combinator parser and CNP BSR set generators. We conclude with a brief discussion of the relationship between BSR sets and parser oracles.

1 Binary subtree sets

A grammar, Γ, has a set $\mathbf{N}$ of nonterminals, a set $\mathbf{T}$ of terminals, a start symbol $S \in \mathbf{N}$ and a set of production rules, $X ::= \alpha$ where $X \in \mathbf{N}$ and $\alpha \in (\mathbf{N} \cup \mathbf{T})^*$. We use the standard Kleene notation $\mathcal{A}^*$ to denote the set of all strings over a set $\mathcal{A}$; the empty string is denoted by ϵ. Conventionally, we use uppercase letters for nonterminals, lower case for terminals, and Greek letters for strings in $(\mathbf{N} \cup \mathbf{T})^*$. We define a derivation step as $\gamma X \delta \Rightarrow \gamma\alpha\delta$, where $X ::= \alpha$, and denote by $\stackrel{*}{\Rightarrow}$ the reflexive transitive closure of $\Rightarrow$. We say

that γ is nullable if $\gamma \stackrel{*}{\Rightarrow} \epsilon$. We define a grammar slot to be an item of the form $X ::= \alpha \cdot \beta$, where $X ::= \alpha\beta$ is a production rule. Grammar slots will be used as labels in GLL-style parsers and in GLL descriptors and Earley items.

The language, $L(\Gamma)$ defined by Γ is the set of all $u \in \mathbf{T}^*$ such that $S \stackrel{*}{\Rightarrow} u$. A sequence $S \Rightarrow \beta_1 \Rightarrow \ldots \Rightarrow \beta_n \Rightarrow u$ is a *derivation* of u. The parsing problem for a grammar, Γ, is to find, for any input u, the set of all derivations of u.

Derivations are classically represented as rooted ordered trees, whose root node is labelled S, whose leaf nodes are labelled with a terminal or ϵ, and whose interior nodes are labelled with nonterminals such that X has children corresponding to some rule $X ::= \alpha$. This is a *derivation tree* for the string obtained when the leaf node labels are read in left to right order.

1.1 Indexed BDTs

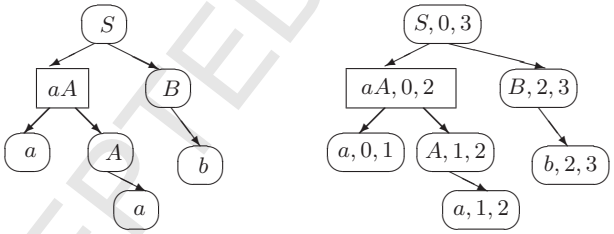
In general, there can be infinitely many derivation trees for a string u. Tomita [34] described the notion of an SPPF, a finite graph structure which embeds precisely the derivation trees, and a generalised LR-style parser that builds SPPFs. We shall describe a worst case cubic size version of the SPPF based on that described in [30]. However our SPPFs do differ slightly from [30], for example the labels of the intermediate nodes are different which can allow more sharing in certain cases.

An indexed binary derivation tree (BDT) is constructed from a derivation tree by first introducing intermediate nodes in the standard way so the tree is binarised from the left. The intermediate nodes are labelled with the concatenation of the labels of their children.

For example, for the grammar, Γ_1,

$$S ::= \ a\ A\ B \mid a\ A\ b \qquad A ::= a \mid c \mid \epsilon \qquad B ::= b \mid B\ c \mid \epsilon$$

a binary derivation tree for aab is on the left below and the corresponding indexed BDT is on the right, intermediate nodes are rectangular.



As noted above, the yield of the root node of the derivation tree is the input string u. In order to combine BDTs, their nodes are indexed with two integer *extents*, i and j, which are the left and right positions, in u, of the substring at their leaves. In detail, the leftmost leaf node has left extent 0. Any other leaf node has left extent equal to the right extent of the leaf node immediately to its left. For any leaf node (y, i, j), if $y = \epsilon$ then $j = i$ otherwise $j = i + 1$. For an internal node with exactly one child (x, i, j) the extents are (i, j). For an internal node with two children (γ, i, l) and (x, k, h) the extents are (i, h). Note, it is easy to show that in the latter case we will always have $l = k$. We refer to i and j as the left and right extents, respectively, of (x, i, j).

A BDT with offset d is the tree obtained from an indexed BDT by adding d to all the extents in all the node labels. For any indexed BDT for $a_1 \ldots a_n$, the subtree rooted at (δ, i, j) is an offset i indexed BDT for a derivation $\delta \stackrel{*}{\Rightarrow} a_{i+1} \ldots a_j$.

A grammar is ambiguous if there is a string which has two or more different indexed BDTs. A parsing technique is deterministic for a grammar Γ if at every step of the parser execution there is at most one grammar production that can be used. No parsing technique which constructs all derivations is deterministic with respect to an ambiguous grammar, and for any practical general parsing technique there are nonambiguous grammars for which initial parts of derivations that ultimately fail are constructed. For example, for the unambiguous grammar, Γ_2,

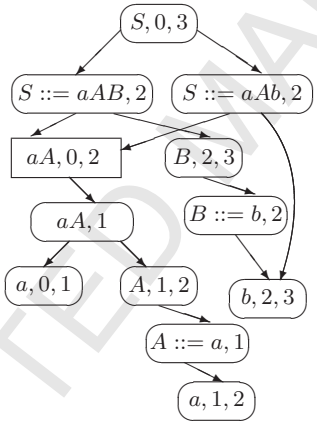
$$S ::= \ A\ C\ a\ B \mid A\ B\ a\ a \qquad A ::= a\ A \mid a \qquad B ::= b\ B \mid b \qquad C ::= b\ C \mid b$$

and input string *aabbaa*, a left to right, one symbol lookahead parser will construct a representation of the partial derivation (which ultimately fails)

$$S \Rightarrow ACaB \Rightarrow aACaB \Rightarrow aaCaB \Rightarrow aabCaB \Rightarrow aabbaB$$

as well as the full derivation of *aabbaa*. Thus, even for nonambiguous grammars, we need an efficient representation of the derivation structure being constructed by a parser.

A binarised SPPF is obtained from a set of indexed BDTs by merging (sharing) nodes with the same label and putting separate families of children under packed nodes. A packed node child, w, of a node (X, i, j) has label $(X ::= \beta x, k)$, a right child labelled (x, k, j) and either $k = i$ and $\beta = \epsilon$, or w also has a left child labelled (β, i, k). If w is the child of an intermediate node then it has label $(\beta x, k)$. For example, the SPPF constructed from both derivations of *aab* in Γ_1 is



We note that there are various forms of SPPF which are produced by different generalised parsers. The SPPF produced by many generalised LL and LR parsers have intermediate and packed nodes labelled with grammar slots rather than right hand side prefixes, and BRNGLR SPPFs are binarised to the right [31]. In the case of a rule of the form $X ::= BB\beta$ where B is nullable, it is possible that the SPPF is a multigraph because the intermediate node (BB, i, i) has left and right child (B, i, i). To avoid this, an additional intermediate node is often used, see [30]. In this paper we are mostly concerned with BSR sets which do not have any graph structure, so these subtle issues are not important.

1.2 Binary subtree representations

If a packed node, w, has children (β, i, k) and (x, k, j) then w has label $(\beta x, k)$ or $(X ::= \beta x, k)$, so we can compute the children, and the parent, of w if the label of w is modified to be $(\beta x, i, k, j)$ or $(X ::= \beta x, i, k, j)$, respectively.

We call a 4-tuple a *binary subtree representation* (BSR) over Γ if it is of the form
(i) $(X ::= \alpha, i, k, j)$, where $0 \leq i \leq k \leq j$, and $X ::= \alpha$ is a production in Γ, or
(ii) (β, i, k, j), where $0 \leq i \leq k \leq j$, $|\beta| > 1$ and for some X and δ, $X ::= \beta\delta$ is a production in Γ.

The *BSR set of an SPPF* is the set of elements (Ω, i, k, j), $i \leq k \leq j$, such that there is a packed node $w = (\Omega, k)$ and either w has two children with extents i, k and k, j or $i = k$ and w has just one child with extents k, j. We refer to such elements as extended packed nodes.

It is clear that we can reconstruct an SPPF from the BSR set of its extended packed nodes. The BSR set corresponding to the SPPF above is

$$\{(S ::= aAB, 0, 2, 3), (S ::= aAb, 0, 2, 3), (aA, 0, 1, 2), (B ::= b, 2, 2, 3), (A ::= a, 1, 1, 2)\}$$

However, as we have already mentioned, it is usually not necessary to have the SPPF at all. For example the standard priority and longest match strategies can be applied by selecting $(X ::= \alpha, i, k, j)$ rather than $(X ::= \gamma, i, l, j)$ if α has higer priority than γ or if $\alpha = \gamma$ and $k > l$.

In Section 3 we shall formally define what it means for a tree to be embedded in a BSR set and give an algorithm for extracting an SPPF if required. In Section 4 we show how to extend Johnson's memoised combinator parsers to generate BSR sets and in Section 5 we give a new, BSR set generating, generalised LL parser. But first we defend our claim that BSR generating parsers are easier to construct by revisiting a narrative that surrounds Earley's algorithm.

2 A BSR set generating Earley parser

Earley presented his algorithm as a recogniser, with a proposed method for turning it into a parser sketched at the end of the paper [8]. It is an indication of the subtlety of turning recognisers into parsers that this sketched method turns out to be incorrect for certain types of grammar. The error lies in the complexity of combining graphs of sub-derivations together in a way that does not create additional incorrect derivations.

This error was noted by Tomita [33] and motivated his creation of the generalised LR approach. However, Tomita's algorithm also contained a subtle error. In the case of grammars with hidden left recursion, that is derivations of the form $X \stackrel{*}{\Rightarrow} \beta X \alpha$ where $\beta \stackrel{*}{\Rightarrow} \epsilon$ and $\beta \neq \epsilon$, the derivation structure specified by Tomita's algorithm was infinite, and thus the parser never terminated. A faster Earley-style recogniser was described in [3], and a corresponding parser was outlined in [4]. As presented, that parser has the same derivation tree combination error as Earley's sketched parser. A correct SPPF-constructing version of Earley's algorithm was given in [27], where the issues that need to be addressed in order to ensure correctness are highlighted.

We give a simple BSR-constructing version of Earley's algorithm which just adds BSR elements as the algorithm proceeds. The algorithm inherits guaranteed termination and worst case cubic order from the original recogniser. Our purpose is to demonstrate that BSR set construction can be trivially added with no algorithm modification and for this reason we use the original predictor/completer version given in [8], with only minor modification to the predictor as described below. For further simplicity we will use the version with no lookahead; extension to the lookahead k version is identical, it is only the underlying Earley recogniser itself whose description is a little more complicated.

The core of Earley's algorithm is the construction of sets of states, the 'Earley sets'. In the original presentation the grammar productions are numbered and a state is a quadruple (p, j, f, α) where p is a production number, j is a integer position in the string on right hand side of production p, $\mathcal{S}_f$ is an Earley set and α is a lookahead string. We omit α and, to make the algorithm easy to read, we shall write the states as Earley items of the form $(X ::= \alpha \cdot \beta, f)$.

The Earley sets are constructed using three procedures: the predictor, the completer and the scanner. The implementation discussion in [8] specifies that the elements of $\mathcal{S}_j$ are stored in a list for processing (we denote this list by $\mathcal{R}_j$) and comments that the completer must be applied to any subsequently added states, we make these applications via a slight modification to the second part of the predictor. The only other change we make to turn the recogniser into a parser is to call the *bsrAdd*() function at the end of each of the three procedures which adds the BSR element arising from the action performed by that procedure. (This is the same function as is used in Section 5 below, illustrating the general reusability of BSR related functions.)

Predictor$((X ::= \alpha \cdot Y\beta, i), j)$:
  for all productions $Y ::= \gamma$ {
    if$(Y ::= \cdot\gamma, j) \notin \mathcal{S}_j$ {
      add $(Y ::= \cdot\gamma, j)$ to $\mathcal{S}_j$ and to $\mathcal{R}_j$ }
  for all $(Y ::= \delta\cdot, j) \in \mathcal{S}_j$ {
    if$(X ::= \alpha Y \cdot \beta, j) \notin \mathcal{S}_j$ {
      add $(X ::= \alpha Y \cdot \beta, j)$ to $\mathcal{S}_j$ and to $\mathcal{R}_j$
      bsrAdd$(X ::= \alpha Y \cdot \beta, i, j, j)$ } } }

Completer$((X ::= \alpha\cdot, i), j)$:
  for all $(Y ::= \delta \cdot X\mu, k) \in \mathcal{S}_i$ {
    if$(Y ::= \delta X \cdot \mu, k) \notin \mathcal{S}_j$ {
      add $(Y ::= \delta X \cdot \mu, k)$ to $\mathcal{S}_j$ and to $\mathcal{R}_j$
      bsrAdd$(X ::= \alpha\cdot, i, k, j)$ } }

Scanner$((X ::= \alpha \cdot b\beta, i), j)$:
  if$(X ::= \alpha b \cdot \beta, i) \notin \mathcal{S}_{j+1}$ {
      add $(X ::= \alpha b \cdot \beta, i)$ to $\mathcal{S}_{j+1}$ and to $\mathcal{R}_{j+1}$
      bsrAdd$(X ::= \alpha b \cdot \beta, i, j, j+1)$ }

bsrAdd$((X ::= \alpha \cdot \beta, i, k, j)$ {
    if$(\beta = \epsilon)$\{ insert $(X ::= \alpha, i, k, j)$ into Υ }
    else if$(|\alpha| > 1)$\{ insert (α, i, k, j) into Υ \}\}

Given an input string $a_1 \ldots a_n$ the functions are called on the elements of each $\mathcal{R}_j$, $0 \leq j \leq n$, starting with *Predictor*$((S' ::= \cdot S, 0), 0)$ where $S' ::= S$ is an augmented start rule. Initialising the global BSR set Υ to be the empty set and calling the above modified versions of the procedures turns the original Earley recogniser directly into a BSR set constructing parser.

3 Embedded trees and SPPF extraction

For BSR parsers we need to know how an indexed binary derivation tree relates/belongs to the BSR set constructed, and we want the definition to be independent of the notion of an SPPF.

In this section we give a formal definition of what it means for a tree to be embedded in a BSR set. Because parsers typically construct partial derivations which apply only to a prefix of the input, the BSR sets constructed can embed partial trees which do not extend to full derivation trees. We shall say that an embedded tree is core in a BSR set Υ if its root node is labelled $(S, 0, n)$, where n is the largest right extent of all the elements in Υ. We require BSR parsers to generate sets whose core embedded trees are precisely the BDTs of the input string, subject to any given disambiguation specification, and we give an SPPF extraction function which builds the SPPF of all the core trees embedded in a given BSR set.

3.1 Embedded trees

Let $\mathbf{W} = \mathbf{N} \cup \mathbf{T}$ denote the union of the nonterminal and terminal sets of a grammar Γ. We say that a BSR set over Γ, Υ, *embeds* an indexed rooted tree as follows:
1. If $(X ::= \epsilon, j, j, j) \in \Upsilon$ then Υ embeds the tree with a single node (ϵ, j, j).
2. For $(X ::= \beta x, i, k, j) \in \Upsilon$ or $(\beta x, i, k, j) \in \Upsilon$, where $x \in \mathbf{W}$, if $x \in \mathbf{T}$ then Υ embeds the tree with a single node (x, k, j), and if $\beta \in \mathbf{T}$ then Υ embeds the tree with a single node (β, i, k).
3. If $(X ::= x, i, i, j) \in \Upsilon$, $x \in \mathbf{W}$, and if Υ embeds a tree with root q labelled (x, i, j) then Υ embeds the tree obtained by adding a new root node (X, i, j) as the parent of q.
4. If $(X ::= \beta x, i, k, j) \in \Upsilon$, where $\beta \neq \epsilon$ and $x \in \mathbf{W}$, and if Υ embeds a tree with root q labelled (β, i, k) and a tree with root q' labelled (x, k, j) then Υ embeds the tree obtained by adding a new root node labelled (X, i, j) as the parent of q and q'.
5. If $(\beta x, i, k, j) \in \Upsilon$, where $\beta \neq \epsilon$ and $x \in \mathbf{W}$, and if Υ embeds a tree with root q labelled (β, i, k) and a tree with root q' labelled (x, k, j) then Υ embeds the tree obtained by adding a new root node labelled $(\beta x, i, j)$ as the parent of q and q'.

We call the largest integer, h, such that Υ contains an element of the form (Ω, l, k, h) the (right) extent of Υ. We say that a tree embedded in a BSR set, Υ, over Γ is Γ*-core* if its root node is labelled $(S, 0, n)$, where S is the start symbol of Γ and n is the extent of Υ. We define the Γ*-core* of Υ to be the smallest subset of Υ which embeds all the Γ-core trees embedded in Υ.

A general BSR parsing algorithm is defined to be an algorithm that constructs, for input u, a BSR set whose Γ-core trees are precisely the binarised derivation trees of u. An Earley BSR parser was given in Section 2 and below we shall describe a GLL-style BSR algorithm and a continuation passing combinator BSR algorithm.

One measure of the efficiency of a BSR parser is the number of elements in an output BSR set which do not belong to any core embedded tree, i.e the extent to which the BSR set is larger than its Γ-core. Such elements arise as a result of local nondeterminism. For example, for the grammar, Γ_2, above and input string *abaa* an Earley BSR parser with one symbol lookahead will generate the BSR set

$$\{(A ::= a, 0, 0, 1), (C ::= b, 1, 1, 2), (AC, 0, 1, 2), (ACa, 0, 2, 3),$$

$$(B ::= b, 1, 1, 2), (AB, 0, 1, 2), (ABa, 0, 2, 3), (S ::= ABaa, 0, 3, 4)\}$$

three elements of which, those containing an instance of C, do not belong to its Γ-core.

3.2 Extracting an SPPF

Although BSR sets are amenable to derivation extraction, disambiguation application and, as oracles see Section 6, to semantic action application, during the language design phase it may be helpful to have an SPPF output for visualisation and correctness checking purposes. It is also the case that SPPFs may be required anyway, for comparison with other existing general parsers or for post-parse use.

It is straightforward to extract an SPPF representation of all the Γ-core trees embedded in a given BSR set over Γ, and once implemented the extraction algorithm can be used with any BSR generating parser. The following algorithm generates the same style of SPPF as constructed by the parsers described in [30] and [26], so intermediate nodes have labels of the form $(X ::= \alpha \cdot \beta, i, j)$ rather than (α, i, j). But, for simplicity of presentation, we have not added the treatment of the special case for rules of the form $X ::= BB\beta$ where B is nullable. In such cases our extraction algorithm will thus produce a multigraph, although the additional clerical detail needed to avoid this is easy to add if required.

The algorithm takes as input a grammar Γ and a BSR set Υ over Γ, and produces as output an SPPF, $\mathcal{G}$, containing the Γ-core trees embedded in Υ.

We call a node (μ, i, j) in $\mathcal{G}$ *extendable* if μ is not ϵ or a terminal. Then $\mathcal{G}$ is constructed from the top down with child nodes being added, at each step, to an extendable leaf, w. The set Υ is searched to find all possible sets of children, and each set of children is added to $\mathcal{G}$ under a packed node child of w. The function $mkPN()$ makes the packed node and identifies its children, the function $mkN()$ makes the child nodes if necessary, and attaches them to the packed node.

If the input Υ has the property that its elements are BSR elements over a grammar Γ and its Γ-core embedded trees are precisely the BDTs for a string u, then $\mathcal{G}$ will be an SPPF for u with respect to Γ.

```
extractSPPF(Υ, Γ) {
 G := empty graph
  let S be the start symbol of Γ
  let n be the extent of Υ
  if Υ has an element of the form (S ::= α, 0, k, n) {
    create a node labelled (S, 0, n) in G
    while G has an extendable leaf node {
      let w = (μ, i, j) be an extendable leaf node of G
      if (μ is a nonterminal X in Γ) {
        for each (X ::= γ, i, k, j) ∈ Υ { mkPN(X ::= γ·, i, k, j, G) } }
      else {
        suppose μ is X ::= α · δ
        if (|α| = 1) mkPN(X ::= α · δ, i, i, j, G)
        else  for each (α, i, k, j) ∈ Υ { mkPN(X ::= α · δ, i, k, j, G) } } } }
  return G }
```

$mkPN(X ::= \alpha \cdot \delta, i, k, j, \mathcal{G})$ {
make a node y in $\mathcal{G}$ labelled $(X ::= \alpha \cdot \delta, k)$
if $(\alpha = \epsilon)$ $mkN(\epsilon, i, i, y, \mathcal{G})$
if ($\alpha = \beta x$, where $|x| = 1$) {
$mkN(x, k, j, y, \mathcal{G})$
if $(|\beta| = 1)$ $mkN(\beta, i, k, y, \mathcal{G})$
if $(|\beta| > 1)$ $mkN(X ::= \beta \cdot x\delta, i, k, y, \mathcal{G})$ } }

$mkN(\Omega, i, j, y, \mathcal{G})$ {
if there is not a node labelled (Ω, i, j) in $\mathcal{G}$ make one
add an edge from y to the node (Ω, i, j) } }

4 Combinator parsing

Conceptually, combinator parsing takes classical recursive descent and replaces the explicit grammar-corresponding lines of code with calls to generic functions. When deployed in a (higher-order) functional programming environment, it is easy to write such parsers using simple character matching functions which are combined using higher order function composition and union operations. Informally, a parser combinator is a function whose input includes zero or more parser functions and whose output is a parser function. For the purposes of this paper, a parser function takes as input an integer $j \geq 0$ and references a global \$-terminated input string held in an array $I = [a_1, \ldots, a_n, \$]$. (If pure functions are required I can be included as a parameter of the parser functions.) The output is a set, F, of integers such that $i \in F$ if and only if the parser has matched $a_{j+1} \ldots a_i$.

We can think of a recogniser R_Γ for a grammar Γ as a parser function which, on input 0, outputs a set F which contains n if and only if $a_1 \ldots a_n$ is a string in the language of Γ. We can build a recogniser from a grammar using functions defined for each terminal and nonterminal together with generic function composition and union combinators.

In this section we shall give an overview of combinator parsing, and show how general continuation-passing style parser combinators can be easily expanded to produce BSR sets. We will adopt an algorithm style rather than functional programming style presentation and avoid some of the clerical overhead that would be required to replace the use of global variables.

4.1 Simple combinators

We define *Ifn* to be the set of functions which take an integer and return a set of integers, $Ifn = \{f \mid f : \mathbb{N} \to \mathcal{P}(\mathbb{N})\}$. Simple recognisers are then elements of *Ifn*.

We could define a combinator *term* which takes a terminal symbol, b and an array I, and returns a terminal recogniser, $term(b, I)$.

$$term(b, I)(j) = \begin{cases} \{j+1\} & \text{if } I[j] = b \\ \emptyset & \text{otherwise} \end{cases}$$

However, the 'cases' approach to specifying a function in terms of its input and output can be cumbersome so where appropriate we shall adopt the more procedural programming style of having an algorithmic specification enclosed in braces. We will use the syntax $p\,;\,q$

to mean carry out the procedure or function call p and then the procedure or function call q. We prefer to focus on non-functional implementations and treat I as a global variable as it is only read by the recognisers. So we specify $tr(b)$, the term recogniser for b, by

$$(tr(b))(j) \;\; \{ \text{ if } (I[j] = b) \text{ return } \{j+1\} \text{ else return } \emptyset \; \}$$

Recognisers are built from term recognisers using binary combinators *seq* and *alt* which take two *Ifn* functions and return their composition and union, respectively.

$$seq(p,q)(j) = \cup\{q(i) \mid i \in p(j)\} \qquad alt(p,q)(j) = (p(j) \cup q(j))$$

(Here $\cup\{q(i) \mid ...\}$ denotes the union of the sets $q(i)$.) For example, for $I = [b, c, \$]$

$$alt(alt(seq(tr(b), tr(c)), tr(a)), tr(b))(0)$$

returns $\{1, 2\}$. Since the final output contains $2 = |I| - 1$, the string has been recognised.

To allow infinite languages, and to allow small specifications for large languages, we can give recogniser expressions names and use these names in other expressions. For example, for Γ_2

$$S ::= \; A\ C\ a\ B \mid A\ B\ a\ a \qquad A ::= a\ A \mid a \qquad B ::= b\ B \mid b \qquad C ::= b\ C \mid b$$

naming

$S \;:\; alt(seq(A, seq(C, seq(tr(a), B))), seq(A, seq(B, seq(tr(a), tr(a)))))$
$A \;:\; alt(seq(tr(a), A), tr(a))$
$B \;:\; alt(seq(tr(b), B), tr(b))$
$C \;:\; alt(seq(tr(b), C), tr(b))$

allows a call $alt(seq(seq(seq(A, C), tr(a)), B), seq(seq(seq(A, B), tr(a)), tr(a)))(0)$ to test whether I is a string in the language $L(\Gamma_2)$.

This is an easy way to build embedded recognisers in languages which have higher order functions of the form *seq* and *alt*, but it also has some well known limitations. If the recognisers are implemented as functions which call each other then the named expressions must not call themselves in a way that results in nontermination, that is the corresponding grammar must not be left recursive. If $alt(p, q)$ is implemented simply as first calling p then calling q and forming the union of the results, then repeated computations may be carried out. For example, for Γ_2 above the function A is called on the same value for both alternates of S. These issues can be addressed using memoisation, as we will discuss below.

4.2 Action execution

The discussion in Section 4.1 is about recognition only. In most cases it is required that some associated semantic action is carried out, and this is usually defined in terms of the structure of the input.

One of the attractions of combinator parsing is that it can provide embedded syntax analysis in a simple way; the syntax analysis functions (recogniser functions) can be written in the host language, the combinators can be provided as general library functions, and a user simply writes the combinator expressions they require. Syntax analysis can be enriched with actions to be carried out during the analysis; this is sometimes referred to

as top down translation. Combinator expressions have the useful recursive descent style property that actions, written in the host language, can be embedded in the expressions. Indeed, combinator parsers are sometimes thought of as applying actions rather performing syntax analysis.

An obvious approach is to parameterise the *tr*() function with the actions which are executed at the same time as the function executes. However, if the underlying grammar is not LL(1), then some actions may be performed that do not correspond to the input being analysed. For example, for the grammar Γ_2 above, if we add the action `print:'b from C'` to the occurrences of $tr(b)$ in combinator expression C then this action will be carried out even if the input is *abaa*.

We can insist, as is common for LR parsers, that embedded actions appear only at the end of alternates, but this does not solve the problem for all grammars. For example, for the grammar

$$S ::= T\ a\ c \qquad T ::= B \mid B\ a \qquad B ::= b$$

(which is unambiguous but not SLR-deterministic [2]) an action associated with the first alternate of T will be carried out even when the input is *baac*.

A more general approach is to collect the actions together in some structured form suitable for selective post-parse execution. The machinery required for producing structured output potentially can compromise the simplicity of the basic *tr*, *seq*, *alt* approach. The components of the structure correspond to portions of the input and, typically, parser functions will need to take and return sets of (structure object, string)-pairs. Using BSR sets avoids the structure complexity.

To some extent the difference between recognition of a syntactically valid input and the need to execute associated semantic actions is less frequently encountered in the combinator expression approach because of the lack of emphasis on a particular grammar. Many grammars can be rewritten to avoid nondeterminism, at least in places where semantic actions would be impacted. Classical parser generator tools are expected to generate a parser from a given grammar, but a writer of combinator expressions for a particular application can write them in any form that is convenient, effectively rewriting the (unspecified) implicit grammar. However, there are languages which can be specified by a non-ambiguous grammar but not by any deterministic grammar, so rewriting the grammar cannot work in general, even for unambiguous languages. Even in the case where a deterministic grammar does exist it may be very large and, even worse, not aligned to the intended semantics, making the required embedded actions difficult to write (and understand).

Of course, there are many ways of addressing the parsing problem for combinator expressions in specific cases, but we are interested in general techniques. Even in the case where semantic actions are functionally pure, that is without side effects, there needs to be disambiguation in the case of general grammars and this cannot be done safely on-the-fly (i.e. without possibly rejecting all derivations). The general parsing approach is to modify a syntax analyser so that it produces a representation of the syntactic structure of the input which can be disambiguated and then traversed by a post-parse semantic action evaluator. We have already described a simple extension which makes an Earley syntax analyser produce BSR sets. It is also easy to extend simple combinator recognisers to produce BSR sets. We shall do this, and then give a fully general BSR combinator parser based on Johnson's continuation passing approach [13].

4.2.1 Parser combinator BSR construction

BSRs are defined with respect to an underlying grammar whose nonterminals are, in this case, the combinator expression names $\mathbf{N}$, and whose terminals $\mathbf{T}$ are the symbols b such that these expressions contain $tr(b)$. Effectively, the sets $\mathbf{T}$ and $\mathbf{N}$ are user defined, and we require them to be disjoint to allow string matching. We let $\mathbf{W} = \mathbf{T} \cup \mathbf{N}$.

We modify the recogniser functions so that they take as parameters the elements (α, i, k, j) required for BSR set construction. The element j is the integer parameter from the recogniser version, and k, i are the other required BSR indices. Conceptually, the string $\alpha \in \mathbf{W}^*$ is the left prefix of an alternate that has been matched before the function is called, and the output is a set of elements of the form $(\alpha x, i, j, l)$, where the function has executed the derivation $x \stackrel{*}{\Rightarrow} a_{j+1}...a_l$. As a side effect the corresponding BSR elements are added to a global BSR set.

In detail, we define the set *Pfn* of parser functions to be functions that take a string in $\mathbf{W}^*$ and three integers, and return a set of such elements.

$$Pfn \ = \ \{f \mid f: \ (\mathbf{W}^* \times \mathbb{N} \times \mathbb{N} \times \mathbb{N}) \rightarrow \ \mathcal{P}(\mathbf{W}^* \times \mathbb{N} \times \mathbb{N} \times \mathbb{N}) \ \}$$

($\mathcal{P}(V)$ denotes the set of all subsets of a set V.) We will define certain *Pfn* functions in terms of a constant global array, I, of elements of $\mathbf{T}$ and allow them to add elements to a global BSR set Υ as a side effect.[1] To turn the recognisers into parsers we simply modify the combinator expressions to add elements to Υ.

For conciseness we specify the following procedures which insert (i.e. add if not already there) elements into the global BSR set, Υ.

$addUe(\alpha, i, k, j)$ { if $(|\alpha| > 1)$ insert (α, i, k, j) into Υ}

$addUr(X, \alpha, i, k, j)$ { insert $(X ::= \alpha, i, k, j)$ into Υ}

The procedure $addUe()$ adds the BSR elements that result from the matching of left-most prefixes of alternates, that is the elements that correspond to the intermediate, binarising, nodes of a binarised derivation tree. At the end of a call to nonterminal expression X, BSR elements of the form $(X ::= \alpha, i, k, j)$ are added by the procedure $addUr()$.

For a terminal symbol, b, specify the *Pfn* terminal parser function, $tp(b)$, by

$(tp(b))(\alpha, i, k, j)$ { $addUe(\alpha, i, k, j)$;
if $(I[j] = b)$ return $\{(\alpha b, i, j, j+1)\}$ else return $\emptyset$}

seq and *alt* are simply extended to elements of *Pfn*

$$seq(p, q)(\alpha, i, k, j) = \cup\{q(\beta, i, k', i') \mid (\beta, i, k', i') \in p(\alpha, i, k, j) \ \}$$

$$alt(p, q)(\alpha, i, k, j) = p(\alpha, i, k, j) \cup q(\alpha, i, k, j)$$

To allow for grammars that have ϵ productions we also define the identity *Pfn* function $eps(\alpha, i, k, j) = (\alpha, i, k, j)$.

Formally, the combinator expression named by $X \in \mathbf{N}$ is defined using a combinator nt which takes an element X of $\mathbf{N}$ and outputs a user specified *Ifn* function $nt(X)$. To invoke $addUr()$ in the correct positions we specify a new combinator, *prs*, which takes as input an element X of $\mathbf{N}$ and outputs $prs(X)$ which calls $addUr()$ on the results of $nt(X)$:

[1] It is easy to modify the parsers to return the BSR set if avoiding global variables is desired.

$prs(X)(\alpha, i, k, j)\{$
$\quad addUe(\alpha, i, k, j);$
$\quad$ let $F = \emptyset$
$\quad$ for $(\beta, j, k', i') \in nt(X)(\epsilon, j, j, j)\{$
$\quad\quad addUr(X ::= \beta, j, k', i');$
$\quad\quad$ add $(\alpha X, i, j, i')$ to F $\};$
$\quad$ return F $\}$

The intention is that the user defines functions $nt(X)$ constructed from *tp*, *prs* and *eps* using *seq* and *alt*. For example, given the definition of *nt* below, $prs(S)$ is a BSR parser for the grammar Γ_2

$$\begin{array}{lcl} nt(S) & = & alt(seq(prs(A), seq(prs(C), seq(tp(a), prs(B)))), \\ & & \quad seq(prs(A), seq(prs(B), seq(tp(a), tp(a))))) \\ nt(A) & = & alt(seq(tp(a), prs(A)), tp(a)) \\ nt(B) & = & alt(seq(tp(b), prs(B)), tp(b)) \\ nt(C) & = & alt(seq(tp(b), prs(C)), tp(b)) \end{array}$$

If the global variable I is set to $[a, b, a, a, \$]$ then $prs(S)(\epsilon, 0, 0, 0)$ returns the set $\{(S, 0, 0, 4)\}$ and creates the global BSR set

$\Upsilon = \{(A ::= a, 0, 0, 1), (C ::= b, 1, 1, 2), (AC, 0, 1, 2), (ACa, 0, 2, 3), (B ::= b, 1, 1, 2),$
$(AB, 0, 1, 2), (ABa, 0, 2, 3), (S ::= ABaa, 0, 3, 4), \}$

4.2.2 Factorised expressions

Formally BSR sets relate to a particular grammar whose production rules have the form $X ::= \alpha$ where α is a string of terminals and nonterminals. Combinator expressions naturally correspond to grammars whose production rules may be 'factorised'.

For example, the user can specify the same language as that specified by Γ_2 using the combinator expressions

$$\begin{array}{lcl} nt(S) & = & seq(prs(A), alt(seq(prs(C), tp(a)), seq(np(B), prs(B)))) \\ nt(A) & = & alt(seq(tp(a), prs(A)), tp(a)) \\ nt(B) & = & seq(tp(b), seq(alt(prs(B), eps), tp(a)) \\ nt(C) & = & seq(tp(b), seq(alt(prs(C), eps), tp(a)) \end{array}$$

The natural corresponding grammar has the (EBNF style) rules

$$S ::= A\,(C\,B \mid B\,a) \qquad A ::= a \mid a\,A \qquad B ::= b\,(B \mid \epsilon)\;a \qquad C ::= b\,(C \mid \epsilon)\;a$$

but the BSR set will be with respect to the 'multiplied out' version

$$S ::= A\,C\,B \mid A\,B\,a \qquad A ::= a \mid a\,A \qquad B ::= b\,B\,a \mid b\,a \qquad C ::= b\,C\,a \mid b\,a$$

Formally then, the BSR set generated by a combinator parser is with respect to the corresponding normal form expressions, which are defined as follows:
1. $tp(b)$ and *eps* are in normal form.
2. $seq(p, q)$ is in normal form if p is of the form $tp(b)$ or *eps*, and if q is in normal form and is of the form $tp(c)$, *eps* or $seq(q_1, q_2)$.

3. $alt(p, q)$ is in normal form if p and q are in normal form and if p is of the form eps, $tp(b)$, or $seq(p_1, p_2)$.

Effectively, the normal form is obtained by multiplying out brackets in the same way that turns factored grammar rules into the standard BNF form. The normal form is unique, and *seq* and *alt* are treated as right associative, which reduces the depth of nested calls, but left associativity could be specified if preferred.

4.3 Continuation passing BSR combinators

In various places in the literature the termination and repeated computation issues mentioned in Section 4.1 have been addressed, see Section 7. In this paper we show how the recognisers resulting from Johnson's solution to the left recursion problem [14] can be turned in to parsers which construct BSR sets.

Even for nonambiguous grammars, combinator parsers may call the same function from the same input position more than once, and thus efficiency may be improved by using memoisation [21, 22]. The first time a call $prs(X)(\alpha, i, k, j)$ is made for a particular X and j, the i and i' such that $(\beta, i, k', i') \in nt(X)(\epsilon, j, j, j)$ can be stored in a table indexed by X and j. Subsequent calls of the form $prs(X)(\gamma, i, h, j)$ can then just use the stored values i'.

If, in addition to storing results, a 'called' flag is used it could ensure termination for grammars with left recursion. However, to be correct it is necessary that the first call $nt(X)(\epsilon, j, j, j)$ computes all the values i' before a second call of the form $prs(X)(\gamma, i, h, j)$ is made. This is not always the case for left recursive grammars. For example, given

$$nt(S) = alt(tp(d), seq(prs(S), tp(a)))$$

the call $prs(S)(\epsilon, 0, 0, 0)$ with $I = [d, a, a, \$]$ will incorrectly fail to parse I.

To deal correctly with grammars which contain left recursion Johnson adopted a continuation passing approach [14]. We extend his, recogniser only, approach to BSR generating combinators. The core idea is to store in a table both the right extents of the results of calls $nt(A)(\epsilon, j, j, j)$ and elements of *Pfn* that apply updates to the table as a side effect.

In grammar terms, at the point where a combinator expression $prs(X)$ is evaluated on an element (α, i, k, j), there is a 'current' grammar rule $Y ::= \alpha X \delta$ which is being 'matched' and when $prs(X)$ returns, the matching will continue on the remaining part, δ, of the rule, from each input position, i', where $(\alpha X, i, j, i')$ is returned by $prs(X)$. So we store δ and i in position $TB[X, j]$ of a table TB, and define 'continuation' *Pfn* functions c_X, which access $TB[X, j]$ and evaluate δ (and any other continuations in $TB[X, j]$) on the returned input index (and any other returned input indices in $TB[X, j]$). Once this is complete, the continuations associated with Y need to be evaluated, so in fact (c_Y, δ, i) is stored in $TB[X, j]$.

Note, the BSR elements created by $prs(X)$ do not have to be recorded in the table because the only action performed is set insertion and there is no context associated with this. In this way the generation of a BSR set rather than a structured derivation representation reduces the complexity of the parser.

Formally then, as above we define

$$Pfn \;=\; \{f \mid f :\; (\mathbf{W}^* \times \mathbb{N} \times \mathbb{N} \times \mathbb{N}) \rightarrow\; \mathcal{P}(\mathbf{W}^* \times \mathbb{N} \times \mathbb{N} \times \mathbb{N})\;\}$$

The parser will construct a global memo table, TB, whose entries are indexed by combinator expression names, X, and integers j, where $0 \leq j \leq |I| - 1$. The entries contain two sets, one of continuation elements (g, δ, i) and one of integers, i'. So

$$TB[X, j] \ \subset \ \mathcal{P}(Pfn \times \mathbf{W}^* \times \mathbb{N}) \times \mathcal{P}(\mathbb{N})$$

and the entries of TB are initialised to $(\emptyset, \emptyset)$. We use the notation $TB[X, j].Pfns$ and $TB[X, j].ints$ to denote the first and second components of a table entry, so

$$TB[X, j] = (\, TB[X, j].Pfns,\, TB[X, j].ints)$$

The Pfn functions we define will update both a BSR set Υ and the table TB as side effects.

For each combinator expression name $X \in \mathbf{N}$ we define the Pfn function c_X by

$c_X(\alpha, i, k, j)$ {
  $addUr(X, \alpha, i, k, j)$;
  if($j \in TB[X, i].ints$) { return $\emptyset$ }
  else {
    add j to TB[X,i].ints ;
    return $\cup\{g(\delta X, l, i, j) \mid (g, \delta, l) \in TB[X, i].Pfns\}$ }

Rather than being parser functions, combinator expressions take an element of Pfn, apply it to perform the next step of the parse, and then return the function which is needed to complete the parse.

Thus we define combinators over the combinator expression functions

$$CF \ = \ \{f \mid f : \ Pfn \ \rightarrow \ Pfn\}$$

The terminal combinator tc takes as input a terminal symbol, b, and returns the CF function $tc(b)$ defined by

$(tc(b)(g))(\alpha, i, k, j)$\{
  $addUe(\alpha, i, k, j)$;
  if $(I[j] = b)$ return $g(\alpha b, i, j, j + 1)$ else return $\emptyset$ }

The continuation function is still the same after the action associated with $tc(b)$ is carried out, but the element it is called with has been updated to reflect the 'match' to b.

seq and alt are again binary composition and union combinators over CF. For $seq(p, q)$ the continuation after the action of p is q followed by the original continuation.

$$(seq(p, q)(g))(\alpha, i, k, j) = p(q(g))(\alpha, i, k, j)$$

$$(alt(p, q)(g))(\alpha, i, k, j) = (p(g))(\alpha, i, k, j) \cup (q(g))(\alpha, i, k, j)$$

The eps combinator is defined as the identity function $eps(g) = g$.

Again, we have a combinator nt which takes an element X of $\mathbf{N}$ and outputs a user specified function, $nt(X)$, from CF. To avoid repeated calls $(nt(X)(g))(\alpha, i, k, j)$ with the same g, i, and j, we define a memoising combinator, mem, which takes as input X. The output CF expression, $mem(X)$, takes as input a Pfn, g, and produces as output another Pfn. This function, $mem(X)(g)$, when called with (α, i, k, j), checks to see if $nt(X)(c_X)$ has already been called on (ϵ, j, j, j), by checking whether there are any elements

in $TB[X,j].Pfns$. If not then the call is made. Otherwise, the continuation (g,α,k) is added to $TB[X,j].Pfns$ and applied to all the results currently in $TB[X,j].ints$.

$(mem(X)(g))(\alpha,i,k,j)$ {
  $addUe(\alpha,i,k,j)$;
  if ($TB[X,j].Pfns = \emptyset$) {
    add (g,α,i) to $TB[X,j].Pfns$;
    return $(nt(X)(c_X))(\epsilon,j,j,j)$ }
  else if $((g,\alpha,i) \in TB[X,j].Pfns)$ { return $\emptyset$ }
  else { add (g,α,i) to $TB[X,j].Pfns$;
    return $\cup\{g(\alpha X,i,j,l) \mid l \in TB[X,j].ints\}$ } }

The user defined functions $nt(X)$ are constructed from *alt* and *seq* applied to *eps* and functions of the form $tc(b)$ and $mem(Y)$. For example, for Γ_2 we have

$$\begin{array}{rcl} nt(S) & = & alt(seq(mem(A), seq(mem(C), seq(tc(a), mem(B)))), \\ & & \quad seq(mem(A), seq(mem(B), seq(tc(a), tc(a))))) \\ nt(A) & = & alt(seq(tc(a), mem(A)), tc(a)) \\ nt(B) & = & alt(seq(tc(b), mem(B)), tc(b)) \\ nt(C) & = & alt(seq(tc(b), mem(C)), tc(b)) \end{array}$$

We define $g_0 \in Pfn$ as $g_0(\alpha,j,k,i) = \{(\alpha,j,k,i)\}$. Then a parser is called as

$$(mem(S)(g_0))(\epsilon,0,0,0).$$

Returning to the example grammar $S ::= d \mid S\ a$ we have

$$nt(S) \quad = \quad alt(\ tc(d),\ seq(mem(S),\ tc(a))\ )$$

and the call $(mem(S)(g_0))(\epsilon,0,0,0)$ with $I = [d,a,a,\$]$ returns the set

$$\{\ (S,0,0,1),\ (S,0,0,2),\ (S,0,0,3)\ \}$$

builds the memo table

TB	0	1	2	3
S	$(\{(g_0,\epsilon,0),(tc(a)(c_S),\epsilon,0)\},\{1,2,3\})$	$(\emptyset,\emptyset)$	$(\emptyset,\emptyset)$	$(\emptyset,\emptyset)$

and constructs the BSR set

$$\Upsilon \ = \ \{\ (S ::= d,0,0,1),\ (S ::= Sa,0,1,2),\ (S ::= Sa,0,2,3)\ \}$$

5 Clustered nonterminal BSR parsing

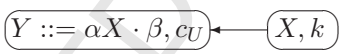
In this section we describe a new form of generalised LL parser which has a modified function call stack and whose extension to a parser is, as for Earley's algorithm above, obtained by simply adding calls to the BSR set constructing function *bsrAdd*(). In common with the basic GLL parsing approach, clustered nonterminal parsing (CNP) extends classical LL(1) recursive descent parsers to all context free grammars. The underlying idea is to directly handle the recursive descent function call stack in a way that allows

nondeterministic choices in a recursive parser for a non-LL(1) grammar to be recorded in 'process descriptors'. Each descriptor records the parser configuration at the point of the descriptor's creation. When the descriptor is 'processed' the parser configuration is reset and the parse continues as though from the point at which the descriptor was created. To enable this, lines of the parser are labelled with grammar slots. So for example, the first line of a section of code which attempts to match a rule $X ::= \alpha$ is labelled with the slot $X ::= \cdot\alpha$, and the line of code which is to be returned to after the rule has been matched has a label of the form $Y ::= \gamma X \cdot \delta$. (Technically there is a bijection between grammar slots and algorithm line labels but to avoid clerical overhead we shall suppress the distinction between a label and the corresponding slot.) The behaviour of such a parser is formally specified by code templates and a set of data structure building support functions, given in Section 5.3. A comparison with GLL is given in Section 5.5. First we give an informal description of the CNP approach.

5.1 CNP: generalised LL parsing

We assume that the input is held in an array $I = [a_0, \ldots, a_{n-1}, \$]$ where \$ is the end of string symbol. In recursive descent style, a clustered nonterminal parser has a section of code for each alternate of each nonterminal. At each point during a parse we are at a current grammar position, $Y ::= \gamma \cdot x\beta$ with a current input position, k. To allow nested call handling, the current return index, h, required when the call to Y is complete, will be recorded in a global variable c_U.

If x is a terminal then it is matched to a_k and if the match is successful the grammar and input positions are moved to $Y ::= \gamma x \cdot \beta$ and $k+1$. If $x = X$ is a nonterminal then the return position $Y ::= \gamma X \cdot \beta$, the current value in c_U and the current input index k are stored. To support efficient look-up the record is indexed by X and k and contains an entry $(Y ::= \gamma \cdot \beta, c_U)$. We think of this graphically



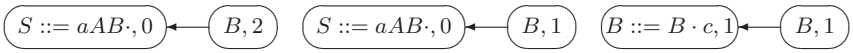
Then, for each rule $X ::= \mu$ such that a_k is in the select set (the lookahead guard set, see below) for $X ::= \mu$ a descriptor $(X ::= \cdot\mu, k, k)$ is created. The parser execution then continues by selecting a descriptor $(V ::= \tau \cdot \eta, l, j)$ and restarting the parse at grammar and input positions $V ::= \tau \cdot \eta$ and j, and with $c_U = l$.

If we are at the end of a rule $Y ::= \gamma\cdot$ then we have successfully matched γ to the input substring $a_{c_U} \ldots a_{k-1}$, and for each child $(Z ::= \tau Y \cdot \eta, l)$ of (Y, c_U) a descriptor $(Z ::= \tau Y \cdot \eta, l, k)$ is created so that execution from the corresponding positions can be continued. In general it is possible for a call to Y to match more than one substring, and for the same sub-parse from Y to belong to several derivations. For example, for the grammar Γ_1 above and input ac we have derivations

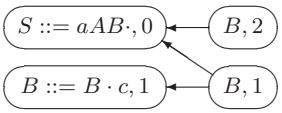
$$S \Rightarrow aAB \Rightarrow acB \Rightarrow ac$$

$$S \Rightarrow aAB \Rightarrow aB \Rightarrow aBc \Rightarrow ac$$

Then from grammar position $S ::= aA \cdot B$ we have a call to B from input positions 1 and 2. We also have a call to B from grammar position $B ::= \cdot Bc$ and input position 1. These generate the following record entries



To keep the size of the records worst case cubic we share nodes with the same label



We call the record structure a *Call Return Forest*, CRF, and refer to the index nodes (X, k) as *cluster nodes*.

The handling of the supporting data structures needed is separated into stand-alone, grammar independent, support functions which are called from the main parser. The processes of adding nodes to a CRF and creating descriptors from children of a cluster node are handled by the CNP support functions *call*() and *rtn*(), see below. Flow of control within the parser is handled using an outer descriptor selection loop together with algorithm line labels which correspond to function call return points. The outer loop selects a descriptor $(Y ::= \alpha \cdot \beta, h, k)$ and then effectively the parser recommences its recursive descent like execution at input position k and at the line of the code for $Y ::= \alpha\beta$ that corresponds to the first symbol of β. A simple example is given in Section 5.2.

Formally, CNP descriptors, (L, k, j), comprise a grammar slot, L, that forms a code label, an integer index of a CRF cluster node, and an input position. As descriptors are created they are stored in a set $\mathcal{R}$. In order to ensure that repeated computations are not performed, the set of all descriptors which have been created are stored in a set $\mathcal{U}$, and an element is only added to $\mathcal{R}$ if it is not already in $\mathcal{U}$. This is packaged into the support function *dscAdd*() defined in Section 5.3.1.

The parser initially adds $(S ::= \cdot\alpha, 0, 0)$ to $\mathcal{R}$ and $\mathcal{U}$ for each production $S ::= \alpha$ such that a_0 is in the select set for $S ::= \alpha$. The select set includes the set of terminals, b, such that $\alpha \stackrel{*}{\Rightarrow} b\beta$ for some β, and the select set test is performed by the support function *testSelect*() defined in Section 5.3.1.

On removing (L, k, j) the CNP steps through the code starting at line L, with additional BSR activity handled by the support function *bsrAdd*(), defined in Section 2. At a line corresponding to $(Y ::= \alpha \cdot b\beta, k, j)$ where b is a terminal, if $b = a_j$ and $|\alpha| \geq 1$ then the BSR $(\alpha b, k, j, j + 1)$ is added to the set being constructed.

At a line corresponding to $(Y ::= \alpha\cdot, k, j)$ the return positions are recovered from the CRF. This is handled by the support function *rtn*(). For each child $(Z ::= \mu Y \cdot \nu, i)$ of the node (Y, k), the descriptor $(Z ::= \mu Y \cdot \nu, i, j)$ is added. If $\nu = \epsilon$, the BSR $(Z ::= \mu Y, i, k, j)$ is also added. If $\nu \neq \epsilon$ and $\mu \neq \epsilon$, the BSR $(\mu Y, i, k, j)$ is added. There is a potential complication in that it is possible for additional children to be added to (Y, k) after a return action has been carried out, and there are cases where it is not possible to order the descriptor processing order to avoid this situation. To deal with it, the return action is recorded as a triple (Y, k, j) in the *contingent return* set $\mathcal{P}$. This return action is then applied to new children when they are created.

Finally, at a line corresponding to $(Y ::= \alpha X \cdot \beta, k, j)$ where X is a nonterminal, a CRF node $(Y ::= \gamma X \cdot \beta, k)$ is created with parent (X, j). If there is no node (X, j) then one is created and descriptors $(X ::= \cdot\gamma, j, j)$ are added to $\mathcal{R}$ and $\mathcal{U}$ for each production $X ::= \gamma$ whose select set contains a_j. This is done by the function *ntAdd*().[2] Linking

[2]The use of *ntAdd*() means that CNP makes fewer attempts to construct existing descriptors than RGLL.

the creation of descriptors associated with the start of a nonterminal X to the creation of the CRF node (X, j) in this way ensures that the descriptors are only created once. The creation of nodes in the CRF is handled by the support function *call*(). If the node (X, j) already exists then *call*() applies any return actions that have already been applied to (X, j) to the new child $(Y ::= \gamma X \cdot \beta, k)$.

CNP parsers are specified via a set of templates into which grammar symbols and production rules are substituted; these are given in Section 5.3. First we given an example.

5.2 Example

Before giving the formal CNP templates we give an example CNP parser for the simple grammar $S ::= d \mid S\ a$ from Section 4.3.

set m to be the length of the input string
read the input into I and set $I[m] := \$$
create CRF node $u_0 = (S, 0)$
$\mathcal{U} := \emptyset$; $\mathcal{R} := \emptyset$; $\mathcal{P} := \emptyset$; $\Upsilon := \emptyset$
ntAdd$(S, 0)$
while $\mathcal{R} \neq \emptyset$ {
    remove a descriptor, (L, k, j) say, from $\mathcal{R}$
    $c_U := k$; $c_I := j$; **goto** L
$S ::= \cdot d$:
    bsrAdd$(S ::= d\cdot, c_U, c_I, c_I + 1)$; $c_I := c_I + 1$
    if$(I[c_I] \in$ FOLLOW$(S))$ { *rtn*(S, c_U, c_I) }; **goto** L_0
$S ::= \cdot Sa$:
    call$(S ::= S \cdot a, c_U, c_I)$; **goto** L_0
$S ::= S \cdot a$:
    if$(testSelect(I[c_I], S, a)$ is false) **goto** L_0
    bsrAdd$(S ::= Sa\cdot, c_U, c_I, c_I + 1)$; $c_I := c_I + 1$
    if$(I[c_I] \in$ FOLLOW$(S))$ { *rtn*(S, c_U, c_I) }; **goto** L_0
L_0: }
if (for some α and l, $(S ::= \alpha, 0, l, m) \in \Upsilon$) { report success}
else {report failure}

5.3 CNP generator specification

Throughout this section the following notation is used.
m: a constant integer whose value is the length of the input
I: a constant array of size $m + 1$ containing the input string
c_I, c_U: integer variables
CRF: a digraph whose nodes are labelled (L, j)
    where L is either a nonterminal or a grammar slot
$\mathcal{P}$: set of CRF return actions represented as triples (X, k, j)
Υ: set of BSRs, $(X ::= \mu, i, k, j)$ and (μ, i, k, j)
$\mathcal{R}$: set of descriptors waiting to be processed
$\mathcal{U}$: set of all descriptors constructed so far

$: end-of-string character

The parser has two global variables c_U and c_I that hold the current CRF node index and input position, respectively, and a set of support functions. The functions *ntAdd*() and *testSelect*() have to be constructed for a given grammar Γ by the parser generator, and the latter makes use of the standard FIRST and FOLLOW sets [2] which must also be constructed by the parser generator. The other functions are grammar independent. The function *bsrAdd*() is specified in Section 2. All functions assume the existence of a global input array I, global sets Υ, $\mathcal{P}$, $\mathcal{U}$, $\mathcal{R}$, and a global CRF graph.

5.3.1 CNP support functions

ntAdd(X, j) {
  for all(grammar rules $X ::= \tau$)) {
    if(*testSelect*$(I[j], X, \tau)$) { *dscAdd*$(X ::= \cdot\tau, j, j)$ } }

testSelect(b, X, α) {
 if($b \in$ FIRST(α) or ($\epsilon \in$ FIRST(α) and $b \in$ FOLLOW(X))) {return true}
 else {return false} }

dscAdd(L, k, i) { **if** $(L, k, i) \notin \mathcal{U}$ {add (L, k, i) to $\mathcal{U}$ and $\mathcal{R}$}}

rtn(X, k, j) {
  if $((X, k, j) \notin \mathcal{P})$ {
    add (X, k, j) to $\mathcal{P}$
    for each child v of (X, k) in the CRF {
      let (L, i) be the label of v
      dscAdd(L, i, j); *bsrAdd*(L, i, k, j) } } }

call(L, i, j) {
  suppose that L is $Y ::= \alpha X \cdot \beta$
  if there is no CRF node labelled (L, i) create one
  let u be the CRF node labelled (L, i)
  if there is no CRF node labelled (X, j) {
    create a CRF node v labelled (X, j)
    create an edge from v to u
    ntAdd(X, j) }
  else { let v be the CRF node labelled (X, j)
    if there is not an edge from v to u {
      create an edge from v to u
      for all $((X, j, h) \in \mathcal{P})$ {
        dscAdd(L, i, h); *bsrAdd*(L, i, j, h) } } } }

5.3.2 The CNP templates

Now we give the code templates which specify the CNP parser. A parser is obtained by substituting the specific grammar production rules into the templates.

For each nonterminal X in the grammar there is a section of the algorithm, *code*(X), which will be defined below. In addition to the grammar slot labels, we require a label L_0

which labels the end of the controlling while loop, then the statement `goto` L_0 is equivalent to `break` in C-style programming languages.

When the descriptors have all been dealt with, the test for acceptance is made by checking for the existence of a BSR of the form $(S ::= \alpha, 0, l, m)$, for some α and l, where m is the length of the input string.

We suppose that the nonterminals of the grammar Γ are $A, \ldots, Z$, with start symbol S. Then the CNP parser for Γ is given by:

set m to be the length of the input string
read the input into I and set $I[m] := \$$
create CRF node $u_0 = (S, 0)$
$\mathcal{U} := \emptyset$; $\mathcal{R} := \emptyset$; $\mathcal{P} := \emptyset$; $\Upsilon := \emptyset$
$ntAdd(S, 0)$
while $\mathcal{R} \neq \emptyset$ {
    remove a descriptor, (L, k, j) say, from $\mathcal{R}$
    $c_U := k$; $c_I := j$; **goto** L
    $code(A)$
    ...
    $code(Z)$
L_0: }
if (for some α and l, $(S ::= \alpha, 0, l, m) \in \Upsilon$) { report success}
else {report failure}

We give the specification for $code(X)$ in terms of functions $code(X ::= \alpha \cdot \beta)$. We refer to the specifications of $code(X ::= \alpha \cdot \beta)$ as the *CNP templates*. Suppose that the grammar rule for X is $X ::= \tau_1 \mid \ldots \mid \tau_p$, we define:

$code(X)$ = $X ::= \cdot\tau_1$:
    $code(X ::= \cdot\tau_1)$
    if$(I[c_I] \in \text{FOLLOW}(X))$ { $rtn(X, c_U, c_I)$ }; **goto** L_0
    ...
$X ::= \cdot\tau_p$:
    $code(X ::= \cdot\tau_p)$
    if$(I[c_I] \in \text{FOLLOW}(X))$ { $rtn(X, c_U, c_I)$ }; **goto** L_0

Given a slot E we define $code(E)$ as follows, where a is any terminal and Y is any nonterminal, α and β are (possibly empty) strings of terminals and nonterminals, and L denotes the label corresponding to the slot $X ::= \alpha Y \cdot \beta$.

$code(X ::= \cdot)$ = $\Upsilon := \Upsilon \cup \{(X ::= \epsilon, c_I, c_I, c_I)\}$

$code(X ::= \alpha a \cdot \beta)$ =
    $bsrAdd(X ::= \alpha a \cdot \beta, c_U, c_I, c_I + 1)\}$; $c_I := c_I + 1$

$code(X ::= \alpha Y \cdot \beta)$ = $call(L, c_U, c_I)$; **goto** L_0
L :

$code(X ::= \cdot x_1 \ldots x_d)$ =
    $code(X ::= x_1 \cdot x_2 \ldots x_d)$
    if$(testSelect(I[c_I], X, x_2 \ldots x_d)\ is\ false)$ **goto** L_0

$code(X ::= x_1 x_2 \cdot x_3 \ldots x_d)$
$\ldots$
if$(testSelect(I[c_I], X, x_d)\ is\ false)$ **goto** L_0
$code(X ::= x_1 \ldots x_d \cdot)$

5.4 Example

For completeness we give the CNP parser generated from the above templates for the grammar Γ_2 used in Sections 1.1, 3.1, 4.1, 4.2.1 above.

$$S ::= A\ C\ a\ B \mid A\ B\ a\ a \qquad A ::= a\ A \mid a \qquad B ::= b\ B \mid b \qquad C ::= b\ C \mid b$$

set m to be the length of the input string
read the input into I and set $I[m] := \$$
create CRF node $u_0 = (S, 0)$
$\mathcal{U} := \emptyset$; $\mathcal{R} := \emptyset$; $\mathcal{P} := \emptyset$; $\Upsilon := \emptyset$
$ntAdd(S, 0)$
while $\mathcal{R} \neq \emptyset$ {
remove a descriptor, (L, k, j) say, from $\mathcal{R}$
$c_U := k$; $c_I := j$; **goto** L
$S ::= \cdot ACaB$:
$call(S ::= A \cdot CaB, c_U, c_I)$; **goto** L_0
$S ::= A \cdot CaB$:
if$(testSelect(I[c_I], S, CaB)$ is false) **goto** L_0
$call(S ::= AC \cdot aB, c_U, c_I)$; **goto** L_0
$S ::= AC \cdot aB$:
if$(testSelect(I[c_I], S, aB)$ is false) **goto** L_0
$bsrAdd(S ::= ACa \cdot B, c_U, c_I, c_I + 1)$; $c_I := c_I + 1$
if$(testSelect(I[c_I], S, B)$ is false) **goto** L_0
$call(S ::= ACaB \cdot, c_U, c_I)$; **goto** L_0
$S ::= ACaB \cdot$:
if$(I[c_I] \in$ FOLLOW$(S))$ { $rtn(S, c_U, c_I)$ }; **goto** L_0
$S ::= \cdot ABaa$:
$call(S ::= A \cdot Baa, c_U, c_I)$; **goto** L_0
$S ::= A \cdot Baa$:
if$(testSelect(I[c_I], S, Baa)$ is false) **goto** L_0
$call(S ::= AB \cdot aa, c_U, c_I)$; **goto** L_0
$S ::= AB \cdot aa$:
if$(testSelect(I[c_I], S, aa)$ is false) **goto** L_0
$bsrAdd(S ::= ACa \cdot a, c_U, c_I, c_I + 1)$; $c_I := c_I + 1$
if$(testSelect(I[c_I], S, a)$ is false) **goto** L_0
$bsrAdd(S ::= ACaa \cdot, c_U, c_I, c_I + 1)$; $c_I := c_I + 1$
if$(I[c_I] \in$ FOLLOW$(S))$ { $rtn(S, c_U, c_I)$ }; **goto** L_0

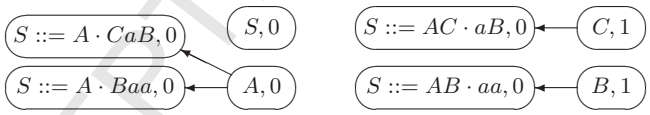
$A ::= \cdot aA$:
$bsrAdd(A ::= a \cdot A, c_U, c_I, c_I + 1)$; $c_I := c_I + 1$
if$(testSelect(I[c_I], A, A)$ is false) **goto** L_0

$call(A ::= aA\cdot, c_U, c_I)$; **goto** L_0
$A ::= aA\cdot$:
if$(I[c_I] \in \text{FOLLOW}(A))$ { $rtn(A, c_U, c_I)$ }; **goto** L_0
$A ::= \cdot a$:
$bsrAdd(A ::= a\cdot, c_U, c_I, c_I + 1)$; $c_I := c_I + 1$
if$(I[c_I] \in \text{FOLLOW}(A))$ { $rtn(A, c_U, c_I)$ }; **goto** L_0

$B ::= \cdot bB$:
$bsrAdd(B ::= b \cdot B, c_U, c_I, c_I + 1)$; $c_I := c_I + 1$
if$(testSelect(I[c_I], B, B)$ is false) **goto** L_0
$call(B ::= bB\cdot, c_U, c_I)$; **goto** L_0
$B ::= bB\cdot$:
if$(I[c_I] \in \text{FOLLOW}(B))$ { $rtn(B, c_U, c_I)$ }; **goto** L_0
$B ::= \cdot b$:
$bsrAdd(B ::= b\cdot, c_U, c_I, c_I + 1)$; $c_I := c_I + 1$
if$(I[c_I] \in \text{FOLLOW}(B))$ { $rtn(B, c_U, c_I)$ }; **goto** L_0

$C ::= \cdot bC$:
$bsrAdd(C ::= b \cdot C, c_U, c_I, c_I + 1)$; $c_I := c_I + 1$
if$(testSelect(I[c_I], C, C)$ is false) **goto** L_0
$call(C ::= bC\cdot, c_U, c_I)$; **goto** L_0
$C ::= bC\cdot$:
if$(I[c_I] \in \text{FOLLOW}(C))$ { $rtn(C, c_U, c_I)$ }; **goto** L_0
$C ::= \cdot b$:
$bsrAdd(C ::= b\cdot, c_U, c_I, c_I + 1)$; $c_I := c_I + 1$
if$(I[c_I] \in \text{FOLLOW}(C))$ { $rtn(C, c_U, c_I)$ }; **goto** L_0
L_0: }
if (for some α and l, $(S ::= \alpha, 0, l, m) \in \Upsilon$) { report success}
else {report failure}

Executing this algorithm with input *abaa* results in CRF



and descriptor, contigent return and BSR sets

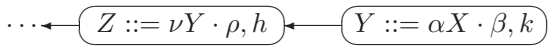
$\mathcal{U} = \{(S ::= \cdot ACaB, 0, 0),\ (S ::= \cdot ABaa, 0, 0),\ (A ::= \cdot aA, 0, 0),\ (A ::= \cdot a, 0, 0),$
$(S ::= A \cdot CaB, 0, 1),\ (S ::= B \cdot Baa, 0, 1),\ (C ::= \cdot bC, 1, 1),\ (C ::= \cdot b, 1, 1),$
$(B ::= \cdot bB, 1, 1),\ (B ::= \cdot b, 1, 1),\ (S ::= AC \cdot aB, 0, 2),\ (S ::= AB \cdot aa, 0, 2)\}$

$\mathcal{P} = \{(A, 0, 1),\ (C, 1, 2),\ (B, 1, 2),\ (S, 0, 4)\}$

$\Upsilon = \{(A ::= a, 0, 0, 1),\ (C ::= b, 1, 1, 2),\ (AC, 0, 1, 2),\ (B ::= b, 1, 1, 2),$
$(AB, 0, 1, 2),\ (ACa, 0, 2, 3),\ (ABa, 0, 2, 3),\ (S ::= ABaa, 0, 3, 4)\}$

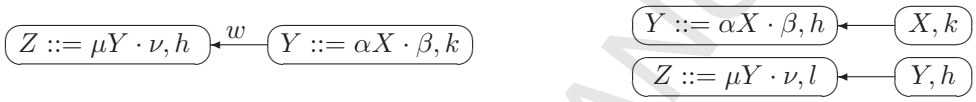
5.5 Comparison of CNP with GLL

Classical GLL parsers [30] follow the recursive descent style even more closely than CNP parsers. The general form of the algorithm, the line labels, and the descriptor based approach is as for CNP parsers. However, the nested return positions are held in a graph structured stack (GSS) rather than a CRF. A GSS node is of the form $(Y ::= \alpha X \cdot \beta, k)$ where k is the input position at the time the node was created, and the node is a parent of the node corresponding to the previous return position



Correspondingly, instead of holding the return input index for the previous call, c_U holds the current GSS node. The GSS merges multiple call chains that can arise from alternative derivation step choices, sharing nodes with the same label and using loops in the case of left recursion. Of course, the support functions that build the GSS are different but correspond to the CNP *call* and *rtn* functions.

In order to construct an SPPF, GLL descriptors contain an SPPF node and the edges of the GSS are labelled with an SPPF node. The GSS fragment on the left corresponds to the CRF fragment on the right



The idea of modifying the GLL GSS by introducing nodes of the form (X, i) was first proposed by Alex ten Brink whilst writing his MSc thesis [32]. The proposal, which was not published and was for a recogniser only, required nodes labelled (X, i) to be connected via edges labelled with the traditional GSS nodes, and did not allow for SPPF construction. The RGLL algorithm [26] is a parser which uses the traditional GSS but uses the notion of 'pop equivalence' to group together GSS nodes, allowing less information to be stored in the descriptors.

CNP has both a space and complexity advantage over classical GLL and RGLL. For GLL there are complex support functions that are needed to handle the SPPF construction, whereas CNP just uses the simple set function *bsrAdd*(). Furthermore, there is no need for the parser to add the children of each node it constructs and thus the corresponding information does not need to be recorded in the descriptor or the GSS.

For readers already familiar with GLL we remark that the functions *call*() and *rtn*() replace the functions *create*() and *pop*() in [30]. The function *call*() only attempts to add descriptors for the alternates of a nonterminal X the first time X is encountered for a given input position j. This makes *call*() slightly more complicated but more efficient than the corresponding *create*() function. The functions $getNodeT()$ and $getNodeP()$ from [30] that build the SPPF are replaced by the simple, and general, *bsrAdd*() function.

In the next section we give some experimental results comparing CNP, classical GLL and RGLL.

5.6 Experimental data

The primary purpose of this paper is to show that BSR set generation simplifies general parsing algorithms. However, to demonstrate the practicality of BSR sets and the space

efficiency of CNP over other GLL algorithms, in this section we report on comparisons between the CNP BSR algorithm defined in this paper and two SPPF generating GLL algorithms: 'classical' GLL (CGLL) as defined in [30] and 'reduced descriptor' GLL (RGLL) as defined in [26].

We first note that we have implemented a version of extractSPPF() which constructs SPPFs of exactly the same form as those in [26] (including the required handling of rules of the form $X ::= BB\beta$ where B is nullable) and thus we can directly compare the sizes of the BSR sets, the GLL SPPFs and the extracted Γ-core SPPFs.

Following the experimental section in [26], we show data for the grammars for ANSI-C (1989), ANSI-C++ (late 1997 draft), C# V1.2 (2002 ECMA standard) and Java (Java Language Specification 1). We also present timings, which indicate an advantage for BSR CNP over SPPF GLL.

The lexical parts of the grammars are replaced by keywords and an external lexer used to produce 'lexicalised' versions of test strings: for instance every identifier in the input string is replaced by the keyword `ID` and a production of the form `identifier ::= 'ID'` is added to each grammar; similarly for string, character, integer, real and Boolean literals.

For C and C++ our input strings comprise the complete source code for the RDP and GTB tool kits which are both written in ANSI-C; for C# we use ten copies of the complete source code of a Twitter client; and for Java, ten copies of an implementation of Conway's Game of Life. In addition, we use two copies of the source code for the ART support library (which is written in C++) for the ANSI C++ grammar.

Summary statistics are presented in Table 1. As mentioned in Section 5.5, in CNP descriptors are added by *call*(), resulting in fewer attempts to create descriptors that already exist. We demonstrate this by reporting the number of 'descriptor finds'. A 'find' is a data structure lookup that either returns a pre-existing object, or if not found then returns a newly created object. Hence, the descriptor find count for each algorithm is the total number of descriptor lookups; the descriptor count is the number of unique descriptors constructed. In RGLL, both descriptor counts and descriptor-find counts are significantly reduced over classical GLL; in CNP the descriptor counts match those of RGLL but the descriptor-find count is reduced further to little more than the number of descriptors.

The BSR set elements essentially correspond to the packed nodes of a classical SPPF, and that is borne out in the table. The slightly smaller BSR set sizes arise from the extra sharing enabled by the use of production prefixes in the BSR elements (see Section 1.1).

There is not a direct correspondence between GSS and CRF nodes. For instance, assuming that lookahead is used in the same way in both approaches, there is a CRF cluster node (X, j) if and only if there is a GSS node $(Z ::= \alpha X \cdot \beta, j)$. However, the set of indices j such that there is a GSS node $(Z ::= \alpha X \cdot \beta, j)$ may not be the same as the set of h such that there is a CRF leaf node $(Z ::= \alpha X \cdot \beta, h)$. The GSS/CRF figures show that, for these real world examples, the GSS has about the same number of nodes as the number of CRF leaf nodes, but the CRF has the additional cluster nodes.

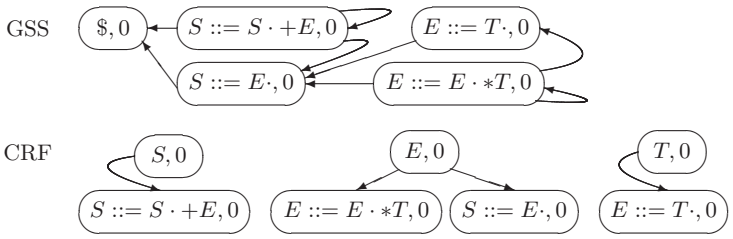
However, in general, the CRF can have far fewer edges than the GSS, and again this is borne out by our data. The reduction in edges arises from nonterminals that are called at the same input position but from different places in the grammar. A natural example is the standard left recursive expression grammar. For the following grammar and input

	Descriptor finds	Descriptors	Symbol nodes	Packed/BSR nodes	Cluster nodes	Nodes	Edges	CPU s
			SPPF/BSR			GSS/CRF		
ANSI C++ on art_support.tok (36,444 tokens)								
GLL	51,209,451	9,493,519	473,257	475,542		1,036,075	4,755,333	6.60
RGLL	9,418,913	1,894,009	473,257	475,542		1,036,075	4,755,333	2.94
CNP	1,894,473	1,894,009		453,248	445,138	1,032,048	1,037,632	1.26
ANSI C++ on gtb_src.tok (36,827 tokens)								
GLL	72,168,406	13,061,222	561,139	562,843		1,270,903	6,392,785	9.52
RGLL	12,893,820	2,366,346	561,139	562,843		1,270,903	6,392,785	3.88
CNP	2,366,528	2,366,346		539,822	523,510	1,265,359	1,271,297	1.6
ANSI C++ on rdp_full.tok (26,551 tokens)								
GLL	52,729,299	9,687,071	425,385	426,291		942,742	4,709,390	6.40
RGLL	9,559,741	1,771,700	425,385	426,291		942,742	4,709,390	2.67
CNP	1,771,798	1,771,700		407,681	386,641	938,788	942,966	1.15
ANSI C on gtb_src.tok (36,827 tokens)								
GLL	11,756,373	4,178,345	297,677	261,401		564,437	2,042,843	1.64
RGLL	4,177,163	1,127,572	297,677	261,401		564,437	2,042,843	1.14
CNP	1,127,804	1,127,572		254,191	230,746	559,723	564,730	0.67
ANSI C on rdp_full.tok (26,551 tokens)								
GLL	8,730,389	3,122,638	222,206	195,799		417,204	1,510,486	1.19
RGLL	3,122,083	844,344	222,206	195,799		417,204	1,510,486	0.91
CNP	844,413	844,344		191,255	169,278	413,650	417,341	0.47
C# 1.2 on twitter.tok (33,840 tokens)								
GLL	4,539,122	2,024,014	255,343	225,052		443,304	1,056,916	1.51
RGLL	2,016,754	837,254	255,343	225,052		443,304	1,056,916	1.25
CNP	837,254	837,254		221,272	234,131	438,652	443,303	0.56
Java JLS1 on life.tok (36,975 tokens)								
GLL	4,738,709	2,302,532	260,377	223,401		505,179	1,249,154	1.30
RGLL	2,302,532	911,430	260,377	223,401		505,179	1,249,154	1.09
CNP	911,430	911,430		211,176	249,028	503,105	505,178	0.62

Table 1: Performance on programming language grammars

a the GSS has eight edges and the CRF has four.

$$S ::= E \mid S + E \qquad E ::= T \mid E * T \qquad T ::= a \mid b$$



An extreme example of this effect is exhibited by the highly ambiguous grammar, Γ_3,

$$S ::= \ b \ \mid \ S \ S \ \mid \ S \ S \ S$$

Table 2 shows that, as one might expect, there is one CRF cluster node per input position and that the number of CRF edges is very significantly reduced compared to the number of GSS edges.

6 Parser oracles

We conclude this paper with a brief discussion of the use of the output of a parser in the subsequent semantic evaluation of a sentence. As we discussed above, it is common for the syntactic structure of a language to be used in the definition of its semantics, and thus for the semantic analysis phase of a translator to be based on the syntactic structure of the input. Indeed, Lang [17] writes

> In practice, the well formed strings of a language must be the projection of some syntactic structure, often a tree structure, to which meaning can be associated. ...
> Hence, it is natural to produce as the result of a parsing algorithm a single structure that represents all possible parses ... from which any specific parse can be extracted in time linear with the size of the extracted parse tree.

Recursive descent parsers are popular in this respect because semantic actions can be added to the parser itself, allowing semantic analysis to be carried out at the same time as the syntax analysis. However, efficient simple recursive descent parsers require an LL(1) syntactic structure.

Lee [18] discusses the idea that a parser should output an oracle: a parser for a grammar Γ takes an input string $u = a_0 \dots a_n$ and produces a Boolean valued function $\mathcal{F}_{\Gamma,u}$ such that

- $\mathcal{F}_{\Gamma,u}(X, i, j) = true$ if there are derivations $S \stackrel{*}{\Rightarrow} a_0 \dots a_{i-1} X a_j \dots a_n$ and $X \stackrel{*}{\Rightarrow} a_i \dots a_{j-1}$
- $\mathcal{F}_{\Gamma,u}(X, i, j) = false$ if there is no derivation $X \stackrel{*}{\Rightarrow} a_i \dots a_{j-1}$
- $\mathcal{F}_{\Gamma,u}(X, i, j)$ can take any value otherwise.

	String length	Descriptor finds	Descriptors	SPPF/BSR: Symbol nodes	SPPF/BSR: Packed/BSR nodes	GSS/CRF: Cluster nodes	GSS/CRF: Nodes	GSS/CRF: Edges
GLL	1	27	15	3	1		3	6
RGLL	1	15	5	3	1		3	6
CNP	1	5	5		1	1	2	2
GLL	5	740	288	31	55		22	144
RGLL	5	357	71	31	55		22	144
CNP	5	95	71		55	5	21	36
GLL	20	27,740	4,863	421	3,820		97	2,784
RGLL	20	9,597	1,031	421	3,820		97	2,784
CNP	20	4,280	1,031		3,820	20	96	591
GLL	30	85,240	11,038	931	13,080		147	6,419
RGLL	30	26,382	2,296	931	13,080		147	6,419
CNP	30	14,070	2,296		13,080	30	146	1,336
GLL	40	191,840	19,713	1,641	31,240		197	11,554
RGLL	40	55,167	4,061	1,641	31,240		197	11,554
CNP	40	32,960	4,061		31,240	40	196	2,381
GLL	50	362,540	30,888	2,551	61,300		247	18,189
RGLL	50	98,952	6,326	2,551	61,300		247	18,189
CNP	50	63,950	6,326		61,300	50	246	3,726
GLL	100	2,702,540	124,263	10,101	495,100		497	73,864
RGLL	100	647,877	25,151	10,101	495,100		497	73,864
CNP	100	505,400	25,151		495,100	100	496	14,951

Table 2: Performance on Γ_3

Here S is the start symbol of Γ and X is any nonterminal. Lee calls the function $\mathcal{F}_{\Gamma,u}$ an oracle for u.

For example, an SPPF for u can be seen as a oracle, the function $\mathcal{F}_{\Gamma,u}(X,i,j)$ returns *true* if there is a node labelled (X,i,j) in the SPPF, and false otherwise. The set of items produced by Earley's algorithm is also an oracle.

However, these two oracle types are not equivalent. A semantic evaluator can 'walk' an SPPF, starting at the root node, selecting packed nodes according to any specified strategy. All walks are guaranteed to correspond to a derivation, and provided the packed node selection strategy is efficient, and there are many efficient strategies, the time taken is proportional to the size of the selected derivation.

It is not straightforward to construct a derivation from the Earley sets. The problem is illustrate by the grammar

$$S ::= A\ B\ \mid C\ B \qquad A ::= b\ b \qquad B ::= b \mid b\ b \qquad C ::= b$$

and the string *bbb*. The sets of Earley items are

$$\mathcal{S}_0 = \{(S ::= \cdot AB, 0), (S ::= \cdot CB, 0), (A ::= \cdot bb, 0), (C ::= \cdot b, 0)\}$$

$$\mathcal{S}_1 = \{(A ::= b \cdot b, 0), (C ::= b\cdot, 0), (S ::= C \cdot B, 0), (B ::= \cdot bb, 1), (B ::= \cdot b, 1)\}$$

$$\mathcal{S}_2 = \{(B ::= b\cdot, 1), (B ::= b \cdot b, 1), (A ::= bb\cdot, 0), (S ::= CB\cdot, 0),$$
$$(S ::= A \cdot B, 0), (B ::= \cdot bb, 2), (B ::= \cdot b, 2)\}$$

$$\mathcal{S}_3 = \{(S ::= AB\cdot, 0), (S ::= CB\cdot, 0), (B ::= bb\cdot, 1), (B ::= b\cdot, 2), (B ::= b \cdot b, 2)\}$$

If we choose $(S ::= AB\cdot, 0)$ as the first derivation item then we know we need to select an item of the form $(B ::= \beta, k)$ from $\mathcal{S}_3$, suppose we choose $(B ::= bb\cdot, 1)$. We then need an item $(A ::= \alpha\cdot, 0)$ from $\mathcal{S}_1$, which does not exist. So we have to backtrack and choose $(B ::= b\cdot, 2)$ instead.

In general then, an oracle of the form defined by Lee may not be sufficient to allow a derivation to extracted efficiently in the sense of Lang, above.

Scott and Johnstone [29] modified the Earley algorithm to construct, on the fly in worst case cubic time, an SPPF which is an oracle that can be used to construct a derivation in time proportional to the size of the derivation. However, as we have discussed above, the need to carry around information required to build the SPPF can significantly complicate a parser in comparison to the corresponding recogniser.

The BSR set which corresponds to the packed nodes of an SPPF is also an oracle, and it is this set that is constructed by the Earley, combinator and CNP algorithms in this paper. A BSR set can be represented, for example, as an array indexed on i, j and Δ, where Δ is a grammar rule or a left prefix of the right hand side of a grammar rule. The (Δ, i, j) entry of the array is then an ordered list of k such that (α, i, k, j) belongs to the BSR set. With this representation a derivation can be obtained from the BSR set in time proportional to the size of the derivation, satisfying Lang's criterion, and disambiguation rules such as longest match and grammar rule priority can be efficiently applied.

It is possible, then, to produce a recursive descent semantics evaluator for a general grammar if a general parser is used to first to provide an oracle that satisfies Lang's criterion. This is a generalisation of the standard approach whereby a general parser produces an SPPF and then a recursive descent style semantic evaluator walks the SPPF.

An advantage of this general style is that disambiguation rules can be specified in the semantic evaluator, which can then use semantic as well as syntactic context.

This is the approach taken by Ridge [24], and suggested earlier in [19]. Ridge uses an Earley parser to generate an oracle which is then used to guide simple grammar combinators which execute specified semantic actions. Ridge's oracle is essentially a representation of the packed nodes of the SPPF constructed by Scott and Johnstone's Earley parser. In order to achieve worst case cubic complexity Ridge extracts an ϵ-free 2-form grammar which is used by his Earley algorithm. General BSR sets do not require this restriction, and Ridge's method could easily be modified to use a BSR set building Earley or CNP parser, see [35].

7 Conclusions and other related work

In this paper we have introduced the BSR set representation as a simpler alternative to structured representations such as SPPFs, we have described a parser version of Johnson's combinator parser and a new GLL-style algorithm which construct such sets, and we have demonstrated the ease with which BSR constructing parsers can be constructed from recognisers using Earley's algorithm.

We are not implying that existing generalised parsers are overly complex. If graph-structured output is required then corresponding additional complexity is unavoidable, but by returning BSR sets this complexity is separated from the underlying parsing algorithm, making comprehension, implementation and comparison with other general algorithms simpler, and allowing the same SPPF extraction function to be reused with any algorithm. Furthermore, we contend that actually for many applications BSR sets can replace the more structurally complex SPPFs.

There is a correlation between CNP parsers and the combinator parsers we have described, specifically between the CNP CRF and contingent return set, $\mathcal{P}$, and the memoisation table TB. The set $TB[X,j].ints$ is essentially equivalent to the CNP set, $\mathcal{P}$, of elements of the form (X,i,j) where $j \in TB[X,j]$. The relationship between $TB[X,j].Ifns$ and the CRF is less direct, and as we are not making any formal use of the relationship we shall describe it without proof. The combinator $nt()$ can be defined to map X to any combinator expression, but because of the associative and distributive properties of seq and alt there is an equivalent expression which is in normal form, as defined above. The elements of $TB[X,j].Ifns$ are of the form $f(c_Y)$ where f is a combinator expression and the alternates of its normal form are right tails of alternates of $nt(X)$. The element $f(c_Y)$ is then closely related to a set of clustered nonterminal CRF elements of the form $((X,i),(Y ::= \alpha X \cdot \beta, j))$, where β is an alternate of f. It is this correlation which inspired our extension of Johnson's recognisers to BSR generating parsers.

We conclude by briefly noting other related work. In [26] there is a GLL style algorithm, RGLL, in which the descriptors contain integers rather than GSS nodes, which was the starting point for our CNP algorithm and which we have compared experimentally with CNP in Section 5.6. It is straightforward to modify the RGLL algorithm in [26] to output BSR sets using versions of the functions described in Section 5. However the modified algorithm will have at its heart the traditional GSS rather than a CRF. A GLL style parser that uses a modified GSS to reduce the number of descriptors created is described in [1] and, as recounted in [26], some of the initial (unpublished) ideas were proposed by

Alex ten Brink.

The first version of a binarised SPPF was introduced in [28]. However, as these graphs were constructed from a GLR parser which naturally produces rightmost derivations, the binarisation is from the right rather than the left. The resulting SPFF is, of course, still worst case cubic. We can easily modify the BSR definition to naturally embed right binarised trees instead of left binarised trees, allowing them to be constructed naturally by a GLR-style algorithm.

Memoised recognisers were introduced in [22] where the technique was applied to Earley's algorithm. In [12] there is a presentation of a continuation passing combinator style parser based on Johnson's approach. The parsers are worst case cubic order and construct binarised SPPFs. An alternative combinator approach which generates BSR sets is described, together with a Haskell implementation, in [35]. The construction of derivation trees by combinator parsers is discussed in [11], and continuation passing combinator parsers are discussed in detail in [19]. Combinator parsers that are fully general and build Tomita-style polynomial order derivation outputs are described in [9].

Devriese and Piessen's grammar combinators [6, 7] offer an alternative solution, within the purely functional programming domain, to the issues regarding (left-)recursion. Their approach makes recursion observable without compromising referential transparency. They also give a discussion on the general topic of observable sharing within embedded languages.

References

[1] Ali Afroozeh and Anastasia Izmaylova. Faster, practical GLL parsing. In *CC 2015*, volume 9031 of *Lecture Notes in Computer Science*, pages 89–108, Berlin Heidelberg, 2015. Springer-Verlag.

[2] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: principles, techniques, and tools*. Addison-Wesley, 1986.

[3] John Aycock and Nigel Horspool. Faster generalised LR parsing. In *Compiler Construction, 8th Intnl. Conf, CC'99*, volume 1575 of *Lecture Notes in Computer Science*, pages 32 – 46. Springer-Verlag, 1999.

[4] John Aycock, R. Nigel Horspool, Jan Janousek, and Borivo Melichar. Even faster generalised LR parsing. *Acta Informatica*, 37(8):633–651, 2001.

[5] Sylvie Billot and Bernard Lang. The structure of shared forests in ambiguous parsing. In *Proceedings of the 27th conference on Association for Computational Linguistics*, pages 143–151. Association for Computational Linguistics, 1989.

[6] D. Devriese and F. Piessens. Explicitly recursive grammar combinators. In *Rocha R., Launchbury J. (eds) Practical Aspects of Declarative Languages. PADL 2011*, volume 6539, pages 84–98, Berlin Heidelberg, 2011. Springer=Verlag.

[7] D. Devriese and F. Piessens. Finally tagless observable recursion for an abstract grammar model. *Journal of Functional Programming*, 22:757–796, 2012.

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

[9] Richard A. Frost, Rahmatullah Hafiz, and Paul Callaghan. Parser combinators for ambiguous left-recursive grammars. In *PADL 2008*, volume 4902 of *Lecture Notes in Computer Science*, pages 167–181, Berlin Heidelberg, 2008. Springer-Verlag.

[10] Dick Grune and Ceriel J. H. Jacobs. *Parsing Techniques: A Practical Guide*. Monographs in Computer Science. Springer, Berlin, 2008.

[11] Graham Hutton and Erik Meijer. Monadic parser combinators. Technical Report NOTTCS-TR-96-4, Nottingham, 1996.

[12] Anastasia Izmaylova, Ali Afroozeh, and Tijs van der Storm. Practical, general parser combinators. In *Proceedings of the 2016 ACM SIGPLAN Workshop on Partial Evaluation and Program manipulation*, pages 1–12, New York, 2016. ACM.

[13] Mark Johnson. The computational complexity of GLR parsing. In Masaru Tomita, editor, *Generalized LR parsing*, pages 35–42. Kluwer Academic Publishers, The Netherlands, 1991.

[14] Mark Johnson. Memoization in top-down parsing. *Computational Linguistics*, 21:405–417, 1995.

[15] P. Klint, T. van der Storm, and J. Vinju. Rascal: A domain specific language for source code analysis and manipulation. In *Source Code Analysis and Manipulation*, pages 108–177. IEEE, 2009.

[16] Donald E Knuth. On the translation of languages from left to right. *Information and Control*, 8(6):607–639, 1965.

[17] Bernard Lang. Recognition can be harder than parsing. In *Computational Intelligence*, volume 10, pages 486–494, 1994.

[18] Lillian Lee. Fast context-free grammar parsing requires fast boolean matrix multiplication. *J. ACM*, 49:1–15, 2002.

[19] Peter Ljunglöf. *Pure Functional Parsing an advanced tutorial*. PhD thesis, Chalmers University of Technology and Göteborg University, 2002.

[20] Scott McPeak and George Necula. Elkhound: a fast, practical GLR parser generator. In Evelyn Duesterwald, editor, *Compiler Construction, 13th Intnl. Conf, CC'04*, Lecture Notes in Computer Science. Springer-Verlag, Berlin, 2004.

[21] D Miche. Memo functions and machine learning. *Nature*, 218:19–22, 1968.

[22] Peter Norvig. Techniques for automatic memoization with applications to context-free parsing. *Computational Linguistics*, 17:91–98, 1991.

[23] Terence Parr. *The Definitive ANTLR Reference: Building Domain-Specific Languages*. Pragmatic Programmers. Pragmatic Bookshelf, 2007.

[24] T. Ridge. Simple, efficient, sound and complete combinator parsing for all context-free grammars, using an oracle. In *SLE 2014*, volume 8706 of *Lecture Notes in Computer Science*, pages 261–281, Berlin Heidelberg, 2014. Springer-Verlag.

[25] Grigore Rosu and Traian Florin Serbanuta. A overview of the K semantic framework. *J.LAP*, 79:297–434, 2010.

[26] E. Scott and A. Johnstone. Structuring the GLL parsing algorithm for performance. *Science of Computer Programming*, 125:1–22, 2016.

[27] Elizabeth Scott. SPPF-style parsing from Earley recognisers. In Anthony Sloane and Adrian Johnstone, editors, *LDTA'07 7th Workshop on Language Descriptions, Tools and Applications*, also in Electronic Notes in Theoretical Computer Science, pages 53–67. Elsevier, 2007.

[28] Elizabeth Scott and Adrian Johnstone. Generalised bottom up parsers with reduced stack activity. *The Computer Journal*, 48(5):565–587, 2005.

[29] Elizabeth Scott and Adrian Johnstone. Recognition is not parsing – SPPF-style parsing from cubic recognisers. *Science of Computer Programming*, 75:55–70, 2010.

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

[31] Elizabeth Scott, Adrian Johnstone, and Giorgios Economopoulos. A cubic Tomita style GLR parsing algorithm. *Acta Informatica*, 44(6):427–461, 2007.

[32] A. P. ten Brink. *Disambiguation mechanisms and disambiguation strategies, Masters Thesis*. Eindhoven University of Technology, 2013.

[33] Masaru Tomita. *Efficient Parsing for Natural Language: A Fast Algorithm for Practical Systems*. PhD thesis, Carnegie Mellon University, Pittsburgh, PA, 1985.

[34] Masaru Tomita. *Efficient parsing for natural language*. Kluwer Academic Publishers, Boston, 1986.

[35] L. Thomas van Binsbergen, Elizabeth Scott, and Adrian Johnstone. GLL parsing with flexible combinators. In *Proceedings of the 11th ACM SIGPLAN International Conference on Software Language Engineering, SLE'18*, pages 16–28. ACM, 2018.

[36] M.G.J. van den Brand, J. Heering, P. Klint, and P.A. Olivier. Compiling language definitions: the ASF+SDF compiler. *ACM Transactions on Programming Languages and Systems*, 24(4):334–368, 2002.

[37] Eelco Visser. Program transformation with Stratego/XT: rules, strategies, tools, and systems in StrategoXT-0.9. In C.Lengauer et. al, editor, *Domain-Specific Program Generation*, volume 3016 of *Lecture Notes in Computer Science*, pages 216–238. Springer-Verlag, Berlin, June 2004.