Faster, Practical GLL Parsing

Ali Afroozeh and Anastasia Izmaylova

Centrum Wiskunde & Informatica, 1098 XG Amsterdam, The Netherlands
{ali.afroozeh, anastasia.izmaylova}@cwi.nl

Abstract. Generalized LL (GLL) parsing is an extension of recursive-descent (RD) parsing that supports all context-free grammars in cubic time and space. GLL parsers have the direct relationship with the grammar that RD parsers have, and therefore, compared to GLR, are easier to understand, debug, and extend. This makes GLL parsing attractive for parsing programming languages.

In this paper we propose a more efficient Graph-Structured Stack (GSS) for GLL parsing that leads to significant performance improvement. We also discuss a number of optimizations that further improve the performance of GLL. Finally, for practical scannerless parsing of programming languages, we show how common lexical disambiguation filters can be integrated in GLL parsing.

Our new formulation of GLL parsing is implemented as part of the Iguana parsing framework. We evaluate the effectiveness of our approach using a highly-ambiguous grammar and grammars of real programming languages. Our results, compared to the original GLL, show a speedup factor of 10 on the highly-ambiguous grammar, and a speedup factor of 1.5, 1.7, and 5.2 on the grammars of Java, C#, and OCaml, respectively.

1 Introduction

Developing efficient parsers for programming languages is a difficult task that is usually automated by a parser generator. Since Knuth's seminal paper [1] on LR parsing, and DeRemer's work on practical LR parsing (LALR) [2], parsers of many major programming languages have been constructed using LALR parser generators such as Yacc [3].

Grammars of most real programming languages, when written in their most natural form, are often ambiguous and do not fit deterministic classes of context-free grammars such as LR(k). Therefore, such grammars need to be gradually transformed to conform to these deterministic classes. Not only is this process time consuming and error prone, but the resulting derivation trees may also considerably differ from those of the original grammar. In addition, writing a deterministic grammar for a programming language requires the grammar writer to think more in terms of the parsing technology, rather than the intended grammar. Finally, maintaining a deterministic grammar is problematic. A real-world example is the grammar of Java. In the first version of the Java Language Specification [4], the grammar was represented in an LALR(1) form, but this format has been abandoned in later versions, most likely due to the difficulties of maintaining an LALR(1) grammar as the language evolved.

Generalized LR (GLR) [5] is an extension of LR parsing that effectively handles shift/reduce conflicts in separate stacks, merged as a Graph Structured Stack (GSS) to trim exponentiality. As GLR parsers can deal with any context-free grammar, there is no restriction on the grammar. Moreover, GLR can behave linearly on LR grammars, and therefore, it is possible to build practical GLR parsers for programming languages [6,7].

Although GLR parsers accept any context-free grammar, they have a complicated execution model, inherited from LR parsing. LR parsing is based on the LR-automata, which is usually large and difficult to understand. As a result, LR parsers are hard to modify, and it is hard to produce good error messages. Many major programming languages have switched from LR-based parser generators, such as Yacc, to hand-written recursive-descent parsers. For example, GNU's GCC and Clang, two major C++ front-ends, have switched from LR(k) parser generators to hand-written recursive-descent parsers[1].

Recursive-descent (RD) parsers are a procedural interpretation of a grammar, directly encoded in a programming language. The straightforward execution model of RD parsers makes them easy to understand and modify. However, RD parsers do not support left-recursive rules and have worst-case exponential runtime. Generalized LL (GLL) [8] is a generalization of RD parsing that can deal with any context-free grammar, including the ones with left recursive rules, in cubic time and space. GLL uses GSS to handle multiple function call stacks, which also solves the problem of left recursion by allowing cycles in the GSS. GLL parsers maintain the direct relationship with the grammar that RD parsers have, and therefore, provide an easy to understand execution model. Finally, GLL parsers can be written by hand and can be debugged in a programming language IDE. This makes GLL parsing attractive for parsing programming languages.

Contributions. We first identify a problem with the GSS in GLL parsing that leads to inefficient sharing of parsing results, and propose a new GSS that provides better sharing. We show that the new GSS results in significant performance improvement, while preserving the worst-case cubic complexity of GLL parsing. Second, we discuss a number of other optimizations that further improve the performance of GLL parsing. Third, we demonstrate how common lexical disambiguation filters, such as follow restrictions and keyword exclusion, can be implemented in a GLL parser. These filters are essential for scannerless parsing of real programming languages. The new GSS, the optimizations, and the lexical disambiguation filters are implemented as part of the Iguana parsing framework, which is available at `https://github.com/cwi-swat/iguana`.

Organization of the paper. The rest of this paper is organized as follows. GLL parsing is introduced in Section 2. The problem with the original GSS in GLL

[1] `http://clang.llvm.org/features.html#unifiedparser`
`http://gcc.gnu.org/wiki/New_C_Parser`

parsing is explained in Section 2.3, and the new, more efficient GSS is introduced in Section 3. Section 4 gives a number of optimizations for implementing faster GLL parsers. Section 5 discusses the implementation of common lexical disambiguation mechanisms in GLL. Section 6 evaluates the performance of GLL parsers with the new GSS, compared to the original GSS, using a highly ambiguous grammar and grammars of real programming languages such as Java, C# and OCaml. Section 7 discusses related work on generalized parsing and disambiguation. Finally, Section 8 concludes this paper and discusses future work.

2 GLL parsing

2.1 Preliminaries

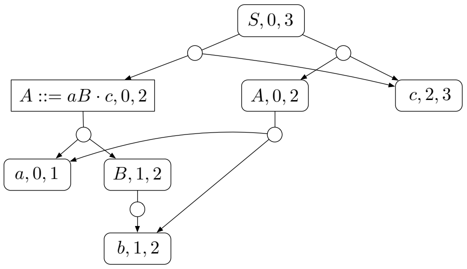
A context-free grammar is composed of a set of nonterminals N, a set of terminals T, a set of rules P, and a start symbol S which is a nonterminal. A rule is written as $A ::= \alpha$, where A (head) is a nonterminal and α (body) is a string in $(T \cup N)^*$. Rules with the same head can be grouped as $A ::= \alpha_1 \mid \alpha_2 \mid \ldots \mid \alpha_p$, where each α_k is called an *alternative* of A. A *derivation step* is written as $\alpha A \beta \Rightarrow \alpha \gamma \beta$, where $A ::= \gamma$ is a rule, and α and β are strings in $(T \cup N)^*$. A *derivation* is a possibly empty sequence of derivation steps from α to β and is written as $\alpha \overset{*}{\Rightarrow} \beta$. A derivation is left-most if in each step the left most nonterminal is replaced by its body. A *sentential* form is a derivation from the start symbol. A *sentence* is a sentential form that only consists of terminal symbols. A sentence is called *ambiguous* if it has more than one left-most derivation.

2.2 The GLL parsing algorithm

The Generalized LL (GLL) parsing algorithm [8] is a fully general, worst-case cubic extension of recursive-descent (RD) parsing that supports all context-free grammars. In GLL parsing, the worst-case cubic runtime and space complexities are achieved by using a Graph-Structured Stack (GSS) and constructing a binarized Shared Packed Parse Forest (SPPF). GSS allows to efficiently handle multiple function call stacks, while a binarized SPPF solves the problem of unbounded polynomial complexity of Tomita-style SPPF construction [9]. GLL solves the problem of left recursion in RD parsing by allowing cycles in the GSS.

GLL parsing can be viewed as a grammar traversal process guided by the input string. At each point during execution, a GLL parser is at a grammar slot (grammar position) L, and maintains three variables: c_I for the current input position, c_U for the current GSS node, and c_N for the the current SPPF node. A grammar slot is of the form $X ::= \alpha \cdot \beta$ and corresponds to a grammar position before or after any symbol in the body of a grammar rule, similar to LR(0) items. A GSS node corresponds to a function call in an RD parser, and is of the form (L, i), where L is a grammar slot of the form $X ::= \alpha A \cdot \beta$, i.e., after a nonterminal, and i is the current input position when the node is created. Note that the grammar slot of a GSS node effectively records the return grammar position, needed to continue parsing after returning from a nonterminal. A GSS edge is of the form (v, w, u), where v and u are the source and target GSS nodes, respectively, and w is an SPPF node recorded on the edge.

GLL parsers produce a binarized SPPF. In an SPPF, nodes with the same subtrees are shared, and different derivations of a node are attached via packed nodes. A binarized SPPF introduces *intermediate* nodes, which effectively group the symbols of an alternative in a left-associative manner. An example of a binarized SPPF, resulting from parsing `"abc"` using the grammar $S ::= aBc \mid Ac$, $A ::= ab$, $B ::= b$ is as follows:



A binarized SPPF has three types of nodes. *Symbol* nodes of the form (x, i, j), where x is a terminal or nonterminal, and i and j are the left and right extents, respectively, indicating the substring recognized by x. *Intermediate* nodes of the form $(A ::= \alpha \cdot \beta, i, j)$, where $|\alpha|, |\beta| > 0$, and i and j are the left and right extents, respectively. Terminal nodes are leaf nodes, while nonterminal and intermediate nodes have *packed nodes* as children. A packed node (shown as circles in the SPPF above) is of the form $(A ::= \alpha \cdot \beta, k)$, where k, the *pivot*, is the right extent of the left child. A packed node has at most two children, both non-packed nodes. A packed node represents a derivation, thus, a nonterminal or intermediate node having more than one packed node is ambiguous.

As mentioned before, a GLL parser holds a pointer to the current SPPF node, c_N, and at the beginning of each alternative, c_N is set to the dummy node, \$. As the parser traverses an alternative, it creates terminal nodes by calls $\textbf{getNodeT}(t, i, j)$, where t is a terminal, and i and j are the left and right extents, respectively. Nonterminal and intermediate nodes are created by calls $\textbf{getNodeP}(A ::= \alpha \cdot \beta, w, z)$, where w and z are the left and right children, respectively. This function first searches for an existing nonterminal node (A, i, j), if $|\beta| = 0$, or intermediate node $(A ::= \alpha \cdot \beta, i, j)$, where i and j are the left extent of w and the right extent of z, respectively. If such a node exists, it is retrieved, otherwise created. Then, w and z are attached to the node via a packed node, if such a packed node does not exist.

In GLL parsing, when the parser reaches a non-deterministic point, e.g., a nonterminal with multiple alternatives, it creates *descriptors*, which capture the parsing states corresponding to each choice, and adds them to a set, so that they can be processed later. A descriptor is of the form (L, u, i, w), where L is a grammar slot, u is a GSS node, i is an input position, and w is an SPPF node. A GLL parser maintains two sets of descriptors: $\mathcal{R}$ for pending descriptors, and $\mathcal{U}$ for storing all the descriptors created during the parsing, to eliminate the duplicate descriptors. A descriptor is added to $\mathcal{R}$, via a call to function **add**, only if it does not exist in $\mathcal{U}$. In addition, a set $\mathcal{P}$ is maintained to store and reuse the results of parsing associated with GSS nodes, i.e., the elements of the form (u, z), where z is an SPPF node. A GLL parser has a main loop that in each iteration, removes a descriptor from $\mathcal{R}$, sets c_U, c_I, and c_N to the respective values in the descriptor, and jumps to execute the code associated with the grammar slot of the descriptor. An example of a GLL parser is given below for the grammar Γ_0: $A ::= aAb \mid aAc \mid a$.

```
$\mathcal{R} := \varnothing; \mathcal{P} := \varnothing; \mathcal{U} := \varnothing$
$c_U := (L_0, 0); c_I := 0; c_N := \$$

$L_0$ :if($\mathcal{R} \neq \varnothing$)
          remove($L, u, i, w$) from $\mathcal{R}$
          $c_U := u$;  $c_I := i$;  $c_N := w$;  goto $L$
       else if (there exists a node $(A, 0, n)$)
          report success
       else report failure

$L_A$ :add($A ::= .aAb, c_U, c_I, \$$)
       add($A ::= .aAc, c_U, c_I, \$$)
       add($A ::= .a, c_U, c_I, \$$)
       goto $L_0$

$L_{\cdot aAb}$ :if($I[c_I] = a$)
          $c_N :=$ getNodeT($a, c_I, c_I + 1$)
       else goto $L_0$
       $c_I := c_I + 1$
       $c_U :=$ create($A ::= aA \cdot b, c_U, c_I, c_N$)
       goto $L_A$

$L_{\cdot aAc}$ :if($I[c_I] = a$)
          $c_N :=$ getNodeT($a, c_I, c_I + 1$)
       else goto $L_0$
       $c_I := c_I + 1$
       $c_U :=$ create($A ::= aA \cdot c, c_U, c_I, c_N$)
       goto $L_A$

$L_{aA \cdot b}$ :if($I[c_I] = b$)
          $c_R :=$ getNodeT($b, c_I, c_I + 1$)
       else goto $L_0$
       $c_I := c_I + 1$
       $c_N :=$ getNodeP($A ::= aAb \cdot, c_N, c_R$)
       pop($c_U, c_I, c_N$);  goto $L_0$

$L_{aA \cdot c}$ :if($I[c_I] = c$)
          $c_R :=$ getNodeT($c, c_I, c_I + 1$)
       else goto $L_0$
       $c_I := c_I + 1$
       $c_N :=$ getNodeP($A ::= aAc \cdot, c_N, c_R$)
       pop($c_U, c_I, c_N$);  goto $L_0$
```

We describe the execution of a GLL parser by explaining the steps of the parser at different grammar slots. Here, and in the rest of the paper, we do not include the check for first/follow sets in the discussion. We also assume that the input string, of length n, is available as an array I. Parsing starts by calling the start symbol at input position 0. At this moment, c_U is initialized by the default GSS node $u_0 = (L_0, 0)$, where L_0 does not correspond to any actual grammar position. Let X be a nonterminal defined as $X ::= \alpha_1 \mid \alpha_2 \mid \ldots \mid \alpha_p$. A GLL parser starts by creating and adding descriptors, each corresponding to the beginning of an alternative: $(X ::= \cdot\alpha_k, c_U, c_I, \$)$. Then, the parser goes to L_0.

Based on the current grammar slot, a GLL parser continues as follows. If the grammar slot is of the form $X ::= \alpha \cdot t\beta$, the parser is before a terminal. If $I[c_I] \neq t$, the parser jumps to L_0, terminating this execution path, otherwise a terminal node is created by **getNodeT**$(t, c_I, c_I + 1)$. If $|\alpha| \geq 1$, the terminal node is assigned to c_R, and an intermediate or nonterminal node is created by

getNodeP$(X ::= \alpha t \cdot \beta, c_N, c_R)$, and assigned to c_N. The parser proceeds with the next grammar slot.

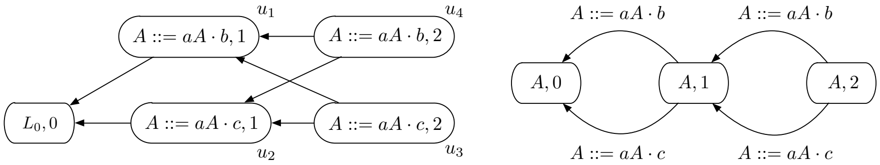
If the grammar slot is of the form $X ::= \alpha \cdot A\beta$, i.e., before a nonterminal, the **create** function is called with four arguments: the grammar slot $X ::= \alpha A \cdot \beta$, c_U, c_I, and c_N. First, **create** either retrieves a GSS node $(X ::= \alpha A \cdot \beta, c_I)$ if such a node exists, or creates one. Let v be $(X ::= \alpha A \cdot \beta, c_I)$. Then, a GSS edge (v, c_N, c_U) is added from v to c_U, if such an edge does not exists. If v was retrieved, the currently available results of parsing A at c_I are reused to continue parsing: for each element (v, z) in $\mathcal{P}$, a descriptor $(X ::= \alpha A \cdot \beta, c_U, h, y)$ is added, where y is the SPPF node returned by **getNodeP**$(X ::= \alpha A \cdot \beta, c_N, z)$, and h is the right extent of z. Finally, the call to **create** returns v, which is assigned to c_U. Then, the parser jumps to the definition of A and adds a descriptor for each of its alternatives.

If the grammar slot is of the form $A ::= \alpha \cdot$, the parser is at the end of an alternative, and therefore, should return from A to the calling rule and continue parsing. This corresponds to the return from a function call in an RD parser. The **pop** function is called with three arguments: c_U, c_I, c_N. Let (L, j) be the label of c_U. First, the element (c_U, c_N) is added to set $\mathcal{P}$. Then, for each outgoing edge (c_U, z, v) from c_U, a descriptor of the form (L, v, c_I, y) is created, where y is the SPPF node returned by **getNodeP**(L, z, c_N). Parsing terminates and reports success if all descriptors in $\mathcal{R}$ are processed and an SPPF node labeled $(S, 0, n)$, corresponding to the start symbol and the whole input string, is found, otherwise reports failure.

2.3 Problems with the original GSS in GLL parsing

To illustrate the problems with the original GSS in GLL parsing, we consider the grammar Γ_0 (Section 2.2) and the input string `"aac"`. Parsing this input string results in the GSS shown in Figure 1(a). The resulting GSS has two separate GSS nodes for each input position, 1 and 2, and each GSS node corresponds to an instance of A in one of the two alternatives: aAb or aAc. This implies that, for example, the following two descriptors, corresponding to the beginning of the first alternative of A, are created and added to $\mathcal{R}$: $(A ::= \cdot aAb, u_1, 1, \$)$, which is added after creating u_1, and $(A ::= \cdot aAb, u_2, 1, \$)$, which is added after creating u_2. Although both descriptors correspond to the same grammar position and the same input position, they are distinct as their parent GSS nodes, u_1 and u_2, are different. The same holds for the following descriptors corresponding to the other alternatives of A: $(A ::= \cdot aAc, u_1, 1, \$)$, $(A ::= \cdot aAc, u_2, 1, \$)$ and $(A ::= \cdot a, u_1, 1, \$)$, $(A ::= \cdot a, u_2, 1, \$)$. This example demonstrates that, although the results of parsing A only depend on the alternatives of A and the current input position, GLL creates separate descriptors for each instance of A, leading to multiple executions of the same parsing actions.

However, the calls corresponding to different instances of A at the same input position are not completely repeated. As can be seen, sharing happens one level deeper in GSS. For example, processing $(A ::= \cdot aAb, u_1, 1, \$)$ or $(A ::= \cdot aAb, u_2, 1, \$)$ matches `a`, increases input position to 2 and moves the grammar



(a) Original GSS

(b) New GSS

Fig. 1: Original and new GSS for parsing "aac" using $A ::= aAb \mid aAc \mid a$.

pointer before A, leading to the call to the same instance of A at input position 2, which is handled by the same GSS node u_4 connected to u_1 and u_2. This sharing, however, happens per nonterminal instance. For example, if we consider the input string "aaacc", a can be matched at input position 2, and therefore, the same result but associated with different instances of A will be stored in set $\mathcal{P}$ as $(u_3, (A, 2, 3))$ and $(u_4, (A, 2, 3))$. Both nodes u_3 and u_4 will pop with the same result $(A, 2, 3)$, and given that both u_3 and u_4 are shared by u_1 and u_2, descriptors that, again, encode the same parsing actions, but account for different parent GSS nodes, will be created: $(A ::= aA \cdot b, u_1, 3, w_1)$, $(A ::= aA \cdot b, u_2, 3, w_1)$ and $(A ::= aA \cdot c, u_1, 3, w_2)$, $(A ::= aA \cdot c, u_2, 3, w_2)$, where $w_1 = (A ::= aA \cdot b, 0, 3)$ and $w_2 = (A ::= aA \cdot c, 0, 3)$.

3 More efficient GSS for GLL parsing

In this section, we propose a new GSS that, compared to the original GSS, provides a more efficient sharing of parsing results in GLL parsing. We use the fact that all calls corresponding to the same nonterminal and the same input position should produce the same results, and therefore, can be shared, regardless of a specific grammar rule in which the nonterminal occurs. The basic idea is that, instead of recording return grammar positions in GSS nodes, i.e., grammar slots of the form $X ::= \alpha A \cdot \beta$, names of nonterminals are recorded in GSS nodes, and return grammar positions are carried on GSS edges. Figure 1(b) illustrates the new GSS resulting from parsing "aac" using $\mathit{\Gamma}_0$.

First, we introduce new forms of GSS nodes and edges. Let $X ::= \alpha \cdot A\beta$ be the current grammar slot, i be the current input position, u be the current GSS node, and w be the current SPPF node. As in the original GLL, at this point, a GSS node is either retrieved, if such a node exists, or created. However, in our setting, such a GSS node is of the form (A, i), i.e., with the label that consists of the name of a nonterminal, in contrast to $X ::= \alpha A \cdot \beta$ in the original GSS, and the current input position. Let v be a GSS node labeled as (A, i). As in the original GLL, a new GSS edge is created from v to u. However, in our setting, a GSS edge is of the form (v, L, w, u), where, in addition to w as in the original GSS, the return grammar position L, i.e., $X ::= \alpha A \cdot \beta$, is recorded.

Second, we remove the default GSS node $u_0 = (L_0, 0)$, which requires a special label that does not correspond to any grammar position. In our setting, the initial GSS node is of the form $(S, 0)$ and corresponds to the call to the grammar start symbol S at input position 0, e.g., $(A, 0)$ in Figure 1(b).

Finally, we re-define the **create** and **pop** functions of the original GLL to accommodate the changes to GSS. We keep the presentation of these functions similar to the ones of the original GLL algorithm [8], so that the difference between the definitions can be easily seen. The new definitions of the **create** and **pop** functions are given below, where L is of the form $X ::= \alpha A \cdot \beta$, $|\alpha|, |\beta| \geq 0$, u and v are GSS nodes, and w, y, z are SPPF nodes.

```
create($L, u, i, w$) {
    if (there exists a GSS node labeled $(A, i)$) {
        let $v$ be the GSS node labeled $(A, i)$
        if (there is no GSS edge from $v$ to $u$ labeled $L, w$) {
            add a GSS edge from $v$ to $u$ labeled $L, w$
            for ($(v, z) \in \mathcal{P}$) {
                let $y$ be the SPPF node returned by getNodeP($L, w, z$)
                add($L, u, h, y$) where $h$ is the right extent of $y$
            }
        }
    } else {
        create a new GSS node labeled $(A, i)$
        let $v$ be the newly-created GSS node
        add a GSS edge from $v$ to $u$ labeled $L, w$
        for (each alternative $\alpha_k$ of $A$) { add($A ::= \cdot\alpha_k, v, i, \$$) }
    }
    return $v$
}

pop($u, i, z$) {
    if ($(u, z)$ is not in $\mathcal{P}$) {
        add $(u, z)$ to $\mathcal{P}$
        for (all GSS edges $(u, L, w, v)$) {
            let $y$ be the SPPF node returned by getNodeP($L, w, z$)
            add($L, v, i, y$)
        }
    }
}
```

The **create** function takes four arguments: a grammar slot L of the form $X ::= \alpha A \cdot \beta$, a GSS node u, an input position i, and an SPPF node w. If a GSS node (A, i) exists (if-branch), the alternatives of A are not predicted at i again. Instead, after a GSS edge (v, L, w, u) is added, if such an edge does not exist, the currently available results of parsing A at i, stored in $\mathcal{P}$, are reused. For each result (v, z) in $\mathcal{P}$, an SPPF node y is constructed, and a descriptor (L, u, h, y) is added to continue parsing with the grammar slot $X ::= \alpha A \cdot \beta$ and the next input position h, corresponding to the right extent of y. If a GSS node (A, i) does not exist (else-branch), such a node is first created, then, an edge (v, L, w, u) is added, and finally, a descriptor for each alternative of A with the input position i and parent node v is created and added.

The **pop** function takes three arguments: a GSS node u, an input position i, and an SPPF node z. If an entry (u, z) exists in $\mathcal{P}$, the parser returns from the function. Otherwise, (u, z) is added to $\mathcal{P}$, and, for each outgoing GSS edge of u, a descriptor is added to continue parsing with the grammar slot recorded on the edge, the current input position and the SPPF node constructed from w and z.

As the signatures of the **create** and **pop** functions stay the same as in the original GLL, replacing the original GSS with the new GSS does not require any modification to the code generated for each grammar slot in a GLL parser. Also note that the new GSS resembles the memoization of function calls used in functional programming, as a call to a nonterminal at an input position is represented only by the name of the nonterminal and the input position.

3.1 Equivalence

As illustrated in Sections 2 and 3, in the original GLL, sharing of parsing results for nonterminals is done at the level of nonterminal instances. On the other hand, in GLL with the new GSS, the sharing is done at the level of nonterminals themselves, which is more efficient as, in general, it results in less descriptors being created and processed. In Section 6 we present the performance results showing that significant performance speedup can be expected in practice. In this section we discuss the difference between GLL parsing with the original and new GSS for the general case, and show that the two GLL versions are semantically equivalent.

The use of the new GSS, compared to the original one, prevents descriptors of the form (L, u_1, i, w) and (L, u_2, i, w) to be created. These descriptors have the same grammar slot, the same input position, the same SPPF node, but different parent GSS nodes. In GLL with the original GSS, such descriptors may be added to $\mathcal{R}$ when, in the course of parsing, calls to different instances of a nonterminal, say A, at the same input position, say i, are made. Each such call corresponds to a parsing state where the current grammar slot is of the form $X ::= \tau \cdot A\mu$ (i.e., before A), and the current input position is i. To handle these calls, multiple GSS nodes of the form $(X ::= \tau A \cdot \mu, i)$, where the grammar slot corresponds to a grammar position after A, are created during parsing. We enumerate all such grammar slots with L_k, and denote GSS nodes (L_k, i) as u_k.

When a GSS node u_k is created, descriptors of the form $(A ::= \cdot\gamma, u_k, i, \$)$ are added. If $a_1 a_2 \ldots a_n$ is the input string and $A \overset{*}{\Rightarrow} a_{i+1} \ldots a_j$, u_k will pop at j, and processing descriptors of the form $(A ::= \cdot\gamma, u_k, i, \$)$ will lead to creation of descriptors of the form $(A ::= \alpha B \cdot \beta, u_k, l, w)$, $i \leq l \leq j$, i.e., in an alternative of A, and of the form $(A ::= \gamma\cdot, u_k, j, (A, i, j))$, i.e., at the end of an alternative of A. All these descriptors encode the parsing actions that do not semantically depend on a specific u_k. Indeed, starting from the same grammar position in an alternative of A, say $A ::= \alpha \cdot \beta$, regardless of a specific u_k, the parsing continues with the next symbol in the alternative and the current input position, and either produces an (intermediate) SPPF node, which does not depend on u_k, moving to the next symbol in the alternative, or fails. Finally, when descriptors of the form $(A ::= \gamma\cdot, u_k, j, (A, i, j))$ are processed, the same SPPF node (A, i, j) will be recorded in set $\mathcal{P}$ for each u_k.

In the original GLL, when u_k is being popped, for each (u_k, z) in set $\mathcal{P}$, where z is of the form (A, i, j), and each outgoing edge (u_k, w, v), a descriptor (L_k, v, j, y), where y is the SPPF node returned by **getNodeP**(L_k, w, z), is added to continue parsing after A. Let v be a GSS node with index h, then h and j are the left and right extents of y, respectively. In the following we show how using the new GSS, descriptors equivalent to (L_k, v, j, y) are created, but at the same time, the problem of repeating the same parsing actions is avoided.

In GLL with the new GSS, when calls to different instances of a nonterminal, say A, at the same input position, say i, are made, a GSS node $u = (A, i)$ is retrieved or created. Similar to the original GLL, when u is created, descriptors of the form $(A ::= \cdot\gamma, u, i, \$)$ are added, and if $A \overset{*}{\Rightarrow} a_{i+1} \dots a_j$, descriptors of the form $(A ::= \alpha B \cdot \beta, u, l, w)$, $i \leq l \leq j$, and of the form $(A ::= \gamma\cdot, u, j, (A, i, j))$ will also be added. The essential difference with the original GLL is that the label of u is A, and therefore, the descriptors corresponding to parsing A at i are independent of the context in which A is used. Upon the first call to A at i, regardless of its current context, such descriptors are created, and the results are reused for any such call in a different context. Finally, when descriptors of the form $(A ::= \gamma\cdot, u, j, (A, i, j))$ are processed, the SPPF node $z = (A, i, j)$ is recorded as a single element (u, z) in set $\mathcal{P}$.

In GLL parsing with the new GSS, whenever the parser reaches a state with a grammar slot of the form $X ::= \tau \cdot A\mu$, and the input position i, there will be an edge (u, L_k, w, v) added to u, where L_k is of the form $X ::= \tau A \cdot \mu$. Finally, for each (u, z) in set $\mathcal{P}$ and each edge (u, L_k, w, v), the descriptor (L_k, v, j, y) will be added, where y is the SPPF node returned by **getNodeP**(L_k, w, z).

3.2 Complexity

In this section we show that replacing the original GSS with the new GSS does not affect the worst-case cubic runtime and space complexities of GLL parsing. To introduce the new GSS into GLL parsing, we changed the forms of GSS nodes and edges. We also re-defined the **create** and **pop** functions to accommodate these changes. However, all these modifications had no effect on the SPPF construction, the **getNode** functions, and the code of GLL parsers that uses **create** and **pop** to interact with GSS. Specifically, this implies that when the main loop of a GLL parser executes, and the next descriptor is removed from $\mathcal{R}$, the execution proceeds in the same way as in the original GLL parsing until the call to either **create** or **pop** is made.

First, we show that the space required for the new GSS is also at most $O(n^3)$. In the new GSS, all GSS nodes have unique labels of the form (A, i), where $0 \leq i \leq n$. Therefore, the new GSS has at most $O(n)$ nodes. In the new GSS, all GSS edges have unique labels of the form (u, L, w, v), where L is of the form $X ::= \alpha A \cdot \beta$, the source GSS node u is of the form (A, i), and the target GSS node v is of the form (X, j). The label of an edge in the new GSS consists of L and w, where w has j and i as the left and right extents, which are also the indices of v and u, respectively. Given that $0 \leq j \leq i \leq n$, the number of outgoing edges for any source GSS node u is at most $O(n)$, and the new GSS has at most $O(n^2)$ edges. Thus the new GSS requires at most $O(n)$ nodes and at most $O(n^2)$ edges.

The worst-case $O(n^3)$ runtime complexity of the original GLL follows from the fact that there are at most $O(n^2)$ descriptors, and processing a descriptor may take at most $O(n)$ time, by calling **pop** or **create**. Now, we show that the worst-case complexity of both **create** and **pop** is still $O(n)$, and the total number of descriptors that can be added to $\mathcal{R}$ is still at most $O(n^2)$. All elements in set $\mathcal{P}$ are of the form (v, z), where v is of the form (A, i), and z has i and j as the left and right extents, respectively, where $0 \leq i \leq j \leq n$. Therefore, the number of elements in $\mathcal{P}$, corresponding to the same GSS node, is at most $O(n)$. Since a GSS node has at most $O(n)$ outgoing edges, $\mathcal{P}$ has at most $O(n)$ elements corresponding to a GSS node, and the new GSS and $\mathcal{P}$ can be implemented using arrays to allow constant time lookup, both **create** and **pop** have the worst-case complexity $O(n)$.

Finally, a descriptor is of the form (L, u, i, w), where w is either \$ or has j and i as the left and right extents, respectively, and j is also the index of u. Thus the total number of descriptors that can be added to $\mathcal{R}$ is at most $O(n^2)$.

4 Optimizations for GLL implementation

The GLL parsing algorithm [8] is described using a set view, e.g., $\mathcal{U}$ and $\mathcal{P}$, which eases the reasoning about the worst-case complexity, but leaves open the challenges of an efficient implementation. The worst-case $O(n^3)$ complexity of GLL parsing requires constant time lookup, e.g., to check if a descriptor has already been added. Constant time lookup can be achieved using multi-dimensional arrays of size $O(n^2)$, however, such an implementation requires $O(n^2)$ initialization time, which makes it impractical for near-linear parsing of real programming languages, whose grammars are nearly deterministic.

For near-linear parsing of real programming languages we need data structures that provide amortized constant time lookup, without excessive overhead for initialization. One way to achieve this is to use a combination of arrays and linked lists as described in [10]. In this approach the user needs to specify, based on the properties of the grammar, which dimensions should be implemented as arrays or linked lists.

In this section we propose an efficient hash table-based implementation of GLL parsers. We show how the two most important lookup structures, $\mathcal{U}$ and $\mathcal{P}$, can be implemented using local hash tables in GSS nodes. The idea is based on the fact that the elements stored in these data structures have a GSS node as a property. Instead of having a global hash table, we factor out the GSS node and use hash tables that are local to a GSS node. In an object-oriented language, we can model a GSS node as an object that has pointers to its local hash tables. In the following, we discuss different implementations of $\mathcal{U}$ and $\mathcal{P}$. We consider

GLL parsing with new GSS, and assume that n is the length of the input, and $|N|$ and $|L|$ are the number of nonterminals and grammar slots, respectively.

Descriptor elimination set ($\mathcal{U}$)*:* set $\mathcal{U}$ is used to keep all the descriptors created during parsing for duplicate elimination. A descriptor is of the form (L, u, i, w), where L is of the form $A ::= \alpha \cdot \beta$, u is of the form (A, j), and w is either a dummy node, or a symbol node of the form (x, j, i), when $\alpha = x$, or an intermediate node of the form (L, j, i). As can be seen, in a descriptor, the input index of the GSS node is the same as the left extent of the SPPF node, and the input index of the descriptor is the same as the right extent of the SPPF node. Also note that the label of the GSS and SPPF node is already encoded in L. Thus we can effectively consider a descriptor as (L, i, j). We consider three implementations of $\mathcal{U}$:

- *Global Array*: $\mathcal{U}$ can be implemented as an array of size $|L| \times n \times n$, which requires $O(n^2)$ initialization time.
- *Global hash table*: $\mathcal{U}$ can be implemented as a single global hash table holding elements of the form (L, i, j).
- *Local hash table in a GSS node*: $\mathcal{U}$ can be implemented as a local hash table in a GSS node. This way, we only need to consider a descriptor as (L, i).

Popped elements ($\mathcal{P}$)*:* The set of popped elements, $\mathcal{P}$, is defined as a set of (u, w), where u is a GSS node of the form (A, i), and w is an SPPF node of the form (A, i, j). For eliminating duplicates, $\mathcal{P}$ can effectively be considered as a set of (A, i, j). We consider three implementations of $\mathcal{P}$:

- *Global Array*: $\mathcal{P}$ can be implemented as an array of size $|N| \times n \times n$, which requires $O(n^2)$ initialization time.
- *Global hash table*: $\mathcal{P}$ can be implemented as a global hash table holding elements of the form (A, i, j).
- *Local hash table in a GSS node*: $\mathcal{P}$ can be implemented as a local hash table in a GSS node. This way we can eliminate duplicate SPPF nodes using a single integer, the right extent of the SPPF node (j).

Hash tables do not have the problem of multi-dimensional arrays, as the initialization cost is constant. However, using a global hash table is problematic for parsing large input files as the number of elements is in order of millions, leading to many hash collisions and resizing. For example, for a C# source file of 2000 lines of code, about 1,500,000 descriptors are created and processed.

Using local hash tables in GSS nodes instead of a single global hash table provides considerable speedup when parsing large inputs with large grammars. First, by distributing hash tables over GSS nodes, we effectively reduce the number of properties needed for hash code calculation. Second, local hash tables will contain fewer entries, resulting in fewer hash collisions and requiring fewer resizing. In the Iguana parsing framework we use the standard `java.util.HashSet` as the implementation of hash tables. Our preliminary results show that, for example, by using a local hash table for implementing $\mathcal{U}$ instead of a global one, we can expect speedup of factor two. Detailed evaluation of the optimizations presented in this section, and their effect on memory usage, is future work.

There are two algorithmic optimizations possible that further improve the performance of GLL parsers. These optimizations remove certain runtime checks that can be shown to be redundant based on the following properties:

1) There is at most one call to the ***create*** *function with the same arguments. Thus no check for duplicate GSS edges is needed.*
The properties of a GSS edge (v, L, w, u) are uniquely identified by the arguments to **create**: L, u, i, w, where L is of the form $X ::= \alpha A \cdot \beta$, and $v = (A, i)$. Therefore, if it can be shown that there is at most one call to **create** with the same arguments, the check for duplicate GSS edges can be safely removed.

Let us consider a call **create**$(X ::= \alpha A \cdot \beta, u, i, w)$. This call can only happen if a descriptor of one of the following forms has been processed, where τ is a possibly empty sequence of terminals and $j \leq i$: (1) $(X ::= \cdot\alpha A\beta, u, j, \$)$ when $\alpha = \tau$; or (2) $(X ::= \gamma B \cdot \tau A\beta, u, j, z)$ when $\alpha = \gamma B\tau$, $|\gamma| \geq 0$. Therefore, for the call to happen more than once, the same descriptor has to be processed again. However, this can never happen as all the duplicate descriptors are eliminated.

2) There is at most one call to the ***getNodeP*** *function with the same arguments. Thus no check for duplicate packed nodes is needed.*
Let us consider a call **getNodeP**$(A ::= \alpha \cdot \beta, w, z)$, where w is either \$ or a non-packed node having i and k as the left and right extents, and z is a non-packed node having k and j as the left and right extents. This call may create and add a packed node $(A ::= \alpha \cdot \beta, k)$ under the parent node, which is either (A, i, j) when $|\beta| = 0$, or $(A ::= \alpha \cdot \beta, i, j)$ otherwise. Clearly, the same call to **getNodeP** will try to add the same packed node under the existing parent node.

Now suppose that the same call to **getNodeP** happens for the second time. Given that a GSS node is ensured to pop with the same result at most once (set $\mathcal{P}$ and **pop**), the second call can only happen if a descriptor of one of the following forms has been processed for the second time, where $u = (A, i)$ and τ is a possibly empty sequence of terminals: (1) $(A ::= \cdot\alpha\beta, u, i, \$)$ when either $\alpha = \tau$ or $\alpha = \tau X$; or (2) $(A ::= \gamma B \cdot \sigma\beta, u, l, y)$, $i \leq l \leq k$, when $\alpha = \gamma B\sigma$, $|\gamma| \geq 0$, and either $\sigma = \tau$ or $\sigma = \tau X$. This can never happen as all the duplicate descriptors are eliminated.

Note that the second optimization is only applicable for GLL parsers with the new GSS. In the original GLL, u can be of the form $(X ::= \mu A \cdot \nu, i)$, and therefore, multiple descriptors with the same grammar slot, the same input position, the same SPPF node, but different parent nodes, corresponding to multiple instances of A, can be added, resulting in multiple calls to **getNodeP** with the same arguments.

5 Disambiguation filters for scannerless GLL parsing

Parsing programming languages is often done using a separate scanning phase before parsing, in which a scanner (lexer) first transforms a stream of characters to a stream of tokens. Besides performance gain, another important reason for a separate scanning phase is that deterministic character-level grammars are virtually nonexistent. The main drawback of performing scanning before parsing is that, in some cases, it is not possible to uniquely identify the type of tokens without the parsing context (grammar rule in which they appear). An example is nested generic types in Java, e.g., `List<List<T>>`. Without the parsing context, the scanner cannot unambiguously detect the type of `>>` as it can be either a right-shift operator or two closing angle brackets.

Scannerless parsing [11,12] eliminates the need for a separate scanning phase by treating the lexical and context-free definitions the same. A scannerless parser solves the problems of identifying the type of tokens by parsing each character in its parsing context, and provides the user with a unified formalism for both syntactical and lexical definitions. This facilitates modular grammar development at the lexical level, which is essential for language extension and embedding [13].

A separate scanning phase usually resolves the character-level ambiguities in favor of the longest matched token and excludes keywords from identifiers. In absence of a separate scanner, such ambiguities should be resolved during parsing. In the rest of this section we show how most common character-level disambiguation filters [14] can be implemented in a GLL parser.

To illustrate character-level ambiguities, we use the grammar below, which is adapted from [14]. This grammar defines a `Term` as either a sequence of two terms, an identifier, a number, or the keyword `"int"`. `Id` is defined as one or more repetition of a single character, and `WS` defines a possibly empty blank.

```
Term  ::= Term WS Term | Id | Num | "int"
Id    ::= Chars
Chars ::= Chars Char | Char
Char  ::= 'a' | .. | 'z'
Num   ::= '1' | .. |'9'
WS    ::= ' ' | ϵ
```

This grammar is ambiguous. For example, the input string `"hi"` can be parsed as either `Term(Id("hi"))`, or `Term(Term(Id("h")),Term(Id("i")))`. Following the longest match rule, the first derivation is the intended one, as in the second one `"h"` is recognized as an identifier, while it is followed by `"i"`. We can use a *follow restriction* ($-\!\!\!/\!\!-$) to disallow an identifier to be followed by another character: `Id ::= Chars -/- Char`. Another ambiguity occurs in the input string `"intx"` which can be parsed as either `Term(Id("intx"))` or `Term(Term("int"), Term(Id("x")))`. We can solve this problem by adding a *precede restriction* ($-\!\!\!\backslash\!\!-$) as follows: `Id ::= Char -\- Chars`, specifying that `Id` cannot be preceded by a character. Finally, we should exclude the recognition of `"int"` as `Id`. For this, we use an exclusion rule: `Id ::= Chars \"int"`.

Below we formally define each of these restrictions and show how they can be integrated in GLL parsing. For follow and precede restrictions we only consider the case where the restriction is a single character, denoted by c. This can be trivially extended to other restrictions such as character ranges or arbitrary regular expressions. We assume that I represents the input string as an array of characters and i holds the current input position.

Follow restriction. For a grammar rule $A ::= \alpha x\beta$, a follow restriction for the symbol x is written as $A ::= \alpha x \mathrel{-\!\!\!/\!\!\!-} c\beta$, meaning that derivations of the form $\gamma A\sigma \Rightarrow \gamma\alpha x\beta\sigma \overset{*}{\Rightarrow} \gamma\alpha x c\tau$ are disallowed. For implementing follow restrictions, we consider the grammar position $A ::= \alpha x\cdot\beta$. If x is a terminal, the implementation is straightforward: if $i < |I|$ and $I[i] = c$, the control flow returns to the main loop, effectively terminating this parsing path. If x is a nonterminal, we consider the situation where a GLL parser is about to create a descriptor for $A ::= \alpha x \cdot \beta$. This happens when pop is executed for a GSS node (x, j) at i. While iterating over the GSS edges, if a GSS edge labeled $A ::= \alpha x \cdot \beta$ is reached, the condition of the follow restriction associated with this grammar position will be checked. If $I[i] = c$, no descriptor for this label will be added.

Precede Restriction. For a grammar rule $A ::= \alpha x\beta$, a precede restriction for the symbol x is written as $A ::= \alpha c \mathrel{-\!\!\!\backslash\!\!\!-} x\beta$, meaning that derivations of the form $\gamma A\sigma \Rightarrow \gamma\alpha x\beta\sigma \overset{*}{\Rightarrow} \tau c x\beta\sigma$ are disallowed. The implementation of precede restrictions is as follows. When a GLL parser is at the grammar slot $A ::= \alpha \cdot x\beta$, if $i > 0$ and $I[i-1] = c$, the control flow returns to the main loop, effectively terminating this parsing path.

Exclusion. For a grammar rule $A ::= \alpha X\beta$, the exclusion of string s from the nonterminal X is written as $A ::= \alpha X \backslash s\beta$, meaning that the language accepted by the nonterminal X should not contain the string s, i.e., $L(X\backslash s) = L(X) - \{s\}$, where L defines the language accepted by a nonterminal. Similar to the implementation of follow restrictions for a nonterminal, when a GSS node (X, j) is popped at i, and the parser iterates over the outgoing GSS edges, if an edge $A ::= \alpha X \cdot \beta$ is found, the condition of the exclusion is checked. If the substring of the input from j to i matches s, no descriptor for the grammar position $A ::= \alpha X \cdot \beta$ is added, which effectively terminates this parsing path.

6 Performance evaluation

To evaluate the efficiency of the new GSS for GLL parsing, we use a highly ambiguous grammar and grammars of three real programming languages: Java, C# and OCaml. We ran the GLL parsers generated from Iguana in two different modes: *new* and *original*, corresponding to the new and original GSS, respectively. Iguana is our Java-based GLL parsing framework that can be configured to run with the new or original GSS, while keeping all other aspects of the algorithm, such as SPPF creation, the same. The optimizations given in Section 4, with the exception of removing checks for packed nodes, which is only applicable to GLL parser with the new GSS, are applied to both modes.

We ran the experiments on a machine with a quad-core Intel Core i7 2.6 GHz CPU and 16 GB of memory running Mac OS X 10.9.4. We executed the parsers on a 64-Bit Oracle HotSpot$^{\text{TM}}$ JVM version 1.7.0_55 with the `-server` flag. To allow for JIT optimizations, the JVM was first warmed up, by executing a large sample data, and then each test is executed 10 times. The median running time (CPU user time) is reported.

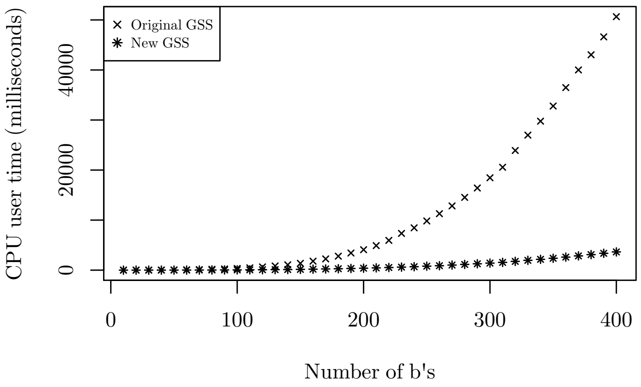


Fig. 2: Running the GLL parsers for grammar $S ::= SSS \mid SS \mid b$

size	time (ms)		# GSS nodes		# GSS edges	
	new	original	new	original	new	original
50	6	35	51	251	3877	18 935
100	45	336	101	501	15 252	75 360
150	151	1361	151	751	34 127	169 285
200	386	4080	201	1001	60 502	300 710
250	791	9824	251	1251	94 377	469 635
300	1403	18 457	301	1501	135 752	676 060
350	2367	32 790	351	1751	184 627	919 985
400	3639	50 648	401	2001	241 002	1 201 410

Table 1: The result of running highly ambiguous grammar on strings of b's.

6.1 Highly ambiguous grammar

To measure the effect of the new GSS for GLL parsing on highly ambiguous grammars, we use the grammar $S ::= SSS \mid SS \mid b$. The results of running a GLL parser with the new and original GSS for this grammar on strings of b's is shown in Figure 2. As can be seen, the performance gain is significant. The median and maximum speedup factors for the highly ambiguous grammar, as shown in Figure 3, are 10 and 14, respectively. To explain the observed speedup, we summarize the results of parsing the strings of b's in Table 1. Note that the number of nodes and edges for the original GSS are slightly more than the numbers reported in [8], as we do not include the check for first and follow sets. As can be seen, GLL with the new GSS has $n+1$ GSS nodes for inputs of length n, one for each call to S at input positions 0 to n. For GLL with the original GSS, there are 5 grammar slots that can be called: $S ::= S \cdot SS$, $S ::= SS \cdot S$, $S ::= SSS\cdot$, $S ::= S \cdot S$, and $S ::= SS\cdot$, which lead to $5n + 1$ GSS nodes. In such a highly ambiguous grammar, most GSS nodes are connected, therefore,

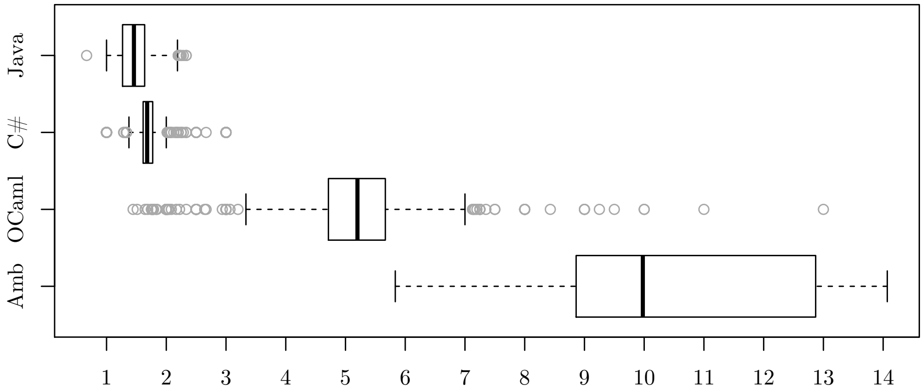


Fig. 3: Comparing the speedup factor of the new and original GSS.

the iteration operations over edges in the create and pop functions will take much more time, as explained in Section 3.1.

6.2 Grammars of programming languages

To measure the effect of the new GSS on the grammars of real programming languages, we have chosen the grammars of three programming languages from their language reference manual.

Java: We used the grammar of Java 7 from the Java Language Specification [15] (JLS). The grammar contains 329 nonterminals, 728 rules, and 2410 grammar slots. We have parsed 7449 Java files from the source code of JDK 1.7.0_60-b19. As shown in Figure 3, the median and maximum speedup factors for Java are 1.5 and 2.3, respectively.

C#: We used the grammar of C# 5 from the C# Language Specification [16]. The grammar contains 534 nonterminals, 1275 rules, and 4195 grammar slots. The main challenge in parsing C# files was dealing with C# directives, such as `#if` and `#region`. C# front ends, in contrast to C++, do not have a separate preprocessing phase for directives. Most C# directives can be ignored as comment, with the exception of the conditional ones, as ignoring them may lead to parse error. As the purpose of this evaluation was to measure the performance of GLL parsers on C# files, and not configuration-preserving parsing, we ran the GNU C preprocessor on the test files to preprocess the conditional directives. The rest of the directives were treated as comments. We have parsed 2764 C# files from the build-preview release of the Roslyn Compiler. As shown in Figure 3, the median and maximum speedup factors for C# are 1.7 and 3, respectively.

OCaml: We used the grammar of OCaml 4.0.1 from the OCaml reference manual [17]. The grammar of OCaml is different from Java and C# in two aspects.

First, OCaml is an expression-based language, as opposed to Java and C#. This provides us with a grammar with different characteristics for testing the effectiveness of the new GSS. Second, the reference grammar of OCaml is highly ambiguous, having numerous operators with different associativity and priority levels. We used a grammar rewriting technique [18] to obtain an unambiguous grammar. The rewritten grammar contains 685 nonterminals, 5728 rules, and 27294 grammar slots. We have parsed 871 files from the OCaml 4.0.1 source release. As shown in Figure 3, the median and maximum speedup factors for OCaml are 5.2 and 13, respectively. The rewriting technique used in [18] to encode precedence rules leads to more rules. This can be one reason for the more significant speedup for the OCaml case, compared to Java and C#. The other possible reason is the nature of OCaml programs that have many nested expressions, requiring high non-determinism. The case of OCaml shows that the new GSS is very effective for parsing large, complex grammars, such as OCaml.

7 Related work

For many years deterministic parsing techniques were the only viable option for parsing programming languages. As machines became more powerful, and the need for developing parsers in other areas such as reverse-engineering and source code analysis increased, generalized parsing techniques were considered for parsing programming languages. In this section we discuss several related work on applying generalized parsing to parsing programming languages.

Generalized parsing. Generalized parsing algorithms have the attractive property that they can behave linearly on deterministic grammars. Therefore, for the grammars that are nearly deterministic, which is the case for most programming languages, using generalized parsing is feasible [19]. For example, the ASF+SDF Meta-Environment [7] uses a variation of GLR parsing for source code analysis and reverse engineering.

The original GLR parsing algorithm by Tomita [5] fails to terminate for some grammars with ϵ rules. Farshi [20] provides a fix for ϵ rules, but his fix requires exhaustive GSS search after some reductions. Scott and Johnstone [21] provide an alternative to Farshi's fix, called Right Nulled GLR (RNGLR), which is more elegant and more efficient. GLR parsers have worst-case $O(n^{k+1})$ complexity, where k is the length of the longest rule in the grammar [9]. BRNGLR is a variation of RNGLR that uses binarized SPPFs to enable GLR parsing in cubic time. Elkhound [6] is a GLR parser, based on Farshi's version, that switches to the machinery of an LR parser on deterministic parts of the grammar, leading to significant performance improvement. Another faster variant of GLR parsing is presented by Aycock and Horspool [22], which uses a larger LR automata, trading space for time.

Disambiguation. Disambiguation techniques that are used in different parsing technologies can be categorized in two groups: implicit or explicit disambiguation. Implicit disambiguation is mostly used in parsing techniques that return at most one derivation tree. Perhaps the name nondeterminism-reducer is a more correct term, as these techniques essentially reduce non-determinism during parsing, regardless if it leads to ambiguity or not. Yacc [3], PEGs [23] and ANTLR [24] are examples of parsing techniques that use implicit disambiguation rules. For example, in Yacc, shift/reduce conflicts are resolved in favor of shift, and PEGs and ANTLR use the order of the alternatives. These approaches do not work in all cases and may lead to surprises for the language engineer.

Explicit disambiguation is usually done using declarative disambiguation rules. In this approach, the grammar formalism allows the user to explicitly define the disambiguation rules, which can be applied either during parsing, by pruning parsing paths that violate the rules, or be applied after the parsing is done, as a parse forest processing step. Post-parse filtering is only possible when using a generalized parser that can return all the derivations in form of a parse forest. Aho et. al show how to modify LR(1) parsing tables to resolve shift/reduce conflicts based on the the priority and associativity of operators [25]. In Scannerless GLR (SGLR) which is used in SDF2 [26], operator precedence and character-level restrictions such as keyword exclusion are implemented as parse table modifications, but some other disambiguation filters such as `prefer` and `avoid` as post-parse filters [14]. Economopoulos et al. [27] investigate the implementation of SDF disambiguation filters in the RNGLR parsing algorithm and report considerable performance improvement.

8 Conclusions

In this paper we presented an essential optimization to GLL parsing, by proposing a new GSS, which provides a more efficient sharing of parsing results. We showed that GLL parsers with the new GSS are worst-case cubic in time and space, and are significantly faster on both highly ambiguous and near-deterministic grammars. As future work, we plan to measure the effect of the new GSS and the optimizations presented in Section 4 on memory, and to compare the performance of our GLL implementation with other parsing techniques.

Acknowledgments. We thank Alex ten Brink who proposed the modification to the GSS in GLL recognizers. Special thanks to Elizabeth Scott and Adrian Johnstone for discussions on GLL parsing, and to Jurgen Vinju for his feedback.

References

1. Knuth, D.E.: On the Translation of Languages from Left to Right. Information and control **8**(6) (1965) 607–639
2. Deremer, F.L.: Practical Translators for LR(k) Languages. PhD thesis, Massachusetts Institute of Technology (1969)
3. Johnson, S.C.: Yacc: Yet Another Compiler-Compiler AT&T Bell Laboratories, `http://dinosaur.compilertools.net/yacc/`.
4. Gosling, J., Joy, B., Steele, G.L.: The Java Language Specification. 1st edn. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (1996)

5. Tomita, M.: Efficient Parsing for Natural Language: A Fast Algorithm for Practical Systems. Kluwer Academic Publishers, Norwell, MA, USA (1985)
6. McPeak, S., Necula, G.C.: Elkhound: A Fast, Practical GLR Parser Generator. In: Compiler Construction, 13th International Conference, CC 2004. (2004) 73–88
7. van den Brand, M., Heering, J., Klint, P., Olivier, P.A.: Compiling language definitions: the ASF+SDF compiler. ACM Trans. Program. Lang. Syst. **24** (2002)
8. Scott, E., Johnstone, A.: GLL parse-tree generation. Science of Computer Programming **78**(10) (October 2013) 1828–1844
9. Johnson, M.: The Computational Complexity of GLR Parsing. In Tomita, M., ed.: Generalized LR Parsing. Springer US (1991) 35–42
10. Johnstone, A., Scott, E.: Modelling GLL parser implementations. In: Software Language Engineering - Third International Conference, SLE 2010. (2010) 42–61
11. Salomon, D.J., Cormack, G.V.: Scannerless NSLR(1) Parsing of Programming Languages. In: Programming Language Design and Implementation. PLDI '89 (1989) 170–178
12. Visser, E.: Scannerless Generalized-LR Parsing. Technical report, University of Amsterdam (1997)
13. Bravenboer, M., Tanter, É., Visser, E.: Declarative, Formal, and Extensible Syntax Definition for AspectJ. In: Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2006. (2006) 209–228
14. van den Brand, M., Scheerder, J., Vinju, J.J., Visser, E.: Disambiguation Filters for Scannerless Generalized LR Parsers. In: Compiler Construction, 11th International Conference, CC 2002. (2002) 143–158
15. Gosling, J., Joy, B., Steele, G., Bracha, G., Buckley, A.: The Java Language Specification Java SE 7 Edition (February 2013)
16. Microsoft Corporation: C# Language Specification Version 5.0. (2013)
17. Leroy, X., Doligez, D., Frisch, A., Garrigue, J., Rémy, D., Vouillon, J.: The OCaml system release 4.01: Documentation and user's manual. (September 2013)
18. Afroozeh, A., van den Brand, M., Johnstone, A., Scott, E., Vinju, J.J.: Safe Specification of Operator Precedence Rules. In: Software Language Engineering - 6th International Conference, SLE. (2013) 137–156
19. Johnstone, A., Scott, E., Economopoulos, G.: Generalised parsing: Some costs. In: Compiler Construction, 13th International Conference, CC 2004. (2004) 89–103
20. Nozohoor-Farshi, R.: GLR Parsing for ϵ-Grammers. In Tomita, M., ed.: Generalized LR Parsing. Springer US (1991) 61–75
21. Scott, E., Johnstone, A.: Right Nulled GLR Parsers. ACM Trans. Program. Lang. Syst. **28**(4) (2006) 577–618
22. Aycock, J., Horspool, R.N.: Faster Generalized LR Parsing. In: Compiler Construction, 8th International Conference, CC'99. (1999) 32–46
23. Ford, B.: Parsing Expression Grammars: A Recognition-Based Syntactic Foundation. In: Principles of programming languages. POPL '04 (2004) 111–122
24. Parr, T., Harwell, S., Fisher, K.: Adaptive LL(*) Parsing: The Power of Dynamic Analysis. In: Object Oriented Programming Systems Languages and Applications. OOPSLA '14, ACM (2014) 579–598
25. Aho, A.V., Johnson, S.C., Ullman, J.D.: Deterministic Parsing of Ambiguous Grammars. In: Principles of Programming Languages. POPL '73 (1973) 1–21
26. Visser, E.: Syntax Definition for Language Prototyping. PhD thesis, University of Amsterdam (1997)
27. Economopoulos, G., Klint, P., Vinju, J.J.: Faster Scannerless GLR Parsing. In: Compiler Construction, 18th International Conference, CC 2009. (2009) 126–141