# GLL Syntax Analysers For EBNF Grammars

Elizabeth Scott and Adrian Johnstone
Royal Holloway, University of London

**Abstract**

GLL is a worst-case cubic, recursive descent based parsing technique which can be applied to all BNF grammars without the need for grammar modification. However, EBNF grammars are often used, both for their compactness and their relative expressive simplicity. In this paper we give a formal specification for a parse tree representation of derivations which reflects the EBNF structure of the grammar, is worst case cubic size, and captures all derivations in the case of ambiguity. Particular care is needed in the case of closures with nullable bodies. We also describe an extension of GLL which directly supports the EBNF constructs. The resulting parsers are worst case cubic and follow the structure of the specifying EBNF grammar, making the parser behaviour easy to reason about. The parsers exploit the efficiency of factorisation and the use of iteration rather than recursion, retaining the structure of the specification in the presence of embedded semantic actions.

Since the 1960s computer language syntax has been conventionally specified using context free grammars (CFG) and BNF, a simple and powerful notation which specifies a set of sentences via rules which are used to rewrite strings. Using this formalism it is easy to generate (construct) sentences in the specified language but harder to decide, given a string, whether it is in the language. Deciding whether a given string $u$ belongs to a language is called the *language recognition problem*. For language translators the semantics of a sentence are also required and for this it is helpful to know the structure of $u$, that is the rewrite steps which are applied to generate $u$. The problem of discovering all sequences of rewrite steps which generate $u$ is called the *parsing problem*.

An algorithm which solves the parsing problem for all CFGs and all input strings is called a *general* parsing algorithm. There is no known linear time general parsing (or recognition) algorithm. The best current general recognition algorithm has order $O(n^{2.37})$ [21, 9] and the current practical general parsing algorithms have at least cubic order. Parsing is well understood in the sense that there do exist efficient linear-time parsing algorithms for subclasses of grammars. The two best known are the LR-table based approach which can be applied to LR(k) grammars and the recursive descent technique which can be applied to LL(k) grammars. However, the precomputed lookahead sets for values of $k > 1$ are large, and the basic recursive descent technique cannot be directly applied to grammars with left recursion. Despite the left recursion limitation, recursive descent remains attractive because of the close correlation between the parser and the grammar.

In order to get close to deterministic performance on near-deterministic grammars, extensions of both the LR and recursive descent techniques have been developed [20, 13, 11, 15, 2, 3, 4, 18]. In particular, GLR algorithms extend the LR technique by efficiently exploring all parse choices, and these have seen broad application in tools such as ASF+SDF [22], Elkhound [10] and Stratego/Spoofax [23]. There are also backtracking and extended lookahead variants of recursive descent which can be made fully general if combined with left recursion removal. GLL [19] is a fully general worst-case cubic extension of recursive descent which can be applied to any grammar without any need for grammar modification. GLL parsers are constructed from parsing templates that handle individual BNF constructs in a way that recalls standard recursive descent techniques. The construction is sufficiently simple that parsers may be written by hand. GLL-inspired parsing underpins the Rascal meta-language [8] (the successor to the GLR-based ASF+SDF toolkit) and is currently being incorporated into the K environment [16]; its intuitive nature makes it more attractive than the generalised LR alternative.

Extended forms of BNF, such as that described by Wirth [24], employ grammar rules whose right hand sides are regular expressions over the grammar symbols. EBNF grammars are often more compact; the ability to factorise the grammar and the use of Kleene closure can make the grammar simpler to write and maintain. The support for regular expressions is particularly important for so called 'character level' grammars in which the lexical level syntax is specified as part of the phrase level grammar. The attractiveness of EBNF is reflected in the fact that LR style parser generators, whose underlying LR tables do not support EBNF, often permit EBNF syntax by transforming to BNF before building the parser. However, grammar transformation has several disadvantages, in particular the extra nonterminals required to replace bracketing constructs create additional stack activity in the parser, iteration has to be transformed into either left or right recursion, which can impact the semantics of operators, and output structures such as derivation trees need to be converted to match the structure of the original grammar. If semantic actions are incorporated into the grammar in the top down translation style, then the position of these actions can be changed when a grammar is converted from EBNF to BNF. Furthermore, direct implementation of EBNF can result in more efficient parsers since iteration can be used in place of recursion, and since grammar factoring permits shared parsing activity.

Recursive descent style parsers naturally support direct implementation of EBNF grammars and in this paper we show how GLL parsers can inherit this advantage. We (i) give a formal specification for an SPPF-style parse tree representation which directly reflects the EBNF structure of the grammar, is worst case cubic size, and captures all derivations in the case of ambiguity, and (ii) describe an EBNF extension of the GLL algorithm which produces these derivation representations.

Informally, a BNF grammar is a set of rewrite rules $X ::= \alpha$ whose left hand sides are variables (called nonterminals) and whose right hand sides are strings of nonterminals and constants (called terminals). Sentences are derived by rewriting the nominated start nonterminal until a string of terminals is obtained. An EBNF grammar is a grammar whose rewrite rules have regular expressions on their right hand sides, and an EBNF rule $X ::= \tau$ corresponds to the (potentially infinite) set of BNF rules whose right hand sides are the strings in $\tau$.

Regular expressions are widely used and intuitively well understood. The structure of a regular expression is easily defined inductively in terms of operators such as concatenation, alternation, and Kleene closure. However, the string replacement definition of a derivation hides the structure of the regular expressions. A rewrite step using an EBNF rule $X ::= (a\ Y \mid b)a$ simply has the form $\mu X \nu \Rightarrow \mu a Y a \nu$. For many applications, the regular expression structure itself is significant. For example, pattern matching searches may need to report which portions of the input string matched a certain subexpression of the regular expression. So the representation of derivations needs to be extended to reflect the construction of a string from a regular expression. Furthermore, the definition of ambiguity for BNF grammars is in terms of unique derivations. This does not extend directly to EBNF grammars because it does not incorporate the ambiguity of a regular expression. For example, $X ::= a\ (a \mid a\ b)(c \mid b\ c)$ is ambiguous as there are two ways of constructing $aabc$, but there is only one derivation of any string in the language.

Representing the set of derivations requires care because, if the grammar contains cycles or if the body of a closure expression is nullable (derives the empty string), there may be infinitely many derivations. For example, denoting the Kleene closure by $\{\}$, we have that for $S ::= \{A\}a$, $A ::= bb|\epsilon$, $S$ may derive $bba$ with arbitrarily many matches of $A$ to $\epsilon$. Even in non-infinite cases the number of derivations can be of unbounded polynomial order. To achieve worst case cubic order, the derivations are represented in the form of a binarised shared packed parse forest (SPPF). The existing SPPF definition is only for BNF grammars, and extending it to EBNF is not straightforward. As we shall see, simply mimicking the structure of a corresponding BNF grammar derivation does not work correctly in subtle cases.

The main contributions of this paper are the provision of a formal definition of SPPF derivation representations for general EBNF grammars, with an SPPF-based definition of grammar ambiguity, and a formal version of the GLL algorithm which directly implements EBNF, using control flow that mimics the factorisation and iteration constructs in the grammar and generating a (worst case cubic size) SPPF.

EBNF GLL parsers are defined inductively over the structure of regular expressions and thus we require, and give, a grammar based definition of regular expressions which is not ambiguous. Even definitions of familiar structures such as FIRST and FOLLOW sets require care, particularly for boundary cases such as closures with nullable bodies. Our systemisation of EBNF grammar specifications and our definitions of corresponding SPPF derivation representations are independent of the GLL algorithm and can be applied to any parsing technology and to regular expression pattern matching.

The paper is structured as follows. We begin with a short discussion on the generalisation of recursive descent, and then give a formal definition of an EBNF grammar via an unambiguous BNF specification of regular expressions. We give formal definitions of FIRST and FOLLOW sets for EBNF grammars and establish the definitions needed for the GLL algorithm. In Section 3 we define the structure of derivation trees and SPPFs for EBNF grammars. In Section 4 we discuss the general behaviour of a GLL parser, together with the required data structures and the support functions which construct them. In Section 5 we give the formal specification for the construction of a GLL parser for an EBNF grammar. The primary thrust of this paper is theoretical, however to demonstrate practicality we have implemented an EBNF GLL parser for Java. In Section 6 we describe results comparing GLL parsers for the BNF and EBNF grammars given in the Java 2 specification document.

In the algorithm described in this paper, in order to focus on the basic approach we do not employ any LL(1) optimisation. However, it is straightforward to modify the algorithm to behave in a deterministic fashion on parts of a grammar which are LL(1).

## 1 Generalising recursive descent

The natural relationship between a recursive descent parser and its underlying grammar make it an attractive approach to parsing. There is a parse function associated with each grammar nonterminal and these functions call each other as the input string is read.

Within the parse functions there are branch statements, and for some grammars there can be more than one branch which is valid at the same position in the input string. This can result in incorrect results, and thus basic recursive descent parsers can only be used with LL(1) grammars [1].

Many techniques have been developed to broaden the applicability of recursive descent. In some cases it is possible to order the branches within the parse functions so that a particular choice is encountered first. If the choices are all valid, that is they all result in a successful parse of the input string, then this approach is valid. For example, it works correctly for the so-called if-then-else ambiguity which exists in many programming languages. The approach can be extended with the addition of backtracking if a particular choice fails, and parsing expression grammars (PEGs) [4] work in this way. For PEGs, precisely one derivation is returned and this is determined by the order in which the grammar rules are written. Simple combinator parsers [12] return a set of results from each parse function, thus allowing multiple branches within the functions to be explored.

In all cases, where multiple parsing choices are explored this needs to be done efficiently. Backtracking is potentially exponential, and both combinator parsers and PEGs have versions which use some form of memoisation to reduce this to polynomial order. The most serious difficulty for recursive descent is that in some cases, when the grammar is left recursive, a parse function can call itself on the same input and in this case the parser does not terminate. It is possible to rewrite the grammar so it is not left recursive, but as language semantics are often built on the grammar syntax, changing the grammar is, at the very least, inconvenient.

GLL [19] uses the graph structured stack approach developed by Tomita [20] to handle the call/return stack associated with the recursive descent parse functions. This allows all possible parse choices to be recorded in 'descriptors' which record the configuration of the parser at any point in the parse. These descriptors can then be processed in turn allowing all possible derivations to be computed. Effectively, all possible traces through the underlying recursive descent parser are executed. The beauty of the GLL approach to recursive descent is that it is naturally of worst case cubic order, even for grammars with left recursion, and it produces a worst case cubic size SPPF representation of all the derivations of its input string.

It is quite common for recursive descent based parsers to employ some form of EBNF. Combinator parsers usually permit embedded factorisation, easily supporting constructs such as $a\,(\,a\,b \mid c\,(\,d \mid c\,)\,)(\,a \mid b\,)$. Many tools use iteration to implement left recursive grammar constructs, and many more allow EBNF to be used in the grammar specification and then convert internally to BNF, [15, 7, 22, 5, 6, 14].

Currently, however, there is no direct GLL support for EBNF grammars. The main issue, and one which is shared with any fully general EBNF supporting parser technology, is the representation of the full set of derivations. None of the EBNF approaches mentioned above deal effectively with ambiguity within the regular expressions themselves, in particular with closure constructs whose bodies can derive the empty string. Representing such derivations in a way that also reflects the regular expression structure is the primary issued addressed in this paper. To generate these structures the GLL algorithm also needs significant extension, and we give this new specification.

## 2 EBNF grammars

It is relatively easy to write a GLL recogniser for an EBNF grammar, but parse tree generation is more subtle. For this we need to take care over the formal definition of an EBNF grammar. There any many compiler implementations based on EBNF grammars but no standard formal definitions of constructs such as FIRST and FOLLOW sets for the fully general case. In this section we develop the definitions we need.

Loosely, a BNF grammar is a set of production rules together with a nominated start nonterminal. For example, in the following productions $S$ is a nonterminal and $a$, $b$ are terminals.

$$S ::= a\ S \qquad\qquad S ::= b\ b\ b$$

Sentences are generated from a BNF grammar by starting with the start nonterminal and repeatedly replacing a nonterminal with the right hand side of one of its productions, until there are no nonterminals remaining. For example,

$$S \Rightarrow a\ S \Rightarrow a\ a\ S \Rightarrow a\ a\ b\ b\ b$$

It is conventional to group together all productions with the same left hand side and separate their right hand sides with the alternation symbol, |.

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

In an EBNF grammar the right hand sides of production rules are regular expressions over the terminals and nonterminals. The common definition of a regular expression gives an ambiguous structure, with operator associativity and priority being used to resolve conflicts. For example, $ab|c$ is defined to be $(ab)|c$, and $(ab)c$ and $a(bc)$ are equivalent. Since our definition of a regular expression will be used by the GLL parser generating templates, we want an unambiguous structure. Thus we begin with a formal unambiguous structural definition of a regular expression over a set $\mathcal{A}$.

### 2.1 Regular expressions

**Definition 1** *A* regular expression *over a set $\mathcal{A}$ is a set of strings of elements of $\mathcal{A}$ defined inductively as follows. (This definition provides disjoint base, closure, concatenation and alternation types allowing an unambiguous SLR(1) specification.)*

*For $b \in \mathcal{A}$, $b$ and $\epsilon$ are* base regular expressions *whose sets are $\{b\}$ and $\{\epsilon\}$ respectively.*

*If $\tau$ is a regular expression then $(\tau)$, $\{\tau\}$, $<\tau>$, and $[\tau]$ are* bracketed regular expressions. *The set $(\tau)$ is the same as the set $\tau$, $[\tau]$ is the set of $\tau$ together with $\epsilon$, $<\tau>$ is the set of all strings which are concatenations of one or more elements of $\tau$, and the set $\{\tau\}$ is the same as the set $[<\tau>]$. We may also refer to $[\tau]$ as an optional expression, to $<\tau>$ and $\{\tau\}$ as closure expressions and to $<\tau>$ as a positive closure expression. If $\tau$ is an alternated regular expression then we say that the bracketed expression is a* bracketed alternated regular expression.

*If $\tau_1, \ldots, \tau_n$, $n \geq 2$, are bracketed or base regular expressions then $\tau_1 \ldots \tau_n$ is a* concatenated regular expression*; its elements are the concatenations $\delta_1 \ldots \delta_n$ where $\delta_j \in \tau_j$, $1 \leq j \leq n$.*

*If $\tau_1, \ldots, \tau_n$, $n \geq 2$, are concatenated, bracketed or base regular expressions then $\tau_1 \mid \ldots \mid \tau_n$ is an* alternated regular expression*; its elements are the union of the sets $\tau_j$, $1 \leq j \leq n$.*

*We write $\delta \in \tau$ if $\delta$ is a string in the regular expression $\tau$.*

The following BNF grammar, in which terminals are quoted to distinguish them from BNF meta-symbols, generates the set of all regular expressions over the set $\{a_1, \ldots, a_k\}$ in a way that reflects the structure in Definition 1. The nonterminals *base*, *bracket*, *cat* and *alt* generate the base, bracketed, concatenated and alternated regular expressions, respectively.

```
reg ::= base | bracket | cat | alt
base ::= 'a1' | ... | 'ak' | '#'
bracket ::=  '(' reg ')' | '{' reg '}' | '[' reg ']' | '<' reg '>'
D ::= base | bracket
cat ::= D cat | D D
E ::= D | cat
alt ::= E '|' alt   |   E '|' E
```

This grammar is SLR(1) (see [1] for a discussion of the LR family of grammars) and hence is unambiguous. Thus the definition of the structure of a regular expression given in Definition 1 is unambiguous. The slightly unusual forcing of the sublanguages of each nonterminal to be disjoint is because there is a GLL template for regular expressions generated by each of `base`, `bracket`, `cat`, and `alt` and, since these sublanguages are disjoint, the straightforward substitution method for building parsers from the templates is deterministic and will define a unique parser for any given grammar.

It is convenient to write $\tau\psi$ where $\tau$ and $\psi$ are concatenated expressions, even though we cannot formally concatenate concatenations. For base, bracketed and concatenated expressions $\tau$, $\psi$ we shall write $\tau\psi$ for the concatenated expression formally obtained by forming the concatenation of all the base and bracketed expressions in $\tau$ and $\psi$.

A note on notation: perhaps the most common notation for regular expressions is operator-style, using `*` for closure and `?` for optional. Parentheses are used to override priorities, $(ab)*$ or $(a|b)?$. The GLL specification employs grammar positions, often called items in table driven parser specifications. A GLL EBNF parser will take different actions at positions immediately before and after opening brackets depending on whether the bracket is a closure or optional expression. It is helpful for the specification, and easier for the reader, to be able to determine the nature of an expression from its opening bracket. Thus in this paper we shall use the Wirth-style notation above, whose equivalence to the operator notation is shown in the following table.

| `[ ]` | `{ }` | `< >` |
|---|---|---|
| `( )?` | `( )*` | `( )+` |

Any concrete implementation may, of course, use the operator notation as long as the semantics of the implementation are correct under translation into the Wirth-style. (For example, $ab*$ should be treated as $a(b)*$ with two distinct grammar positions before and after the opening (, and $(a|b)*$ is different from $((a|b))*$.)

## 2.2 EBNF grammars and grammar slots

In recursive descent fashion, a GLL parser attempts to generate the given input string by starting with the start symbol and constructing a sequence of strings, at each step taking the left most nonterminal in the current string and replacing it with the right hand side of one of its productions. Thus a GLL parser has a section of code for each nonterminal, and such a section can be 'jumped' to from multiple places, corresponding to different instances of the corresponding nonterminal in the grammar. The return positions are labelled and prior to a jump the return label is put on a stack. Sections of parser code corresponding to alternation and closure expressions can also result in multiple control flow paths which need to be labelled to ensure all paths are followed. However, the 'return' position associated with a subexpression of a regular expression is unique to that subexpression and does not need to be held on the stack. Thus in our definitions of an EBNF grammar, we treat alternated regular expressions which are the right hand sides of grammar production rules slightly differently from other regular expressions.

**Definition 2** *An* EBNF grammar *consists of*
*a finite set* $\mathbf{T}$ *of terminals*
*a finite set* $\mathbf{N}$ *of nonterminals disjoint from* $\mathbf{T}$
*a start symbol* $S \in \mathbf{N}$
*a finite set of rules* $A ::= \tau_1 \mid \dots \mid \tau_t$*, at most one for each nonterminal* $A \in \mathbf{N}$*, where each* $\tau_k$*,* $1 \leq k \leq t$*, is a base, bracketed or concatenated regular expression over the set* $\mathbf{T} \cup \mathbf{N}$*. We refer to the* $\tau_k$ *as the* production alternates *or* productions *of* $A$*.* [1]

To define the language generated by an EBNF grammar we just extend the simple BNF replacement of a nonterminal by one of its production alternates to the case where the alternate is any regular expression. (The definition does not capture the structure of elements in the language, this is discussed in detail in Section 3.)

**Definition 3** *A* derivation step *is an expansion* $\alpha A\beta \Rightarrow \alpha\delta\beta$ *where* $\alpha, \beta \in (\mathbf{T} \cup \mathbf{N})^*$*,* $A ::= \tau$ *is a production and* $\delta \in \tau$*. A* derivation *of* $\rho$ *from* $\sigma$ *is a sequence* $\sigma \Rightarrow \beta_1 \Rightarrow \beta_2 \Rightarrow \dots \Rightarrow \beta_{n-1} \Rightarrow \rho$*, also written* $\sigma \overset{*}{\Rightarrow} \rho$ *or, if* $n > 0$*,* $\sigma \overset{+}{\Rightarrow} \rho$*. A derivation is* left-most *if at each step the left-most nonterminal is replaced.*

As we shall see in Section 4, the GLL algorithm relies on parser code line labels which correspond to positions in the grammar. These positions are also used to label the nodes of the EBNF derivation tree structure and are key to correctly representing bracketed regular expressions with nullable bodies.

**Definition 4** *A* grammar slot *is any position immediately before or immediately after a base or bracketed regular expression on the right hand side of a grammar rule.*

In the style of LR(0) items, we use a 'dot' to indicate a grammar slot: $A ::= \mu \cdot x\nu$ denotes the slot immediately before $x$.

In the parser specification we will need to identify particular instances of regular expressions by their position in the production rule. Thus it is convenient to use instance annotations to uniquely define each terminal, nonterminal and $\epsilon$ instance in the EBNF grammar productions. These instances, which are written as superscripts to the symbols, essentially correspond to slots.

[1]This definition only allows one grammar rule for each nonterminal so that there is only one place in a corresponding GLL parser where code associated with a given nonterminal is situated.

**Definition 5 (Symbol instances)** We call $x^{(j)}$ a symbol (nonterminal, terminal or $\epsilon$) *instance* and say that $x$ is the *underlying symbol* of the instance $x^{(j)}$. We call a regular expression, $r$, whose elements are symbol instances an *expression instance*, and we write $exp(r)$ for the underlying expression which has the instance superscripts removed.

We refer to an EBNF grammar in which all the symbols on the right hand sides of productions have be uniquely instanced as an *instanced grammar.* Any expression instance is uniquely defined by any of the symbol instances it contains, and has an associated nonterminal which is the left hand side of the production in which it occurs. We shall use an instanced grammar and its underlying non-instanced grammar interchangeably where no confusion results.

As with traditional recursive descent parsers, GLL parsers exploit FIRST and FOLLOW sets to restrict potential derivation choices. In the next two sections we define FIRST and FOLLOW sets for EBNF grammars.

## 2.3 FIRST sets for EBNF grammars

FIRST sets for EBNF grammars are a straightforward extension of the FIRST sets in BNF grammars. The right hand side of the rule for a nonterminal $A$ is a regular expression and effectively the FIRST set of $A$ is the union of the FIRST sets of all the strings belonging to the regular expression. However, the formal definition we give is recursive on the structure of the regular expression, and hence is constructive in nature.

**Definition 6** *For a nonterminal $A$, a terminal $b$ and regular expressions $\psi, \tau_1, \ldots, \tau_k$ we define* FIRST() *to be the smallest sets satisfying*

$\text{FIRST}(\epsilon) = \{\epsilon\}$ *and* $\text{FIRST}(b) = \{b\}$

$\text{FIRST}(A) = \text{FIRST}(\tau_1) \cup \ldots \cup \text{FIRST}(\tau_k)$, *where* $A ::= \tau_1 \mid \ldots \mid \tau_k$

$\text{FIRST}(\ (\psi)\ ) = \text{FIRST}(\psi)$

$\text{FIRST}(\ [\psi]\ ) = \text{FIRST}(\psi) \cup \{\epsilon\}$

$\text{FIRST}(\ \{\psi\}\ ) = \text{FIRST}(\psi) \cup \{\epsilon\}$

$\text{FIRST}(\ <\psi>\ ) = \text{FIRST}(\{\psi\}\psi) = \text{FIRST}(\psi)$

$$\text{FIRST}(\tau_1 \ldots \tau_k) = \begin{cases} \text{FIRST}_\epsilon(\tau_1) \cup \text{FIRST}(\tau_2 \ldots \tau_k) & \textit{if } \epsilon \in \text{FIRST}(\tau_1) \\ \text{FIRST}(\tau_1) & \textit{otherwise} \end{cases}$$

$\text{FIRST}(\tau_1 \mid \ldots \mid \tau_k) = \text{FIRST}(\tau_1) \cup \ldots \cup \text{FIRST}(\tau_k)$

*where* $\text{FIRST}_\epsilon(\tau) = \text{FIRST}(\tau) \backslash \{\epsilon\}$.

*For a regular expression instance, $r$, we define* $\text{FIRST}(r) = \text{FIRST}(exp(r))$.

## 2.4 FOLLOW sets for EBNF grammars

The FOLLOW set of a nonterminal, $X$, in a BNF grammar is the set of terminals which can occur after $X$ in some sentential form. There is an equivalent inductive definition in which $\text{FOLLOW}(X^{(j)})$ of an instance of $X$ in, say, the grammar rule $Y ::= \alpha X \beta$ is defined to be the union of the sets $\text{FIRST}(\beta\, x)$, for $x \in \text{FOLLOW}(Y)$, and then $\text{FOLLOW}(X)$ is defined to be the union of the FOLLOW sets of all instances $X^{(j)}$. To extend this definition to EBNF grammars we need to address the problem that for an instance $Y ::= \alpha X \beta$, $\beta$ may not be a regular expression. For example, in $Y ::= a\ (\ b \mid b\ [\ X\ b\ ]\ c\ )\ d$ how should $\text{FIRST}(\ b\ ]\ c\ )\ d)$ be defined? The solution is to define FOLLOW sets of nonterminal instances in terms of FIRST sets of Regular Expression slots.

**Definition 7** *An* RE slot *is any position immediately before or immediately after a base or bracketed expression in a regular expression or an instanced regular expression.*

*An RE slot has an* underlying regular expression *obtained by removing the 'dot'. Thus an RE slot is of the form $\xi \cdot \eta$ where $\xi\eta$ is a regular expression or in an instanced regular expression.*

Next we give the definition of FIRST$(h)$ where $h$ is an RE slot with underlying regular expression $\tau$. If $h$ is of the form $\cdot\tau$ or $\tau\cdot$ then FIRST$(h)$ is defined directly. Otherwise $h$ is of the form $\xi \cdot \eta$ and FIRST$(h)$ is defined inductively over the structure of $h$.

$\textsc{first}(\cdot\tau) = \textsc{first}(\tau)$
$\textsc{first}(\tau\cdot) = \{\epsilon\}$
$\textsc{first}((\xi \cdot \eta)) = \textsc{first}([\xi \cdot \eta]) = \textsc{first}(\xi \cdot \eta)$

$$\begin{aligned} \textsc{first}(\{\xi \cdot \eta\}) &= \textsc{first}(< \xi \cdot \eta >) \\ &= \begin{cases} \textsc{first}(\xi \cdot \eta) \cup \textsc{first}(\xi\eta) & \text{if } \epsilon \in \textsc{first}(\xi \cdot \eta) \\ \textsc{first}(\xi \cdot \eta) & \text{otherwise} \end{cases} \end{aligned}$$

$$\textsc{first}(\tau_1 \ldots \cdot h_i \tau_{i+1} \ldots \tau_n) = \begin{cases} \textsc{first}_\epsilon(h_i) \cup \textsc{first}(\tau_{i+1} \ldots \tau_n) & \text{if } \epsilon \in \textsc{first}(h_i) \\ \textsc{first}(h_i) & \text{otherwise} \end{cases}$$

$\textsc{first}(\tau_1 \mid \ldots \mid \tau_{i-1} \mid \cdot h_i \mid \ldots \mid \tau_n) = \textsc{first}(h_i)$
For a slot instance, $r_1 \cdot r_2$, we define $\textsc{first}(r_1 \cdot r_2) = \textsc{first}(exp(r_1) \cdot exp(r_2))$.

We can now define the 'instance' follow set associated with each nonterminal instance on the right hand sides of (instanced) EBNF grammar productions, and from these define the full FOLLOW sets.

**Definition 8** *For a nonterminal instance $X^{(j)}$ with instanced grammar slot $Y ::= \alpha X^{(j)} \cdot \beta$ we define* FOLLOW$(X^{(j)})$ *to be the smallest set such that*

$$\textsc{follow}(X^{(j)}) = \begin{cases} \textsc{first}_\epsilon(\alpha X^{(j)} \cdot \beta) \cup \textsc{follow}(Y) & \textit{if } \epsilon \in \textsc{first}(\beta) \\ \textsc{first}(\alpha X^{(j)} \cdot \beta) & \textit{otherwise} \end{cases}$$

*and* FOLLOW$(X)$ *is defined to be the union of all the sets* FOLLOW$(X^{(j)})$*, together with the end-of-string symbol, \$, if $X$ is the start symbol.*

## 2.5 The function TestSelect()

GLL parsers use FIRST and FOLLOW sets to exclude control flow choices. Potential branches in the parser code are 'guarded' by a test against the current input symbol. The basic idea is that at each step in a parse, the parser is attempting to derive a portion of the input string using some production $Y ::= \xi\eta$, that a rightmost segment of the currently parsed input has been derived from the $\xi$ part of the rule, and that the current input symbol is, say, $b$. This parse attempt will be terminated unless $b$ is in FIRST$(\xi \cdot \eta)$, or, if this set contains $\epsilon$, $b$ is in FOLLOW$(Y)$.

In our GLL algorithm these tests will be performed using the function $testSelect(b, L)$, where $b$ is a terminal or \$ and $L$ is an instanced grammar slot. This function is defined as follows

```
$testSelect(b, Y ::= r_1 \cdot r_2)$ {
  if($b \in \textsc{first}(r_1 \cdot r_2)$ or ($\epsilon \in \textsc{first}(r_1 \cdot r_2)$ and $b \in \textsc{follow}(Y)$)) {return true}
  else {return false} }
```

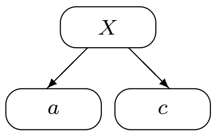
## 3 EBNF derivation trees and SPFFs

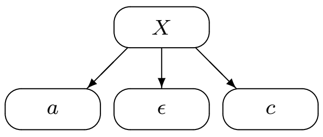
To define parsers for EBNF grammars we need to define their output structures. It turns out that extending the standard SPPF representation of multiple BNF derivation trees is complicated, particularly because closure expressions can have nullable bodies. In this section we develop EBNF SPPFs, giving motivation for our design choices.

A derivation tree is a graphical representation of a derivation of some sentence $u$. Read from left to right the leaves of the derivation tree form a string equal to $u$. For BNF grammars the structure of a derivation tree for $u$ matches exactly one left-most derivation of $u$. A node $v$ labelled $X$ corresponds to a particular instance of the symbol $X$ in the derivation and if this instance is replaced using the grammar rule $X ::= x_1 \dots x_d$ then $x_1, \dots, x_d$ label the children of $v$. We now describe EBNF derivation trees and then define the binarised EBNF SPPFs which efficiently combine all the derivation trees of a given sentence.

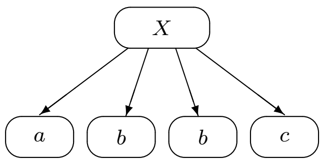
### 3.1 EBNF derivation trees

There are several choices to be considered when extending the definition of a derivation tree to EBNF grammars. For example, it is perhaps tempting to put nothing in the derivation tree when an optional expression is not used. Then the rule $X ::= a[b]c$ would generate the tree fragment on the left, below, from $ac$. However, this would create problems with a rule of the form $Y ::= [b]$ as it would result in $Y$ labelling a leaf node, breaching the property that the sequence of leaf nodes forms the input string. Thus we add an $\epsilon$-labelled node when an optional expression is not used.

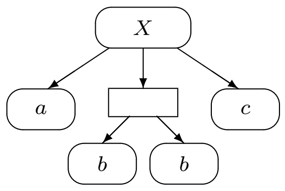




For closure expressions we can choose to add the elements matched in iteration style, in sequence with the previous elements, or to create a new node grouping them together. For example, the rule $X ::= a\{b\}c$ and input $abbc$ gives



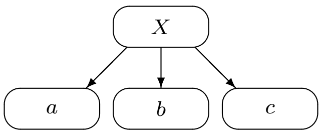
or

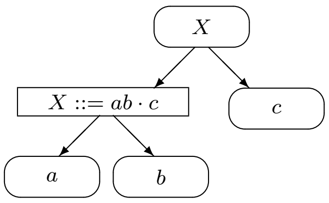


The root-node based choice (on the right above) creates extra nodes and, more importantly, requires extra stack usage in the GLL algorithm. However, when we come to combine derivation trees into a single SPPF structure the root-node based choice allows correct handling of nullable bodies.

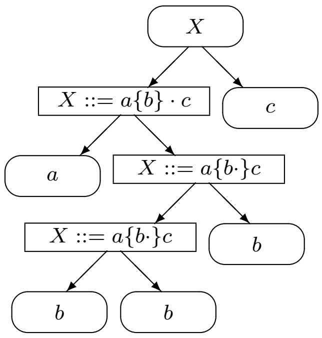
We want the derivation trees also to capture the structure of the regular expressions. This will be achieved naturally as a side effect of the use of binarised derivation trees to ensure worst case cubic size SPPFs.

For deterministic parsers, derivation tree construction is simple in the sense that only one tree is being constructed and the sequence of children of the node about to be constructed is known. Generalised parsers need to construct multiple trees concurrently and the sequences of children of the multiple 'next' nodes need to be stored. To get a cubic bound on the size of the derivation tree representation, and to allow a GLL parser to efficiently construct the derivation trees, the trees are 'binarised' from the left, repeatedly combining the left-most pair of children under a new grammar-slot-labelled intermediate node so that each node has at most two children. For example, the fragment of derivation tree associated with the rule $X ::= abc$ and its binarised equivalent are:

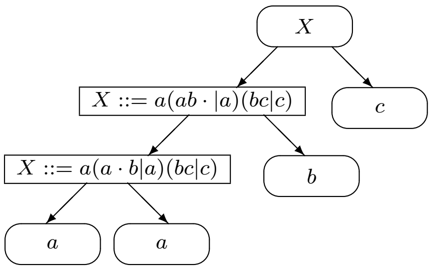


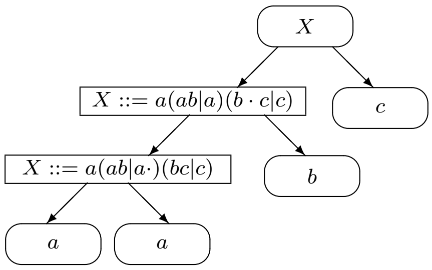


For a more complex example, the following tree is generated from the grammar $X ::= a\{b\}c$ and input string $abbbc$.



By labelling the intermediate nodes, we can distinguish between different matches to a regular expression. For the rule $X ::= a(a\ b \mid a)(b\ c \mid c)$ and input $aabc$ we get two derivation trees with the same structure but different labels on the intermediate nodes, reflecting the choices in the regular expression:





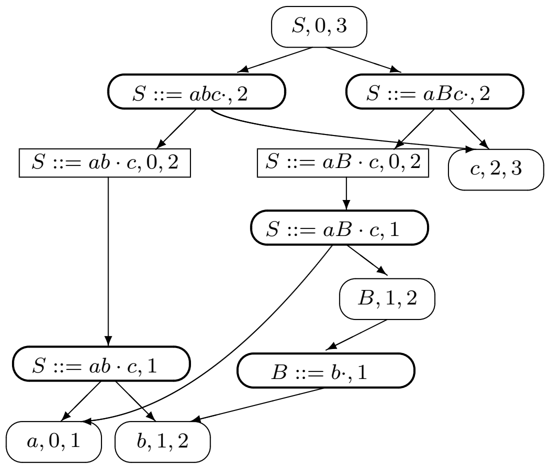
(The issue of distinguishing between constructions when a rule has length one or two, and thus does not generate an intermediate node, is covered by labelled packed nodes which are used to represent ambiguity, introduced in the next section.)

## 3.2 SPPFs for EBNF grammars

We now have a formulation of binarised derivation trees for EBNF grammars, and these trees can be constructed by a GLL parser. However, for some grammars an input string will have multiple derivation trees, which need to be represented compactly. Even for unambiguous grammars, nondeterminism causes several partial trees to be constructed before the final derivation is found. Next we give a discussion of the problems that need to be addressed when extending the BNF SPPF approach to EBNF, and then we give the extensions for each of the regular expression constructs.

The basic SPPF principle is to compress several trees by sharing common parts and recording alternative derivations under special packed nodes. Nodes with the same label are merged and have a packed node child to combine each distinct family of former children. Any derivation tree can be recovered by choosing an appropriate packed node child of each symbol node. A binarised SPPF is a combination of binarised derivation trees.

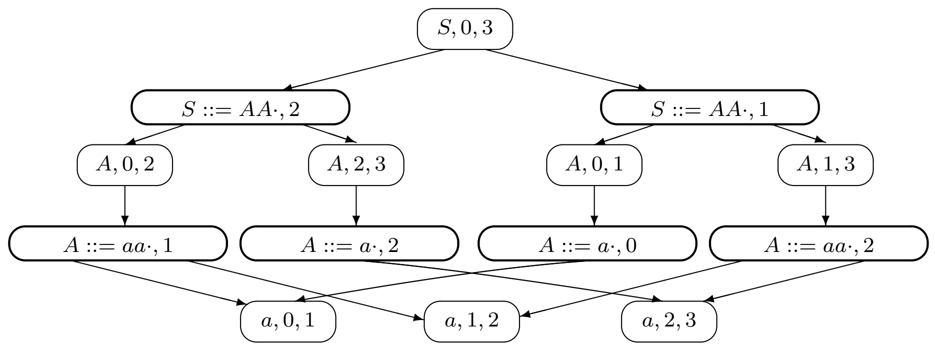
In detail, a binarised SPPF has three types of SPPF nodes: symbol nodes, with labels of the form $(x, i, j)$ where $x$ is a terminal, nonterminal or $\epsilon$ and $0 \leq i \leq j$; intermediate nodes, with labels of the form $(t, i, j)$; and packed nodes, with labels of the form $(t, k)$, where $0 \leq k \leq n$ and $t$ is a slot in a grammar rule (see Definition 4). Packed nodes are drawn as ovals but can be distinguished as they have only one integer field. For example, for the grammar $S ::= a\ b\ c \mid a\ B\ c$, $B ::= b \mid \epsilon$ and input $abc$ we have the following SPPF



Formally, the slots labelling packed nodes are from the instanced grammar, but for ease of reading we omit the instances when we write the labels out.

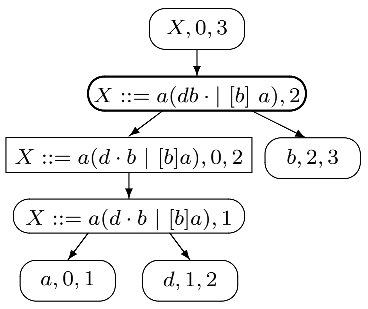
For a symbol node $(x, i, j)$, if $x$ is a terminal then $j = i + 1$, if $x$ is $\epsilon$ then $i = j$, and if $x$ is a nonterminal then $x \overset{*}{\Rightarrow} a_{i+1} \ldots a_j$, where $a_p$ is the $p$th input symbol. A packed node $(t, k)$ has a single integer, $k$, called the *pivot*, in its label. For a symbol or intermediate node with label $(x, i, j)$, we call $(i, j)$ the *extent* of the node ($i$, $j$ are the left and right extents respectively). The SPPF is bipartite in the sense that the children of a symbol or intermediate node are always packed nodes and the children of packed nodes are never packed nodes. An SPPF corresponds to an ambiguous input string if and only if there is an SPPF symbol or intermediate node which is reachable from the root and which has more than one packed node child.

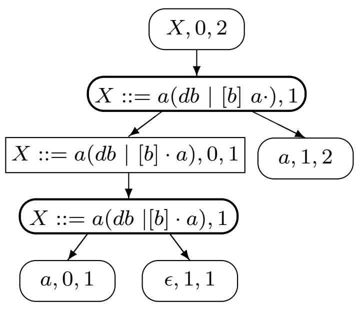
The labelling choice for derivation trees nodes is vital as this forms the basis of the packing and sharing. In an ambiguous grammar, a node with the same label can have more than one set of children, corresponding to different derivations of the same input. Node labels determine when two sets of children can have the same parent and allow the parser to efficiently determine whether a node with a given set of children already exists. In an SPPF, symbol and intermediate nodes are uniquely defined by their labels and packed nodes are uniquely defined by their label and parent node. Packed node labels include the pivot to allow for the possibility that the symbol immediately to the left of the slot may derive more than one portion of the substring. For example, for the grammar $S ::= AA$, $A ::= a|aa$ the string $aaa$ has the following SPPF.



There is one case, involving positive closures with nullable bodies, where an SPPF packed node needs to have two different sets of children, and so we need to duplicate the packed node. To preserve the property that the labels of packed nodes are unique, and thus a parser can efficiently decide whether a particular packed node already exists, we label the duplicate packed node using an alternative slot notation, in which $:$ denotes the slot position. So, for each slot the form $X ::= \alpha \cdot \beta$, we have an alternative notation $X ::= \alpha : \beta$ (which is only used in the labelling of the duplicated packed nodes, see closure expressions II in Section 3.2.3 below).

For a production $X ::= \tau$ where $\tau$ does not contain closure expressions, an associated packed node has labels which correspond in a natural way to the labels of its parent and children. For example, the following subgraphs of an SPPF correspond to derivations from $X ::= a \ ( \ d \ b \ | \ [ \ b \ ] \ a \ )$


$X, 0, 3$
$X ::= a(db \cdot \mid [b] \ a), 2$
$X ::= a(d \cdot b \mid [b]a), 0, 2$
$b, 2, 3$
$X ::= a(d \cdot b \mid [b]a), 1$
$a, 0, 1$
$d, 1, 2$


$X, 0, 2$
$X ::= a(db \mid [b] \ a\cdot), 1$
$X ::= a(db \mid [b] \cdot a), 0, 1$
$a, 1, 2$
$X ::= a(db \ |[b] \cdot a), 1$
$a, 0, 1$
$\epsilon, 1, 1$

There are, however, several special cases for general regular expressions. To give a full specification of the EBNF SPPFs we describe the structure in terms of the possible parent and children of a packed node. Then an EBNF SPPF is any directed graph in which
(i) intermediate and nonterminal symbol nodes have one or more child nodes, each of which is a packed node
(ii) terminal and $\epsilon$ symbol nodes have no children
(iii) packed nodes have one parent and one or two children, each of which is a symbol or intermediate node, as specified in the remainder of this section.
For the SPPFs we construct in this paper we require the additional property
(iv) there are integers $0 \leq n_1 \leq n_2$ and for each $n_1 \leq l \leq n_2 - 1$ there is exactly one terminal symbol node with extent $(l, l+1)$ and the only other leaf nodes are $\epsilon$ nodes.

We also note that our binarisation does not normally create an intermediate node for the leftmost position in a grammar rule or closure body. This saves a significant number of nodes in the SPPF.

### 3.2.1 Packed node labels and slot notations

Given an SPPF packed node $v$ with label $(X ::= \xi \cdot \eta, k)$, $X ::= \xi\eta$ must be a grammar rule, and the extents of the parent and children of $v$ must be of the form $(i, j)$, $(i, k)$, and $(k, j)$ respectively. In the case where $v$ has only one child the extent of the child and the parent must be $(k, j)$.

For BNF we have the important property, discussed above, that all the (packed node) children of a symbol or intermediate node have unique labels. To maintain the unique label property for EBNF SPPFs, in one special case, described in Section 3.2.3, we use a different version of slot notation in which $:$ indicates the slot position. In fact, an SPPF slot label of $v$ must be of one of the forms

$$X ::= \epsilon\, \cdot \quad X ::= \xi' x \cdot \eta \quad X ::= \xi'\{\tau\} \cdot \eta \quad X ::= \xi' < \tau > \cdot \eta$$

$$X ::= \xi'[\tau] \cdot \eta \quad X ::= \xi' < \psi_1 : \psi_2 > \eta'$$

In the penultimate case $\tau$ must be nullable, and in the last case we must have that $\psi_1\psi_2$ is nullable and that $X ::= \xi' < \psi_1 \cdot \psi_2 > \eta'$ has the first-in-positive-Closure property defined in Section 3.2.3. The slot position may not be immediately after a closing parenthesis or immediately after any opening bracket or alternation symbol; in the latter cases the derivation position is already represented by a slot to the left.

In Sections 3.2.2 and 3.2.3 we specify the permitted parents and children of $v$ in terms of the possible slot forms above. However, to formally specify the parent labels and to describe the subtle aspects resulting from productions involving closures, we first introduce some notation for EBNF grammar slots which have particular relationships to a given slot, and we define some Boolean valued variables which denote particular grammar slot properties. The reader may choose to skip over these definitions and just refer back to them while reading Sections 3.2.2 and 3.2.3.

$\underline{aL}$: if $L$ is a slot immediately after an alternate in an alternated regular expression, i.e. immediately before some $|$, then $aL$ denotes the slot immediately after the last alternate in the expression. Otherwise, $aL$ is the same as $L$. For example

$$L:\ X ::= \xi\,(\,\beta\,\cdot\,|\,\alpha\,)\,\eta \qquad aL:\ X ::= \xi\,(\,\beta\,|\,\alpha\cdot)\,\eta$$

$\underline{pL}$: we let $nL$ (*next* slot) denote the slot immediately after a slot $L$ if there is one, and $L$ otherwise. Then, if $L$ is a slot immediately before a closing parenthesis, ), or square bracket, ], we define $pL = p(nL)$ and if $L$ is a slot immediately before an alternation symbol, $|$, then $pL = p(aL)$. Otherwise $pL$ is $L$. For example

$$L:\ X ::= \xi\,[\,x\,(\,\beta\,\cdot\,|\,\alpha\,)\,|\,\rho\,]\,y\,\eta \qquad pL:\ X ::= \xi\,[\,x\,(\,\beta\,|\,\alpha\,)\,|\,\rho\,]\,\cdot\,y\,\eta$$

$\underline{qL}$: we let $bL$ (*before* slot) denote the slot immediately before a slot $L$ if there is one, and $L$ otherwise. If $L$ is a slot at the start an alternate in an alternated regular expression, i.e. immediately after some $|$, then $sL$ (*start* slot) denotes the slot immediately before the first alternate in the expression. Otherwise, $sL$ is the same as $L$. Then if $L$ is a slot immediately after an opening square bracket, [, or parenthesis, (, we define $qL = q(bL)$ and if $L$ is a slot immediately after an alternation symbol, $|$, then $qL = q(sL)$. Otherwise $qL$ is $L$. For example

$$L:\ X ::= \xi\,[\,(\,\beta\,|\,\cdot\,\alpha\,)\,x\,|\,\rho\,]\,y\,\eta \qquad qL:\ X ::= \xi\,\cdot\,[\,(\,\beta\,|\,\alpha\,)\,x\,|\,\rho\,]\,y\,\eta$$

$\underline{lhs(L)}$: if $L$ is $X ::= \xi \cdot \mu$ then $lhs(L)$ is $X$.

$\underline{L^{:}}$ : if $L$ is $X ::= \xi \cdot \mu$ then $L^{:}$ is $X ::= \xi : \mu$

(Because derivation trees are built from the bottom up only $pL$, $L^{:}$ and $lhs(L)$ are used by the GLL algorithm. However, we use $qL$ in our description of the SPPFs.)

The label of the parent of a packed node depends on whether the slot label $X ::= \xi \cdot \eta$ is the end of an alternate, bracket or production, and these properties are captured using $pL$ and the *eoR* predicate:
A grammar slot $L$ is *eoR*, end-of-rule, if it is the end of a production rule, $X ::= \tau\cdot$

There is a degenerate case in which a packed node has only one child, and, for clarity and space efficiency, the two left-most leaf nodes of a derivation from $X$ are usually siblings (there is no additional intermediate node above the left most leaf node). There is also a special case which is needed for a positive closure with nullable body. To formally define these cases we define the *fiR* and *fipC* grammar slot predicates. These in turn are specified using the RE slot predicates *fiRE* and *ffCE*.

An RE slot $s = \alpha \cdot \beta$ is defined to be *fiRE*, first-in-Regular-Expression, if either
(i) $s$ is $x\cdot$ where $x$ is a terminal, nonterminal, or $\epsilon$, or
(ii) $s$ is of the form $s'\tau$, $(s')$, $\tau|s'$, or $s'|\tau$, where $s'$ is *fiRE* and $\tau$ is a regular expression.

A grammar slot $L$ is *fiR*, first-in-Rule, if it is of the form $X ::= \alpha \cdot \beta$ where $\alpha \cdot \beta$ is *fiRE* and if $L$ is not *eoR* (in particular $\beta \neq \epsilon$), $aL$ is not a slot immediately before a closing parenthesis, and $L$ is not of the form $X ::= \psi A \cdot A\phi$ where $A$ is a nullable nonterminal.[2] For example, $X ::= Y \cdot a\ Y$ and $X ::= (a\ Y \mid (b \cdot a \mid a))$ are *fiR* but $X ::= (a\ Y \mid (ba \mid a\cdot))$ is not.

An RE slot $s = \alpha \cdot \beta$ is defined to be *ffCE*, first-for-Closure-Expression, if either
(i) $s$ is $\delta\cdot$ where $\delta$ is a terminal, nonterminal, $\epsilon$, $[\tau]$, $\{\tau\}$, or $< \tau >$, or
(ii) $s$ is of the form $s'\tau$, $(s')$, $[s']$, $\tau|s'$, or $s'|\tau$, where $s'$ is *ffCE* and $\tau$ is a regular expression.

A grammar slot $L$ is *fipC*, first-in-positive-Closure, if it is of the form $X ::= \alpha < \psi_1 \cdot \psi_2 > \beta$ where $\psi_1 \cdot \psi_2$ is *ffCE*.

RE slots which are *fiRE* have the form $\xi x \cdot \eta$, where $x$ is a terminal, a nonterminal, or $\epsilon$, and we call $x$ *the symbol of the slot.*

Where appropriate, we shall apply the notations defined above to instanced grammar rule and regular expression slots in an equivalent way.

### 3.2.2 Parents of packed nodes

Essentially, when the slot position of a packed node label $X ::= \xi \cdot \eta$ is at the end of the rule or alternate, or immediately before a terminal or nonterminal then the parent label is the left hand side nonterminal, the slot after the enclosing bracket or the same as the packed node label, respectively.

$$\begin{tikzcd}
X, i, j \arrow[d] & X ::= \tau(\alpha|\beta) \cdot x\mu, i, j \arrow[d] & X ::= \tau \cdot x\mu, i, j \arrow[d] \\
X ::= \tau\cdot, k & X ::= \tau(\alpha \cdot |\beta)x\mu, k & X ::= \tau \cdot x\mu, k
\end{tikzcd}$$

Formally, if $pL$ is *eoR*, that is if the subgraph rooted at $v$ represents a derivation from the whole expression $\xi\eta$, then the parent of $v$ is a symbol node labelled with *lhs(L).* There is a special case for the base node of a closure with a nullable body, discussed further

[2]This is required to ensure the SPPF does not have two edges between the same pair of nodes, see [17] for an example

below, in which a packed node with slot label $X ::= \xi'\{\tau\} \cdot \eta'$ has a parent with slot label $X ::= \xi'\{\tau\cdot\}\eta'$. Otherwise the parent of $v$ is an intermediate node with slot label $pL$.

The parent of a packed node $(X ::= \xi : \eta, k)$ is the same as the parent of the node labelled $(X ::= \xi \cdot \eta, k)$.

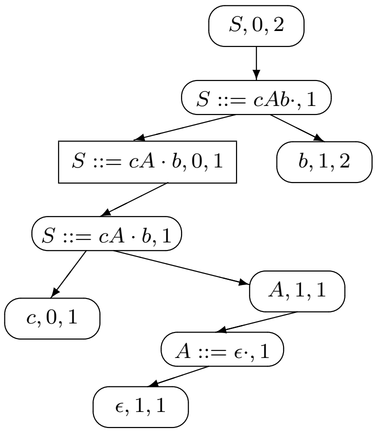
### 3.2.3 Children of packed nodes

Now we consider the labels of the children of a packed node $v$ with label $(X ::= \xi \cdot \eta, k)$. We assume that the parent of $v$ has extent $(i, j)$ and consider each of the cases listed in Section 3.2.1. The technical details are somewhat complex, particularly for the closure expressions. We have organised the presentation to have some motivational material first, then an example and then the formal details. The reader can just initially read the motivation and the examples, and then refer to the formal details for clarification if necessary.

**Packed nodes labelled $(X ::= \xi' x \cdot \eta, k)$ or $(X ::= \epsilon\cdot, k)$**

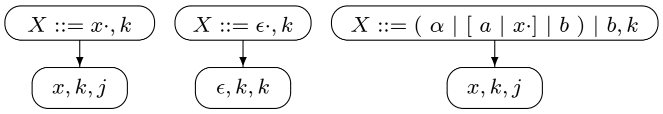
*Motivation:* This case is relatively straightforward. The only subtlety is that the binarisation stops one level above the leaf nodes. So a sequence of three node siblings, in string form $y(u_1, u_2, u_3)$, is binarised to $y(z(u_1, u_2), u_3)$ not to $y(z(w(u_1), u_2), u_3)$. This is determined by the use of the *fiR* attribute.

*Example:* For the grammar $S ::= \ c\ A\ b,\ A ::= \ a\ A \mid \epsilon$ the string $cb$ has SPPF



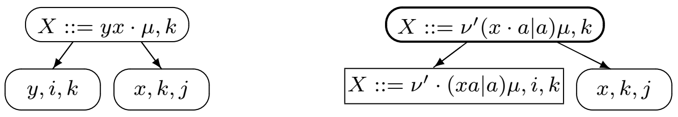
*Formal details:* If $X ::= \xi \cdot \eta$ is *fiR* then there is no packed node with this label. Otherwise, if $v$ is labelled $(X ::= \xi' x \cdot \eta, k)$ then it must have a right child labelled $(x, k, j)$.

If $q(X ::= \xi' \cdot x\eta)$ is of the form $X ::= \cdot\xi\eta$ then $v$ has no left child.



If $q(X ::= \xi' \cdot x\eta)$ is of the form $X ::= \zeta\{\cdot\sigma\}\nu$ or $X ::= \zeta < \cdot\sigma > \nu$ then $v$ can have either no left child or a left child labelled $X ::= \zeta\{\sigma\cdot\}\nu$ or $X ::= \zeta < \sigma\cdot > \nu$, respectively, as discussed in detail later in this section.

If $q(X ::= \xi' \cdot x\eta)$ is *fiR* with symbol $y$ then $v$ must have left child labelled $(y, i, k)$, so the intermediate node above the left-most leaf is omitted, see the example on the left below. Otherwise, $v$ must have a left child of which is an intermediate node labelled $(q(X ::= \xi' \cdot x\eta), k)$.

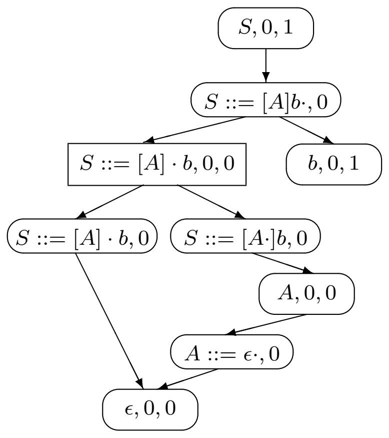


$X ::= yx \cdot \mu, k$
$y, i, k$
$x, k, j$
$X ::= \nu'(x \cdot a|a)\mu, k$
$X ::= \nu' \cdot (xa|a)\mu, i, k$
$x, k, j$

## Optional expressions – $X ::= \xi'[\tau] \cdot \eta$

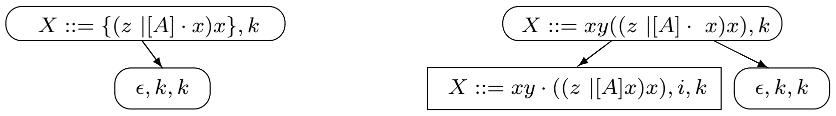
*Motivation:* Packed nodes with labels of the form $(X ::= \xi'[\tau] \cdot \eta, k)$ are permitted because, if $\epsilon \in \text{FIRST}(\tau)$, there are two derivations of $\epsilon$. The slot $X ::= \xi'[\tau_1|\gamma|\tau_2] \cdot \eta$ is used to label the packed node when the bracket itself has been matched to $\epsilon$ and $X ::= \xi'[\tau_1|\gamma \cdot |\tau_2]\eta$ when $\epsilon$ is matched to $\gamma$.

*Example:* For the grammar $S ::= [A]\ b,\ A ::= a \mid \epsilon$ the string $b$ has SPPF



*Formal details:* A packed node $v$ labelled $(X ::= \xi'[\tau] \cdot \eta, k)$ must have right child node labelled $(\epsilon, k, k)$. If $q(X ::= \xi' \cdot [\tau]\eta)$ is of the form $X ::= \cdot\xi\eta$ then $v$ has no left child. If $q(X ::= \xi' \cdot [\tau]\eta)$ is of the form $X ::= \zeta\{\cdot\sigma\}\nu$ or $X ::= \zeta < \cdot\sigma > \nu$ then $v$ can have either no left child or a left child labelled $X ::= \zeta\{\sigma\cdot\}\nu$ or $X ::= \zeta < \sigma\cdot > \nu$, respectively.

If $q(X ::= \xi' \cdot [\tau]\eta)$ is *fiR* with symbol $y$ then $v$ must have left child labelled $(y, i, k)$. Otherwise, $v$ must have a left child which is an intermediate node labelled $(q(X ::= \xi' \cdot [\tau]\eta), k)$.

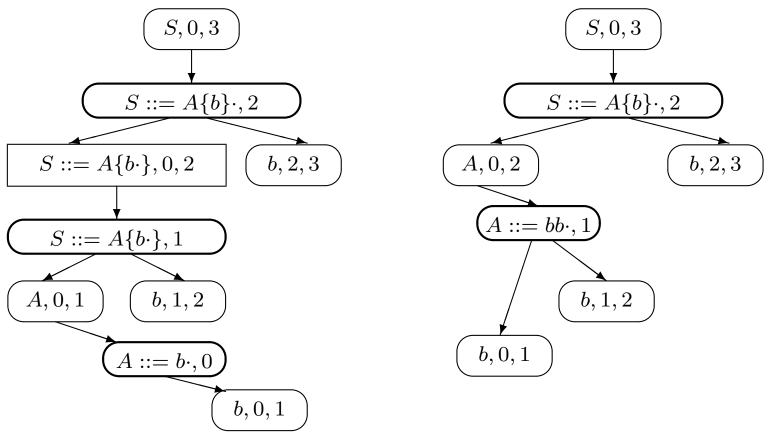


$X ::= \{(z\ |[A] \cdot x)x\}, k$
$\epsilon, k, k$
$X ::= xy((z\ |[A] \cdot\ x)x), k$
$X ::= xy \cdot ((z\ |[A]x)x), i, k$
$\epsilon, k, k$

Similarly, for closures $\{\tau\}$ we use the packed nodes $(X ::= \xi'\{\tau\} \cdot \eta, k)$ when the closure bracket has been matched. In the case where $\tau$ is nullable, this node is used as the base case for the iteration loop, as we shall see later in this section.

## Closure expressions I – $X ::= \xi'\{\tau\} \cdot \eta$, $X ::= \xi' < \tau > \cdot\eta$

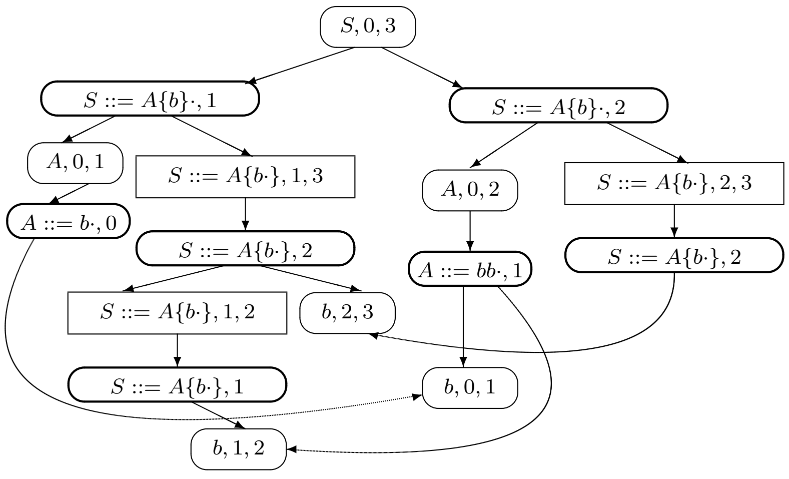
*Motivation:* Consider the grammar $S ::= A\{b\}$, $A ::= bb|b$ and the string $bbb$. If we were to represent the repeated elements from $\{b\}$ in sequence and label the packed nodes in the 'obvious' way, we would get the following two derivation trees



In both cases the packed node children of the node $(S, 0, 3)$ represent derivations whose last step is to match the third $b$ to the positive closure expression and thus they have the same label. But they have different children, so must be different nodes in the SPPF, breaching the requirement that an SPPF label uniquely defines a packed node child of a given node. There are various schemes that might be used to generate different labels but any scheme must also work for obscure grammars such as $S ::= \{B|\epsilon\}\{B\}$, $B ::= Ba|\epsilon$.

Detecting, for a general grammar, where a different label needs to be used is closely related to the ambiguity problem as we would need to decide for each production $X ::= \alpha\{\tau\}\beta$ whether there exist $u, v, w \in \mathbf{T}^*$ such that $\alpha\overset{*}{\Rightarrow}uv$, $\alpha\overset{*}{\Rightarrow}u$, $\tau\overset{*}{\Rightarrow}w$ and $\tau\overset{*}{\Rightarrow}vw$; in addition, closures with nullable bodies need SPPFs with loops. Our approach, as mentioned in Section 3.1, is to create a local root node for the closure expression, in the same way that a nonterminal-labelled symbol node is the root node of the subtree it generates. This node, which is the right child of $v$, is an intermediate node with label of the form $(X ::= \xi'\{\tau\cdot\}\eta, k, j)$.

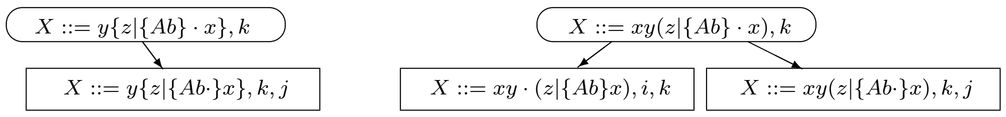
*Example:* The SPPF for $S ::= A\{b\}$, $A ::= bb|b$ and the string $bbb$ is



*Formal details:* A packed node $v$ labelled $(X ::= \xi'\{\tau\}\cdot\eta, k, j)$ has right child either $(\epsilon, k, k)$ (see the optional case above) or $(X ::= \xi'\{\tau\cdot\}\eta, k, j)$.

If $q(X ::= \xi'\cdot\{\tau\}\eta)$ is of the form $X ::= \cdot\xi\eta$ then $v$ has no left child. If $q(X ::= \xi'\cdot\{\tau\}\eta)$ is of the form $X ::= \zeta\{\cdot\sigma\}\nu$ or $X ::= \zeta < \cdot\sigma > \nu$ then $v$ can have either no left child or a left child labelled $X ::= \zeta\{\sigma\cdot\}\nu$ or $X ::= \zeta < \sigma\cdot > \nu$, respectively.

If $q(X ::= \xi' \cdot \{\tau\}\eta)$ is *fiR* with symbol $x$, then $v$ has left child which is a symbol node labelled $x$. Otherwise $v$ has a left child which is an intermediate node whose slot label is $q(X ::= \xi' \cdot \{\tau\}\eta)$.

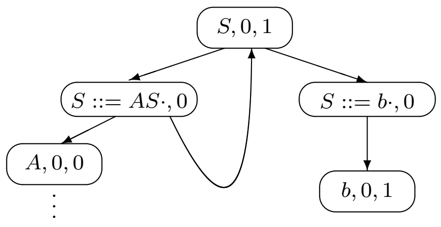

$X ::= y\{z|\{Ab\} \cdot x\}, k$
$X ::= y\{z|\{Ab\cdot\}x\}, k, j$
$X ::= xy(z|\{Ab\} \cdot x), k$
$X ::= xy \cdot (z|\{Ab\}x), i, k$
$X ::= xy(z|\{Ab\cdot\}x), k, j$

The same structures apply to nodes with labels of the form $(X ::= \xi' < \tau > \cdot\eta, k)$, except that they cannot have a single child $(\epsilon, k, k)$.

It might be expected that the closure expression subtree root node should have slot label $X ::= \xi'\{\tau\} \cdot \eta$, the same as its packed node parent. However, for example, for the grammar $S ::= B\{a\}b$, $B ::= \epsilon \mid b$ and the string $ab$ this convention would lead to an SPPF in which two different intermediate nodes have the same label $(S ::= B\{a\} \cdot b, 0, 1)$. Thus we label the root node of a closure expression subtree with the slot immediately before the closing bracket.

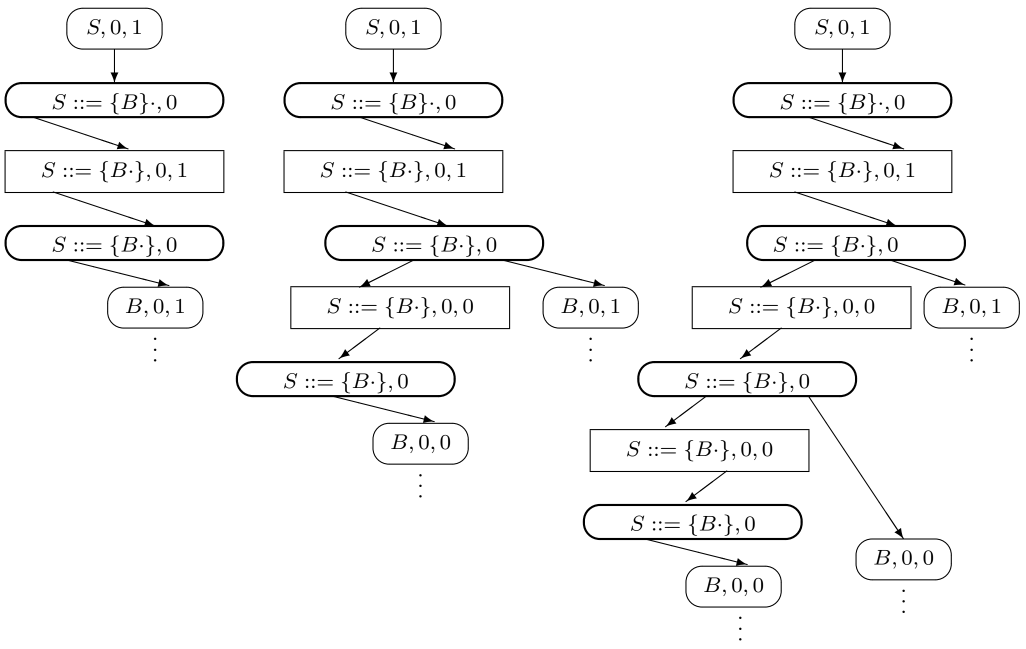
### Closure expressions II – nullable bodies

*Motivation:* For BNF grammars there can be infinitely many derivation trees if the grammar contains a cycle, that is if there is a nonterminal $Y$ such that $Y \overset{+}{\Rightarrow} Y$. These are compressed into a finite SPPF by introducing loops. For example, for the grammar $S ::= A\ S \mid b\ \ A ::= \epsilon \mid a$, the top section of the SPPF for $b$ is



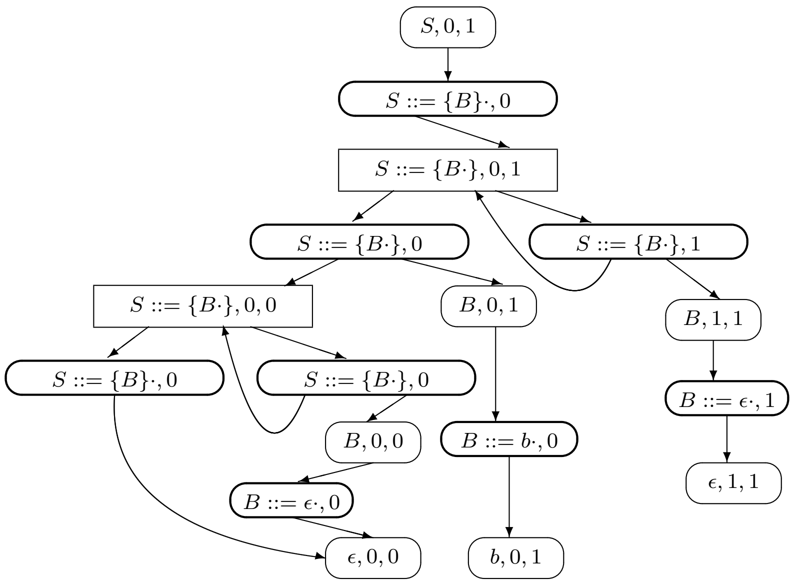
EBNF grammars without cycles can also generate infinitely many derivations of a sentence if they include a closure expression whose body derives $\epsilon$. The merging of nodes with the same label naturally creates loops in the SPPF but the base cases for the loops need special treatment.

Consider the grammar $S ::= \{B\}$, $B ::= b \mid \epsilon$. The following are the top sections of three of the infinitely many derivation trees of the string $b$.



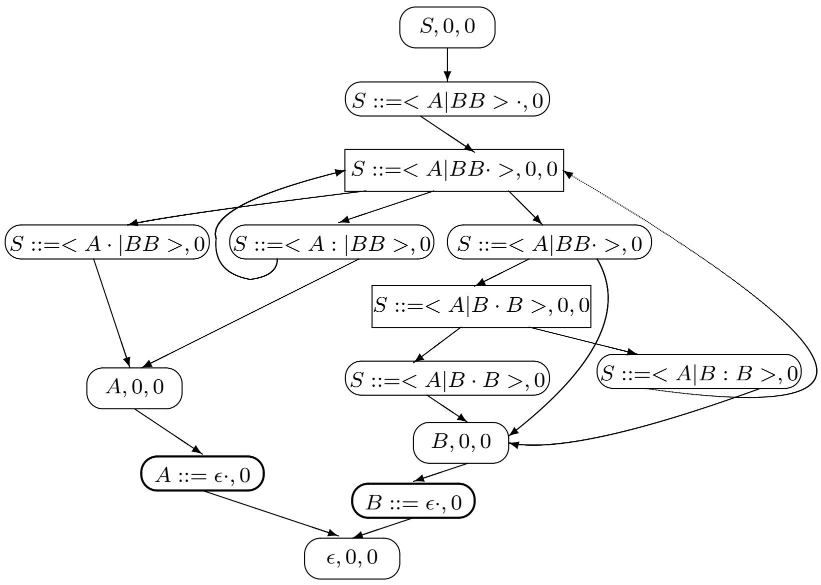
In the rightmost tree the node $(S ::= \{B\cdot\}, 0, 0)$ occurs twice and the merger of these nodes in the SPPF creates the loop that represents the infinitely many $\epsilon$ derivations that are possible at this point. However, in the middle tree the packed node child of $(S ::= \{B\cdot\}, 0, 0)$ forms the base case of the loop. Furthermore, the packed node child of $(S ::= \{B\cdot\}, 0, 1)$ in the left tree has a different set of children from the same node in the other trees. We address these issues by using the packed node introduced above for optional expressions with nullable bodies as the base case for the loop.

*Example:* Including further $\epsilon$ iterations of the closure expression after $b$ has been derived, the full SPPF for $b$ in the above grammar is

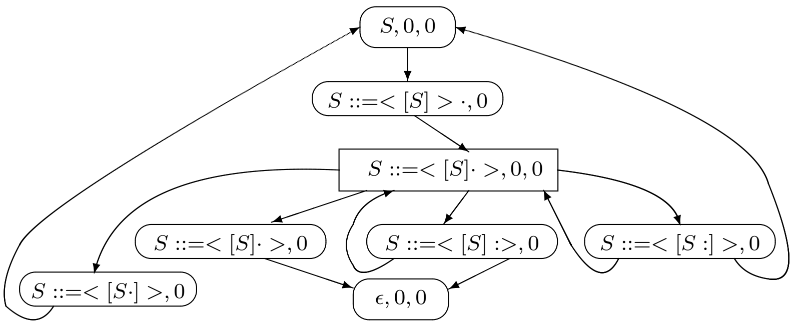


There is no need for an additional optional node for expressions $\{\tau\}$ when $\tau$ is nullable, because the iteration base case performs the role.

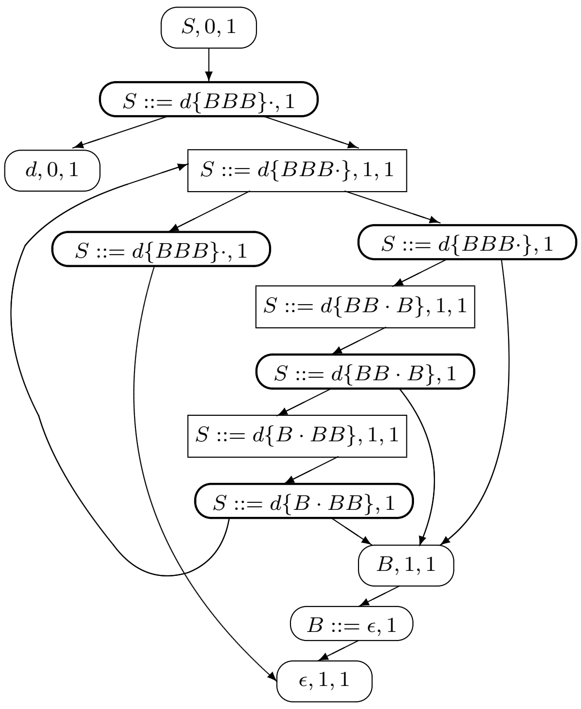
*Motivation:* For positive closures $X ::= \alpha < \tau > \beta$ with nullable $\tau$ the base case for the iteration is the first match of $\tau$ to $\epsilon$, so we represent this separately from the second and subsequent matches. There is no problem with defining such SPPFs, but the labels on the corresponding packed nodes must be different, because they will have the same parent. For a *fipC* slot $X ::= \alpha < \psi_1 \cdot \psi_2 > \beta$ where $\psi_1\psi_2 = \tau$ is nullable, the packed node corresponding to the first match of $\tau$ to $\epsilon$ is labelled $(X ::= \alpha < \psi_1 \cdot \psi_2 > \beta, k)$. The 'duplicate' packed node, corresponding to the second and subsequent matches, is labelled with the version in which the slot symbol is :, $(X ::= \alpha < \psi_1 : \psi_2 > \beta, k)$.
*Examples:* The SPPF for $\epsilon$ in the grammar $S ::=< A|BB >\ \ A ::= a|\epsilon\ B ::= b|\epsilon$ is



To illustrate the delicacy of the SPPF definition in the face of positive closure we give the SPPF for $\epsilon$ with respect to the grammar $S ::=< [S] > \mid a$



In our formulation closure expressions are matched from the left, so $\{\tau\}$ is treated in the left associative form $\{\tau\}\tau$. To see this effect consider the example
$S ::= d\{BBB\}\ B ::= b|\epsilon$. The EBNF SPPF for $d$ is



Here the packed node labelled $(S ::= d\{B \cdot BB\}, 1)$ has right child labelled $(B, 1, 1)$, corresponding to the first part of the 'last' derivation of $BBB$, and left child labelled $(S ::= d\{BBB\cdot\}, 1, 1)$ corresponding to the 'previous' derivations from $\{BBB\}$. (The notion of the first part of a closure body is captured by the *ffCE* property defined in Section 3.2.1.)

*Formal details:* A packed node with label $(X ::= \xi'\{\tau_1 \cdot \tau_2\}\eta', k)$, were $\tau_1 \cdot \tau_2$ is nullable and *ffCE*, may have a left child whose label is $(X ::= \xi'\{\tau_1\tau_2\cdot\}\eta', k, k)$ and right child that is the root of an SPPF for $\epsilon$.

A packed node $v$ with label $(X ::= \xi' < \psi_1 : \psi_2 > \eta, k)$, $X ::= \xi' < \psi_1 \cdot \psi_2 > \eta$ must be *fipC*, $\psi_1\psi_2$ must be nullable, and $v$ must have a left child labelled $(X ::= \xi' < \psi_1\psi_2\cdot > \eta, k, k)$ and the same right child as $(X ::= \xi' < \psi_1 \cdot \psi_2 > \eta, k)$.

### 3.2.4 Remarks

The definition of EBNF SPPFs that we have given in this section is deliberately independent of any particular parsing technology which might be used to construct them. In Section 4 we shall give *getNode*() functions which construct packed nodes, with their parents and children, as part of an EBNF version of the GLL algorithm. These functions can be seen as an operational definition of the EBNF SPPFs.

It is easy to show that each symbol node has at most $O(n)$ packed node children and hence that the size of the SPPF is at most $O(n^3)$, where $n$ is the length of the input string.

We note that SPPFs contain more nodes than are really necessary. Packed nodes are only needed for ambiguous subderivations. We expect that, post parse, an ambiguity resolution phase will be applied to select the derivation required by the user, and at this point all packed nodes should be removed anyway.

It is also possible to give a version of the SPPF which has just packed nodes together with intermediate scaffolding ambiguity nodes where required. This is essentially the internal structure used by ASF+SDF for its BNF SPPFs. The packed nodes would then be labelled with both the extents and the pivot, $(X ::= \xi \cdot \eta, i, k, j)$, but the total number of SPPF nodes and edges would be reduced significantly. We have not done this in this theoretically oriented presentation because the resulting graphs are still worst case cubic in size and they are less clearly related to the original derivation trees, making the illustrating examples less clear.

## 4 GLL parsing of EBNF grammars

We now define a version of the GLL parsing approach which constructs SPPFs of the form defined in the previous section.

A GLL parser for an EBNF grammar has a section of code for each production of each nonterminal. The parser effectively traverses the grammar using the input string to guide the traversal. When a terminal symbol is reached it is matched against the current input symbol, when a nonterminal, $X$ say, is reached the traverser records its current grammar position for subsequent return, and then jumps to the section of code for $X$. When the parser reaches the end of the code for $X$, the return position is retrieved and the parser jumps back to that point. To allow for nested jumps a chain of return positions is constructed, corresponding to the familiar function call and return stack mechanism. However, the call and return mechanism is handled explicitly to allow multiple traversal threads to be maintained. Thus the lines of the parsing algorithm are labelled. Most of the labels correspond to grammar slots and we shall not distinguish between the labels and the corresponding slots. There are also labels associated with each nonterminal and labels used for loops in the closure expressions.

The input string is assumed to be held in an array $I$ and the GLL parser maintains three global variables, the current input position $c_I$, the top of the current call/return chain, $c_U$, and the SPPF node, $c_N$, which has just been constructed. Although the GLL algorithm is top down in the recursive descent sense, the SPPF is constructed 'on the way back up', so children are built before their parents. The call/return chains are combined into a single graph structure, the GSS, in which the nodes are labelled with the return labels and there are arcs from each node to the node immediately below it in the chain.

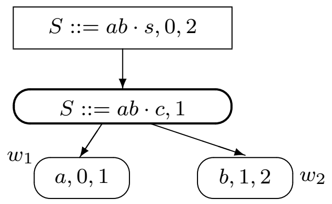
We shall describe formally the construction of a GLL parser for a given EBNF grammar by specifying templates based on the structural definition of a regular expression. We shall also give formal specifications of the SPPF and GSS construction functions. Before this we give an informal description and an example. Throughout this section we shall use the end of string symbol \$ as the bottom-of-stack symbol, thus avoiding the need to have special cases when a stack is empty. We shall also use $\Delta$ to denote a 'dummy' SPPF node which is used when there is no node to consider. Again this removes the need for special cases.

### 4.1 GLL parser behaviour

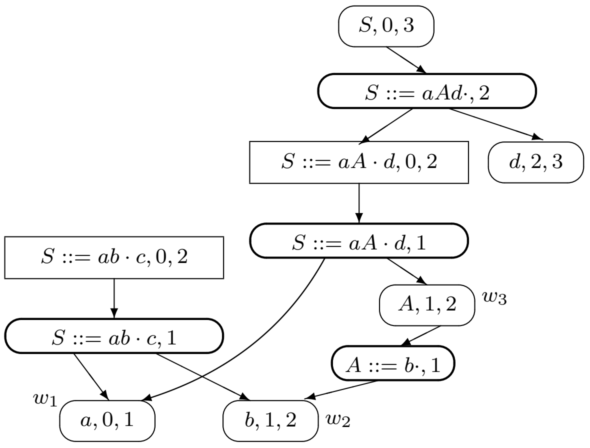
To illustrate the approach consider the simple grammar

$$S ::= a\ b\ c \mid a\ A\ d \qquad A ::= b \mid d$$

and input string $abd$. We read the first input symbol $a$ and begin to traverse the grammar from the start symbol $S$. Both alternates of the rule for $S$ have $a$ in their FIRST sets so we record the second choice in a descriptor $(S ::= \cdot aAd, \$, 0, \Delta)$, the \$ and $\Delta$ values indicate that there is no current stack or SPPF node, and the integer 0 records that the input pointer needs to be returned to the start position. Then we proceed to the position $S ::= \cdot abc$. The symbol to the right of the slot is a terminal so we match it to the current input symbol, create an SPPF node $w_1$ labelled $(a, 0, 1)$, read the next input symbol, $b$, and move to the next grammar position $S ::= a \cdot bc$, with $w_1$ as the current SPPF node. Again we match the input symbol, create an SPPF node $w_2$ labelled $(b, 1, 2)$ and then an intermediate node labelled $(S ::= ab \cdot c, 0, 2)$ with grandchildren $w_1$ and $w_2$. We then move to the grammar position $S ::= ab \cdot c$, at which point this traversal terminates as the input symbol, $d$, is not matched. The following SPPF fragment has been constructed so far.



Next the recorded traversal $(S ::= \cdot aAd, \$, 0, \Delta)$ is resumed. The first input symbol $a$ is read and matched. As the SPPF node labelled $(a, 0, 1)$ already exists this is made the current node and the traverser moves to position $S ::= a \cdot Ad$. Since the symbol to the right is a nonterminal, the return position $S ::= aA \cdot d$ and current SPPF node $w_1$ are pushed onto a stack and the traverser jumps to the productions for $A$. Since the current input symbol is $b$ there is only one viable alternate and the traversal moves to the position $A ::= \cdot b$ where the input symbol is matched. At position $A ::= b \cdot$ an SPPF node $w_3$ labelled $(A, 1, 2)$ with grandchild $w_2$ is created, then the stack is popped, an SPPF intermediate node labelled $S ::= aA \cdot b$ with grandchildren $w_1$ and $w_3$ is created, and the traversal continues from the retrieved position $S ::= aA \cdot d$. The last input symbol is matched in the same way and this traversal terminates in success, with the input pointer at the end of the string and an empty stack. All traversal paths have now been explored and the following SPPF constructed.



In general, we can think of the GLL parser as traversing the grammar, associating process configurations with multiple branches in the traversal to allow all traversals ultimately to be explored. At any point in a traversal the parser is at some grammar slot,

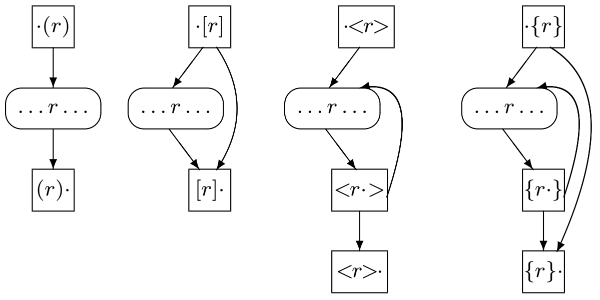
$L$, with current input position $c_I$, stack top $c_U$ and SPPF node $c_N$, represented by the process descriptor $(L, c_U, c_I, c_N)$.

If the slot is immediately before a terminal which is not the current input symbol then this traversal thread terminates. If the slot is immediately before a terminal which is the current input symbol, or immediately before $\epsilon$, an SPPF packed node is created whose left child is the current node and whose right child is labelled with the input terminal, or $\epsilon$. This packed node is made a child of a node labelled with the next slot or, if the next slot is the end of an alternate, the left hand side nonterminal of the slot, and the parser moves to the next slot.

If the slot is immediately before a nonterminal, $A$ say, a GSS node $u$ is created labelled with the next slot (the 'return' position) and the current input position. An arc labelled with the current SPPF node is created from $u$ to the current GSS node and then $u$ becomes the current GSS node. For each production of the rule for $A$ the current input symbol is tested against the selector set for that production. If the test is successful a process descriptor is created which records the slot at the start of that production, the current GSS node and input pointer, and the dummy SPPF node (because the node ultimately created will have its left child retrieved from the GSS arc). These descriptors are recorded in a set $\mathcal{U}$ and, if they do not already exist, added to a set $\mathcal{R}$ for subsequent processing. This traversal thread then terminates.

If the slot is at the end of a production of a grammar rule then the current GSS node is 'popped'. For each child, $v$, of the current SPPF node, a descriptor $(L, v, i, w)$ is created, where $(L, i)$ is the label of the current GSS node and $w$ is an SPPF node. If $z$ is the label of the arc to $v$ then $w$ is the SPPF node whose packed node child has left child $z$ and right child the current SPPF node.

For the EBNF bracketed and alternated regular expressions, the parser traverses all the associated control flow paths, illustrated in the following diagrams. Where there are branches in the control flow, process descriptors are created and, if they do not already belong to $\mathcal{U}$, added to $\mathcal{U}$ and $\mathcal{R}$.



In detail, if the traverser is immediately before a bracketed expression $(r)$, $\{r\}$, $< r >$ or $[r]$ then, provided that *testSelect*() returns true, the parser traverses the body $r$ of the expression. In addition, if the slot is immediately before an optional expression $[r]$ or a closure expression $\{r\}$, a descriptor is added to $\mathcal{U}$ corresponding to the slot immediately after the expression.

If the traverser is immediately before a closing bracket ) or ], the traversal continues from the slot immediately after the bracket. If the traverser is immediately before a closing bracket } or $>$ then the parser creates a descriptor for the slot immediately after the bracket and then traverses the body $r$ of the expression again.

As we discussed with respect to SPPFs, there are subtleties around optional and closure expressions whose body is nullable, which will be discussed with the formal specification in Section 5.

If the traverser is immediately before an alternated expression then a descriptor is created for each alternate for which *testSelect*() returns true and then the traversal thread is terminated. If the traverser is at the end of an alternate in an alternated expression then it moves to the slot at the end of the expression, where a test for repeated traversal is conducted. If this test detects repetition then the traversal thread terminates, otherwise it continues from this point.

When a traversal thread terminates, an unprocessed descriptor $(L, u, i, w)$ is selected from $\mathcal{R}$ and traversal resumes from $L$ with $c_U = u$, $c_I = i$ and $c_N = w$. If $\mathcal{R}$ is empty then the parser terminates. The SPPF will contain all the derivations of the input.

## 4.2 An example

The following is a GLL parser for the EBNF grammar

$$S ::= A \ < b > \qquad A ::= (\ d \mid A\ c)\ d$$

The function *add*() creates descriptors, *create*() builds the GSS and the *getNode*() functions build the SPPF. These functions will be formally defined in Sections 4.3 and 4.4. The line labels have no semantics, however we use E for labels that also label GSS nodes, J for the start of a nonterminal code block, C for the start of closure constructs and A for the end of alternated subexpressions.

```
     read the input into $I$ and set $I[m] := \$$
     create GSS node $u_0 := (L_0, 0)$
     $\mathcal{R} := \emptyset$ ; $\mathcal{U} := \emptyset$ ; $c_U := u_0$; $c_N := \Delta$; $c_I := 0$;  goto $J_S$
$L_0$:  if $(\mathcal{R} \neq \emptyset)$ { remove $(L, u, i, w)$ from $\mathcal{R}$
                 $c_U := u$; $c_I := i$; $c_N := w$;  goto $L$ }
      else  if (there exists and SPPF node labelled $(S, 0, m)$) report success
            else report failure

$J_S$:  if $(testSelect(I[cI], S ::= \cdot A < b >))$ $add(L_4, c_U, c_I, \Delta)$
      goto $L_0$

$L_4$:  $c_U := create(E_0, c_U, c_I, c_N)$;  goto $J_A$
$E_0$:  if $(testSelect(I[cI], S ::= A \cdot < b >)$ is false)  goto $L_0$
      $c_U := create(E_1, c_U, c_I, c_N)$;  $c_N = \Delta$
$C_1$:  if$(testSelect(I[cI], S ::= A < \cdot b >)$ is false)  goto $L_0$
      $c_R := getNodeT(b, c_I)$;  $c_I := c_I + 1$
      $c_N := getNode(S ::= A < b \cdot >, c_N, c_R)$
      if $(testSelect(I[cI], S ::= A < b > \cdot))$ $pop(c_U, c_N)$
      goto $C_1$
$E_1$:  if$(I[cI] \in \text{FOLLOW}(S))$ $pop(c_U, c_N)$
      goto $L_0$

$J_A$:  if$(testSelect(I[cI], A ::= \cdot(d|Ac)d))$ $add(L_5, c_U, c_I, \Delta)$
```

```
    **goto** $L_0$

$L_5$:  **if**($testSelect(I[cI], A ::= (\cdot d|Ac)d)$) $add(L_6, c_U, c_I, c_N)$
    **if**($testSelect(I[cI], A ::= (d|\cdot Ac)d)$) $add(L_7, c_U, c_I, c_N)$
    **goto** $L_0$

$L_6$:  $c_R := getNodeT(d, c_I)$;  $c_I := c_I + 1$
    $c_N := getNode(A ::= (d \cdot |Ac)d, c_N, c_R)$;   **goto** $A_5$
$L_7$:  $c_U := create(E_2, c_U, c_I, c_N)$;   **goto** $J_A$
$E_2$:  **if**($testSelect(I[cI], A ::= (d|A \cdot c)d)$ is false)  **goto** $L_0$
    $c_R := getNodeT(c, c_I)$;  $c_I := c_I + 1$
    $c_N := getNode(A ::= (d|Ac\cdot)d, c_N, c_R)$
$A_5$:  **if**($testRepeat(T_5, c_U, c_I, c_N)$)  **goto** $L_0$
    **if**($testSelect(I[cI], A ::= (d|Ac) \cdot d)$ is false)  **goto** $L_0$
    $c_R := getNodeT(d, c_I)$;  $c_I := c_I + 1$
    $c_N := getNode(A ::= (d|Ac)d\cdot, c_N, c_R)$
    **if**($I[cI] \in \text{FOLLOW}(S)$) $pop(c_U, c_N)$
    **goto** $L_0$
```

### 4.3 The GSS construction functions

The functions which build the GSS are the same as those for the BNF GLL parsers. We note that the function *add*() ensures that elements are added to $\mathcal{R}$ only once so this set can be implemented as a stack or a queue. We also note that it is not possible, in general, to ensure that a GSS node has all its children constructed before any pop action is applied to it. Thus, to ensure that a previously applied pop action is applied to a newly added child, the function *pop*() maintains a set $\mathcal{P}$ of pop actions already applied to the nodes and *create*() applies these actions to any new child it constructs.

```
$add(L, u, i, w)$ {
  **if** (($L, u, i, w$) $\notin \mathcal{U}$ { add ($L, u, i, w$) to $\mathcal{U}$ and to $\mathcal{R}$ } }

$pop(u, z)$ {
  **if** (($u \neq u_0$) and ($u, z$) $\notin \mathcal{P}$) {
        add ($u, z$) to $\mathcal{P}$
        let ($L, k$) be the label of $u$
        **for** each edge ($u, w, v$) {
              let $y$ be the node returned by $getNode(L, w, z)$
              $add(L, v, h, y)$ where $z$ has right extent $h$ } } }

$create(L, u, i, w)$ {
  **if** there is not already a GSS node labelled ($L, i$) create one
  let $v$ be the GSS node labelled ($L, i$)
  **if** there is not an edge from $v$ to $u$ labelled $w$ {
        create an edge from $v$ to $u$ labelled $w$
        **for** all (($v, z$) $\in \mathcal{P}$) {
              let $y$ be the node returned by $getNode(L, w, z)$
```

```
            $add(L, u, h, y)$ where $z$ has right extent $h$ } }
    **return** $v$ }
```

## 4.4 SPPF constructing functions

The SPPF is constructed by the *getNode*() functions which are called with a grammar slot, $L$, as a parameter. The functions construct a symbol or intermediate node and a packed node, and the corresponding labels are related to, but not always equal to, $L$. We use the notation given in Section 3.2.1. As we have said, packed nodes are labelled with instanced slots, while the other nodes are labelled with uninstanced slots and symbols. Where possible, we suppress the difference to make the presentation less cluttered, however in this section the parameters, $L$, to the *getNode*() functions are instanced grammar slots and we write $exp(L)$ for the corresponding uninstanced slots.

To illuminate the correspondence between the structure of a grammar and its GLL parser, we put the SPPF construction code into separate functions. In a production parser many of these functions can be 'in-lined'. Further efficiency can be obtained where tests are carried out on the current position (slot) to determine which form of SPPF construction function is used as these could be replaced with a modification to the parser generation process. However, in this paper we are focusing on the general algorithm structure not implementation efficiency so we have functions to build $\epsilon$ and terminal labelled SPPF nodes and a function, *getNode*(), which builds a nonterminal or intermediate node together with a packed node child.

The role of $getNode(L, w, z)$ is to create a node, $y$, with a packed node child, $v$, whose label is determined by $L$ and which has left child $w$ and right child $z$. The clerical detail in the function is to handle the special cases and ensure that $v$ has the correct label, as discussed in Section 3. Recall, as discussed in Section 3.2.3, that for slots which are *fipC*, we may need two packed nodes labelled with the same slot. We handle this using the label $L^{\cdot}$, which is instantiated by *getNode*(). The function *getBaseNode*() constructs the parts of the SPPF required for the special case where the body of a closure expression is nullable. The attributes *eoR*, *fiR*, *fipC*, and *pL* are defined in Section 3.2.1.

```
getNodeE($i$) {
    **if** there is no SPPF node $y$ labelled $(\epsilon, i, i)$ create one
    return $y$ }

getNodeT($a, i$) {
    **if** there is no SPPF node $y$ labelled $(a, i, i+1)$ create one
    return $y$ }

getNode($L, w, z$) {
    **if** ($L$ is *fiR*) { return $z$ }
    **else** {
      let $k$ and $j$ be the left and right extent of $z$
      **if** $w = \Delta$ set $i := k$  **else** let $i$ be the left extent of $w$
      **if** ($pL$ is *eoR*) set $t := lhs(L)$  **else** set $t := pL$
      **if** there does not exist an SPPF node $y$ labelled $(exp(t), i, j)$ create one
      if ($L$ is $fipC$ and $w \neq \Delta$ and $i = k$)  set $L' := L^{:}$   else set $L' := L$
      if $y$ does not have a child labelled $(L', k)$ {
            create one with right child $z$ and, if $w \neq \Delta$, left child $w$ }
      return $y$ } }

$getBaseNode(L, M, i)$ {
    if there does not exist an SPPF node $y$ labelled $(exp(L), i, i)$ create one
    if $y$ does not have a child labelled $(M, i)$ { create one with child $(\epsilon, i, i)$ }
    return $y$ }
```

## 5 Formal GLL EBNF parser specification

In this section we give the code templates which specify the GLL parser for any EBNF grammar. A parser is obtained by substituting the specific instanced grammar production rules into the templates.

### 5.1 Additional grammar slot notation

Recall, from Definition 5, that we use grammars in which each terminal, nonterminal and $\epsilon$ on the right hand sides of rules is instanced. The instances uniquely identify the position of a subexpression in the right hand side of a production. Also recall that a grammar slot is any position immediately before or immediately after a base or bracketed regular expression on the right hand side of a grammar rule or instanced grammar rule.

The templates essentially define the lines of the parser code which need to be generated for each grammar slot, and the corresponding code will need to refer to related slots.

For a base or bracketed instanced regular expression $r$ we shall write $L_r$ for the slot immediately before $r$ and $E_r$ for the slot immediately after $r$. So, if $r$ lies in the instanced rule $X ::= h_1 r h_2$, we have

$$L_r : \quad X ::= h_1 \cdot r h_2 \qquad\qquad E_r : \quad X ::= h_1 r \cdot h_2$$

For a concatenated instanced regular expression $r = r_1 r_2 \dots r_d$, we shall write $L_r$ for the slot immediately before the first expression in $r$ and $E_r$ for the slot immediately after the last expression in $r$. So $L_r = L_{r_1}$ and $E_r = E_{r_d}$.

For an alternated instanced regular expression $r = r_1 \mid \dots \mid r_d$, $L_r$ is $L_{r_1}$ and $E_r$ is $E_{r_d}$.

Note that there is only one grammar slot between any two characters in a grammar rule and in some cases $L_r = L_s$ or $L_r = E_s$ with $r \neq s$.

### 5.2 Avoiding repeated grammar traversal

It is important to ensure both that a GLL parser has reasonable worst case complexity and, for practical efficiency, that traversal paths are not repeated. Symbol and intermediate SPPF nodes are uniquely defined by their labels, so there are at most $O(n^2)$ of these, where $n$ is the input length. Each such node has at most $O(n)$ children giving a cubic bound on the size of the SPPF. There are at most $O(n^2)$ descriptors and the use of the set $\mathcal{U}$ ensures that no descriptor is added to $\mathcal{R}$ more than once. Thus to ensure that the parsers have worst case cubic run-time complexity, we only require that processing a descriptor has order at most $O(n)$.

It is possible for a GLL parser to return to the same configuration, i.e. the same point in the code with the same input pointer, stack node and SPPF node, more than once. For BNF grammars this can only happen after a pop action, and the return point is a slot immediately after a nonterminal. For EBNF grammars it can also happen when two alternates of an alternated regular expressions match the same substring or when the body of a closure or optional expression derives $\epsilon$. For BNF, recording descriptors in the set $\mathcal{U}$ ensures that a path already explored by the algorithm is not repeated. Descriptors are created for the nonterminal return slots as part of the stack activity needed to ensure the correct matching of call and returns. But there is no stack activity associated with the end of an alternate in an alternated regular expression or the repetition of a loop in a closure expression. Thus, in these cases we record the corresponding configurations in a set $\mathcal{TR}$. Elements of $\mathcal{TR}$ are 4-tuples $(T, u, i, w)$ where $T$ is a unique label associated with the regular expression, $u$ is a GSS node, $i$ is an integer and $w$ is an SPPF node. The function *testRepeat*$(T, u, i, w)$ tests whether $(T, u, i, w)$ is in $\mathcal{TR}$. If it is then the next descriptor will be processed, if it is not then it will be added and the execution will continue. The function *testRepeat*() is defined as follows:

```
testRepeat$(T, u, i, w)$ {
    if $((T, u, i, w) \in \mathcal{TR})$ { return true }
    else { add $(T, u, i, w)$ to $\mathcal{TR}$ and return false } }
```

(Note: the elements of $\mathcal{TR}$ will be of the form $(T, c_u, c_I, c_N)$ when they are added so in fact we can store 5-tuples $(T, j, c_U.slot, i, c_N.slot)$ and thus the worst case size of $\mathcal{TR}$ is $O(n^2)$.)

For alternated expressions, at the end of each alternate there is a call to *testRepeat*() and the thread is terminated if the test returns true. For optional expressions $[r]$ whose body derives $\epsilon$ we create a part of the derivation tree corresponding to $\epsilon$ and add a descriptor to continue this thread once the body, $r$, has been considered. We use *testRepeat*() to add the element to $\mathcal{TR}$, preventing the same thread from being continued in the traversal in which the body derives $\epsilon$. The algorithm then continues with $r$, without creating a descriptor. For closure expressions whose body is nullable it is possible to return to the start of the associated code loop without having read any more input or built any new SPPF nodes. Thus a repeat test is also placed at the start of the loop.

With the tests, EBNF GLL parsers are worst-case cubic in both space and run-time complexity. To avoid unnecessary tests, the repeat test for optional and closure expressions is only included when the body of the expression is nullable.

## 5.3 Outer level GLL algorithm

We begin by giving the template for generating the outer level of a GLL parser. This initialises the global variables, and contains the outer loop which is executed once for each descriptor created. For each nonterminal $X$ in the grammar there is a section of algorithm, *code*$(X)$, the template for which is given in the next section. Throughout the algorithm the following notation is used:

$m$ is a constant integer whose value is the length of the input
$I$ is a constant integer array of size $m + 1$ containing the input string

$c_I$ is an integer variable holding the current input position
GSS is a weighted digraph whose nodes are labelled with elements of the form $(L, j)$
  where $L$ is a grammar slot or $L_0$
$c_U$ is a GSS node variable, $c_N$, $c_T$ and $c_R$ are SPPF node variables
$\mathcal{P}$ is a set of GSS node, SPPF node, integer triples
$\mathcal{R}$ is the set of descriptors waiting to be processed
$\mathcal{U}$ is the set of all parser descriptors constructed so far
$\Delta$ denotes a dummy SPPF node, used to avoid special cases
\$ denotes the end-of-string character

In addition to the grammar slots, we require a label $L_0$ which labels the start of the descriptor selection code and a label $J_X$, for each nonterminal $X$, which labels the start of the code associated with $X$. We also need labels $A_r$ for each instanced alternated regular expression $r$ to label the repeat test, labels $C_r$ for each instanced closure expression $r$ to label the start of the associated loop, and labels $T_r$ for regular expression instances that employ the repeat test.

When the descriptors have all been dealt with, the test for acceptance is made by checking for the existence of the SPPF node labelled with the start symbol and the extent $(0, m)$.

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

```
       set $m$ to be the length of the input string
       read the input into $I$ and set $I[m] := \$$
       create GSS node $u_0 = (L_0, 0)$
       $\mathcal{U} := \emptyset$; $\mathcal{R} := \emptyset$; $\mathcal{P} := \emptyset$; $c_U := u_0$; $c_N := \Delta$; $c_I := 0$
       **goto** $J_S$
$L_0$: **if** $\mathcal{R} \neq \emptyset$ {
           remove a descriptor, $(L, u, i, w)$ say, from $\mathcal{R}$
           $c_U := u$; $c_N := w$; $c_I := i$;  **goto** L }
       **else**  **if** (there exists an SPPF node labelled $(S, 0, m)$) { report success }
              **else** { report failure }
$J_A$: *code*$(A)$
     ...
$J_Z$: *code*$(Z)$
```

## 5.4 Template for *code*$(X)$

Consider the instanced grammar rule $X ::= r_1 \mid \ldots \mid r_p$. We give the specification for *code*$(X)$ in terms of functions *code*$(r_i)$. We refer to the specifications of *code*$(X)$ and *code*$(r)$ as the *EBNF GLL parser templates.*

```
$code(X) =$          $\textbf{if}(testSelect(I[c_I], L_{r_1}))$ { $add(L_{r_1}, c_U, c_I, \Delta)$ }
                     ...
                     $\textbf{if}(testSelect(I[c_I], L_{r_p})$ { $add(L_{r_p}, c_U, c_I, \Delta)$ }
                     $\textbf{goto } L_0$
               $L_{r_1}$ : $code(r_1)$;   $\textbf{if}(I[c_I] \in \text{FOLLOW}(X))$ { $pop(c_U, c_N)$ };   $\textbf{goto } L_0$
                     ...
               $L_{r_p}$ : $code(r_p)$;   $\textbf{if}(I[c_I] \in \text{FOLLOW}(X))$ { $pop(c_U, c_N)$ };   $\textbf{goto } L_0$
```

## 5.5 Templates for regular expressions

For an instanced regular expression $r$ we define *code*$(r)$ recursively on the structure of the underlying regular expression $\tau = exp(r)$.

### Base expressions

If $exp(r)$ is $\epsilon$

$$code(\epsilon^{(j)}) \quad = \quad c_R := getNodeE(c_I);\;\; c_N := getNode(E_{\epsilon^{(j)}}, c_N, c_R)$$

If $exp(r)$ is a terminal $a$

```
$code(a^{(j)})$   =   $c_R := getNodeT(a, c_I)$;   $c_I := c_I + 1$
                      $c_N := getNode(E_{a^{(j)}}, c_N, c_R)$
```

If $exp(r)$ is a nonterminal $Y$

```
$code(Y^{(j)})$   =        $c_U := create(E_{Y^{(j)}}, c_U, c_I, c_N)$;  $\textbf{goto } J_Y$
                  $E_{Y^{(j)}}$ :
```

### Bracketed expressions

If $r$ is a bracketed expression $(s)$

$$code(r) \quad = \quad code(s)$$

If $r$ is a bracketed expression $[s]$ and $\epsilon \notin \text{FIRST}(s)$

```
$code(r)$   =       $\textbf{if}(testSelect(I[c_I], E_r))$ {
                      $c_R := getNodeE(c_I)$;  $c_T := getNode(E_r, c_N, c_R)$
                      $add(E_r, c_U, c_I, c_T)$}
                    $\textbf{if}(testSelect(I[c_I], L_s)\ is\ false)$ $\textbf{goto } L_0$
                    $code(s)$
                $E_r$ :
```

If $r$ is a bracketed expression $[s]$ with $\epsilon \in \text{FIRST}(s)$

```
$code(r)$   =       $\textbf{if}(testSelect(I[c_I], E_r))$ {
                       $c_R := getNodeE(c_I)$;  $c_T := getNode(E_r, c_N, c_R)$
                       $add(E_r, c_U, c_I, c_T)$;  $testRepeat(T_r, c_U, c_I, c_T)$}
                    $code(s)$
                    $\textbf{if}(testRepeat(T_r, c_U, c_I, c_N))$ $\textbf{goto } L_0$
                $E_r$ :
```

As we have already discussed, for closure expressions we have a local root node in the SPPF but we do not need to record the return position at the end of a closure expression because, unlike nonterminal calls, this is uniquely defined. However, we do need to record the SPPF node which will ultimately be the left sibling of the node associated with the closure expression, and nesting and ambiguity mean that this record will need to be some form of GSS. For simplicity we use the GSS already employed and treat closure expressions via the *create*() function in the same way as nonterminals.

If $r$ is a bracketed expression $\{s\}$ and $\epsilon \notin \text{FIRST}(s)$

```
code(r)  =        $c_U := create(E_r, c_U, c_I, c_N)$
                  if$(testSelect(I[c_I], E_r))$ {
                      $c_N := getNodeE(c_I);\ \ pop(c_U, c_N)$}
                  $c_N := \Delta$
             $C_r$ : if$(testSelect(I[c_I], L_s)\ is\ false)$ goto $L_0$
                  $code(s)$
                  if$(testSelect(I[c_I], E_r))\ \ pop(c_U, c_N)$
                  goto $C_r$
             $E_r$ :
```

If $r$ is a bracketed expression $\{s\}$ and $\epsilon \in \text{FIRST}(s)$

```
code(r)  =        $c_U := create(E_r, c_U, c_I, c_N)$;
                  $c_N := getBaseNode(E_s, E_r, c_I)$
                  if$(testSelect(I[c_I], E_r))\ pop(c_U, c_N)$
             $C_r$ : if$(testRepeat(T_r, c_U, c_I, c_N))$  goto $L_0$
                  if$(testSelect(I[c_I], L_s)\ is\ false)$ goto $L_0$
                  $code(s)$
                  if$(testSelect(I[c_I], E_r))\ pop(c_U, c_N)$
                  goto $C_r$
             $E_r$ :
```

Note, we may expect the call to *getBaseNode*() to be guarded by a follow set test. However, if the body can match $\epsilon$ then it must, even if it must also match a non-empty string before proceeding, i.e. even if the follow set test fails at the start of the process. For example, for the expression $\{a|\epsilon\}b$ and string $ab$ there is still an $\epsilon$ subtree before, and after, the $a$ is matched.

If $r$ is a bracketed expression $< s >$ and $\epsilon \notin \text{FIRST}(s)$

```
code(r)  =        $c_U := create(E_r, c_U, c_I, c_N);\ \ c_N := \Delta$
             $C_r$ : if$(testSelect(I[c_I], L_s)\ is\ false)$ goto $L_0$
                  $code(s)$
                  if$(testSelect(I[c_I], E_r))\ \ pop(c_U, c_N)$
                  goto $C_r$
             $E_r$ :
```

If $r$ is a bracketed expression $< s >$ and $\epsilon \in \text{FIRST}(s)$

```
code(r)  =        $c_U := create(E_r, c_U, c_I, c_N)$;   $c_N := \Delta$
            $C_r$ : **if**$(testRepeat(T_r, c_U, c_I, c_N))$  **goto** $L_0$
                **if**$(testSelect(I[c_I], L_s)$ $is\ false)$ **goto** $L_0$
                $code(s)$
                **if**$(testSelect(I[c_I], E_r))$ $pop(c_U, c_N)$
                **goto** $C_r$
            $E_r$ :
```

## Concatenated and alternated expressions

If $r$ is a concatenated expression $r_1 \dots r_d$

```
code(r)  =        $code(r_1)$
                  **if**$(testSelect(I[c_I], L_{r_2})$ $is\ false)$  **goto** $L_0$
                  $code(r_2)$
                  ...
                  **if**$(testSelect(I[c_I], L_{r_d}$ $is\ false)$ **goto** $L_0$
                  $code(r_d)$
```

If $r$ is an alternated regular expression $r_1 \mid \dots \mid r_d$

```
code(r)  =        **if**$(testSelect(I[c_I], L_{r_1}))$ { $add(L_{r_1}, c_U, c_I, c_N)$ }
                  ...
                  **if**$(testSelect(I[c_I], L_{r_d}))$ { $add(L_{r_d}, c_U, c_I, c_N)$ }
                  **goto** $L_0$
         $L_{r_1}$ : $code(r_1)$;   **goto** $A_r$
                  ...
       $L_{r_{d-1}}$ : $code(r_{d-1})$;   **goto** $A_r$
         $L_{r_d}$ : $code(r_d)$
          $A_r$ : **if**$(testRepeat(T_r, c_U, c_I, c_N))$ **goto** $L_0$
```

Note, we need $A_r$ to be different from $E_r$ because the latter is also the end of the last alternate. If this alternate ends with a nonterminal then *pop*() will have created a descriptor $(E_r, c_U, c_I, c_N)$ and then *testRepeat*() will return true even though no traversal thread may yet have been carried out from $(E_r, c_U, c_I, c_N)$.

## 6 Experimental observations

The main focus of this paper is a theoretical exposition of EBNF SPPFs, which compactly encode all derivations of a given string, together with an efficient native-EBNF GLL algorithm which generates those SPPFs; correctness, not performance, is our primary goal. However, in this section we demonstrate the practicality of the algorithm using data generated from our ART implementation.

A native-EBNF implementation of GLL is desirable so as to maintain the close correspondence between a grammar and its parser, which aids the maintenance of large grammar specifications. However, native-EBNF GLL parsers can also be more efficient than pure-BNF parsers because of (a) reduced stack activity and (b) reduced concurrency, as measured by the number of descriptors created during a parse. Consider a BNF grammar of the form

```
    S ::= a B c         B ::= x x | x y
```

A GLL parser for this grammar generates stack activity (and thus GSS nodes) associated with the nonterminal B, whereas the factorised EBNF equivalent

```
    S ::= a (x x | x y) c
```

built using the parsing templates described above is effectively implemented in-line, with no stack activity and requiring two rather than three descriptors. Performance improvement is also obtained from the use of closures: during parsing each closure encountered causes a single descriptor to be created, but thereafter the bodies are matched using iteration with no corresponding GSS or descriptor activity. An equivalent recursive BNF grammar would generate GSS nodes and descriptors for each recursive call. We illustrate these effects, and the overall practicality of the approach, by reference to the standard grammar for Java.

The Java Language Specification exists in five editions, usually referred to as JLS1 (August 1996), JLS2 (June 2000), JLS3 (May 2005) and JLS7 (February 2013) and JLS8 (February 2015). Java is an interesting language from our perspective because the authors of the language standard have recognised the tension between the needs of the compiler implementer and the needs of the language user. The language's features are described with the aid of a pedagogic grammar which underpins the informal presentation of the language's semantics, but a different grammar is given to facilitate implementation using well-known near-deterministic parser generators.

The pedagogic grammars are written in BNF enhanced with optional instances. In JLS1, considerable care is taken to isolate the problems in the pedagogic grammar and to show an equivalent LALR(1) grammar (though there is no formal proof of correctness). In JLS2, which is the version we shall use, the LALR(1) grammar is replaced by an EBNF grammar which is claimed to be the basis for the reference implementation. There is no discussion of the conversion process, and in fact the grammar contains several errors and infelicities.

Of course, we expect the JLS2 BNF grammar to display significant amounts of nondeterminism as its purpose is to elucidate individual language features, not to form a substrate for implementation. As a result, simply displaying a speed up between the JLS2 EBNF and BNF grammars is an inadequate experiment; we might just be measuring the difference between a highly nondeterministic and an equivalent near-deterministic grammar. To help separate out the effects, as well as giving data for the JLS2 BNF grammar we have constructed synthetic BNF grammars from the JLS2 EBNF grammar by converting closures to left- and right-recursive forms, and then multiplying out parentheses. This mimics what would need to be done to use an EBNF grammar with a BNF-only parser generator. On our test strings, these synthetic grammars generate roughly a quarter of the descriptors generated by the rather nondeterministic JLS2 BNF grammar.

To construct our grammars we automatically extracted the grammars from the PDF source of the language documents using tooling which has also been used to process language standards documents for ANSI-C, ANSI-C++, C# and Pascal. This tooling and the corresponding grammars, including the Java grammar we have used in this paper, can

be found at
`www.cs.rhul.ac.uk/research/languages/projects/grammars/index.html`

For the pedagogic BNF grammar, we suppressed repeated rules which arise from the informal presentation of the semantics and added the rule

```
SimpleTypeName ::= Identifier
```

which is missing from the PDF. The resulting grammar is labelled BNF in the results tables.

For the EBNF grammar, we corrected the rules for `Expression`, `Expression3` and `MethodOrFieldRest` which contain typographic errors in the language standard. Here are the revised rules:

```
Expression ::=  Expression1 { AssignmentOperator Expression1 }

Expression3 ::=  PrefixOp Expression3 |
                 '(' Expression | Type ')' Expression3 |
                 Primary { Selector } { PostfixOp }

MethodOrFieldRest ::=  VariableDeclaratorsRest |
                       MethodDeclaratorRest
```

We also added the following missing rules:

```
ExpressionStatement ::= StatementExpression ';'
ForInitOpt ::= [ ForInit ]
ForUpdateOpt ::= [ ForUpdate ]
```

When we ran the resulting GLL parser, we noted unexpected levels of ambiguity in the EBNF grammar. Recall that the purpose of this grammar is to be near-deterministic, so it is surprising that it generated more ambiguity nodes than the pedagogic grammar. On investigation, in the productions for `Expression1` and `Expression2` we found instances of optional regular expressions with a nullable body:

```
Expression1 ::= Expression2 [ Expression1Rest ]
Expression1Rest ::= [ '?' Expression ':' Expression1 ]
Expression2 ::= Expression3 [ Expression2Rest ]
Expression2Rest ::= { Infixop Expression3 } |
                    Expression3 'instanceof' Type
```

We rewrote these to equivalent unambiguous forms:

```
Expression1 ::= Expression2 [ Expression1Rest ]
Expression1Rest ::= '?' Expression ':' Expression1
Expression2 ::= Expression3 [ Expression2Rest ]
Expression2Rest ::= < Infixop Expression3 > |
                    Expression3 'instanceof' Type
```

| Java grammar | CPU s | Tokens/s | Descriptors | GSS nodes | GSS edges | SPPF nodes |
|---|---|---|---|---|---|---|
| BNF | 0.94 | 15,931 | 1,122,895 | 281,983 | 635,873 | 261,991 |
| EBNF | 0.23 | 65,270 | 177,604 | 79,943 | 85,572 | 208,377 |
| RightBNF | 0.28 | 52,857 | 265,485 | 96,273 | 129,842 | 225,198 |
| LeftBNF | 0.29 | 50,995 | 294,750 | 93,834 | 136,534 | 232,677 |

Table 1: Life

| Java grammar | CPU s | Tokens/s | Descriptors | GSS nodes | GSS edges | SPPF nodes |
|---|---|---|---|---|---|---|
| BNF | 1.68 | 17,079 | 2,169,365 | 517,443 | 1,144,133 | 505,661 |
| EBNF | 0.38 | 75,265 | 381,464 | 162,373 | 180,832 | 432,077 |
| RightBNF | 0.57 | 50,258 | 551,025 | 195,543 | 267,242 | 454,068 |
| LeftBNF | 0.74 | 38,619 | 612,820 | 190,514 | 281,374 | 468,747 |

Table 2: Simulated annealing

In addition, we rewrote instances of nonterminals with suffix `Opt` with appropriate regular expressions, so that, for instance, `ArgumentsOpt` was replaced by `[ Arguments ]`. The resulting grammar is labelled EBNF in the results tables.

Finally, we used the transformations built into our ART parser generator tool to transform each closure into an equivalent left (or right) recursive form, and multiplied out parentheses. These conversions produce the pure BNF grammars referred to as LeftBNF and RightBNF in the results tables.

As test strings, we used the Java source for two small graphical applications which use the Swing graphics library: (i) an implementation of Conway's Game of Life and (ii) an implementation of several heuristic solutions to the Travelling Salesman Problem including one based on simulating annealing. After lexicalisation, these programs were 1,502 and 2,873 tokens long respectively; the test strings comprised ten concatenated copies of each program to give strings of 15,020 and 28,730 tokens. Each parser was run ten times on each string to ensure timing repeatability. The host computer was a Toshiba Tecra Z50-A with a four CPU Core-i7-4600U processor running at 2.1GHz under Windows-7 64-bit (V6.1 build 7601) with 4GByte of memory. We used Oracle's distribution 1.8.0_121 of Java under Eclipse Neon to compile the generated parsers.

We give counts for the number of descriptors, GSS nodes, GSS edges and SPPF nodes. We do not give a count for SPPF edges since, in our implementation, there is no explicit representation of SPPF edges. However, in an SPPF, the order of the number of edges is the same as the order of the number of nodes.

The results show that the GLL parser for the JLS2 EBNF grammar is significantly faster than the pedagogic BNF grammar, and the internal data structures are significantly smaller. This result is, to some degree, a function of the high nondeterminancy in the pedagogic grammar as shown by the factor four difference in descriptor counts. More realistically, compared to the synthetic BNF grammars directly derived from the EBNF grammar, we find speed improvement factors of between 1.2 and 1.94. The absolute run times, corresponding to throughputs using Java of around 65,000–75,000 tokens per second, confirm that our generalised parsing technology is practical. In all cases, the sizes of the data structures also reduces. This is mostly driven by the reduced stack activity arising from the use of iteration within closures: the number of descriptors and the number of GSS edges reduce by 35–40% when using the EBNF grammar.

## 7 Concluding remarks

In this paper we have given a structural definition of regular expressions which can be specified by an SLR(1) grammar and is hence unambiguous. Furthermore, the sublanguages generated by non-terminals corresponding to each template type are disjoint, so there is a straightforward deterministic method for building parsers from the templates. We have defined SPPFs for EBNF grammars, highlighting the problems that can arise if these structures are designed in what initially seems an obvious way. We have extended the GLL algorithm to EBNF grammars in a way that reflects the regular expression based structure of the grammar, and we have demonstrated the practicality of the approach using the Java Standard grammar.

A primary attraction of EBNF grammars is their compactness and relative expressive simplicity. When complex grammar rework is required to generate an efficient parser then it is unlikely that the same grammar is suitable for describing the language semantics. Generalised parsing technologies do not require deterministic grammars, allowing grammar rework to be limited to straightforward manipulations, for efficiency purposes, which do not impact on pedagogic transparency. Given the difficulties of formally (or even informally) establishing grammar correspondences, if the developers of the Java language standard had access to an EBNF GLL parser generator, would they have given a modified pedagogic grammar and dispensed with a separate implementation grammar? It would be interesting to compare the performance of a 'natural' EBNF grammar for Java with the heavily reworked near-deterministic JLS2 EBNF grammar, to allow the value of the reworking be assessed.

As an alternative to the approach presented in this paper, we could consider the regular expressions to simply be compact specifications of a (possibly infinite) set of alternates. So $X ::= \tau$ is treated as $\{X ::= \alpha \mid \alpha \in \tau\}$. The derivation trees and SPPFs are then defined as for BNF grammars. However, a corresponding parser would need machinery for selecting and managing the choice of strings $\alpha \in \tau$ at each step in a derivation, and of course the relationship with the particular regular expression $\tau$ is lost. When considering complexity of parsers the size of the grammar is treated as a constant, but if rules such as $X ::= \{a\}$ are treated in the above way then there are effectively infinitely many grammar rules and operationally the size of the grammar becomes input string dependent. The theoretical study of EBNF SPPFs presented in this paper allows a relatively straightforward GLL EBNF parser construction process and results in worst case cubic parsers.

Finally we note that the EBNF GLL approach can be used for regular expression 'matching'. The parser for $S ::= \tau$ will produce all the matches of a string $u$ to the regular expression $\tau$, in terms of the structure of $\tau$.

We would also like to thank the referees for their very careful reading of this paper and for their helpful comments and corrections.

## References

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

[2] 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.

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

[4] Bryan Ford. Packrat parsing: Simple, powerful, lazy, linear time. In *Proceedings of the 2002 International Conference on Functional Programming*, Oct 2002.

[5] Bryan Ford. Parsing expression grammars: a recognition-based syntactic foundation. In Neil D. Jones and Xavier Leroy, editors, *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2004, Venice, Italy, January 14-16, 2004*, pages 111–122. ACM, 2004.

[6] Robert Grimm. Better extensibility through modular syntax. In Michael I. Schwartzbach and Thomas Ball, editors, *Proceedings of the ACM SIGPLAN 2006 Conference on Programming Language Design and Implementation, Ottawa, Ontario, Canada, June 11-14, 2006*, pages 38–51. ACM, 2006.

[7] Adrian Johnstone and Elizabeth Scott. `rdp` – an iterator based recursive descent parser generator with tree promotion operators. *SIGPLAN notices*, 33(9), September 1998.

[8] 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.

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

[10] 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.

[11] Mark-Jan Nederhof and Janos J. Sarbo. Increasing the applicability of LR parsing. In H.Bunt and M. Tomita, editors, *Recent Advances in Parsing Technology*, pages 35–57. Kluwer Academic Publishers, The Netherlands, 1996.

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

[13] Rahman Nozohoor-Farshi. GLR parsing for $\epsilon$-grammars. In Masaru Tomita, editor, *Generalized LR Parsing*, pages 60–75. Kluwer Academic Publishers, The Netherlands, 1991.

[14] Terence Parr and Kathleen Fisher. Ll(*): the foundation of the antlr parser generator. In *PLDI*, pages 425–436, 2011.

[15] Terence John Parr. *Language translation using PCCTS and C++*. Automata Publishing Company, 1996.

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

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

[18] Elizabeth Scott and Adrian Johnstone. Right nulled GLR parsers. *ACM Transactions on Programming Languages and Systems*, 28(4):577–618, July 2006.

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

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

[21] L. Valiant. General context-free recognition in less than cubic time. *J. Comput. Syst. Sci.*, 10:308–315, 1975.

[22] 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.

[23] 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.

[24] Niklaus Wirth. What can we do about the unnecessary diversity of notation for syntactic definitions? *Communications of the ACM*, 20(11), November 1977.