MASTER

Exploring and visualizing GLL parsing

Cappers, B.C.M.

Award date:
2014

Link to publication
Exploring and visualizing
GLL parsing

Master Thesis

Bram Cappers (B.C.M.)

Supervisor:
Prof. dr. M.J.G. van den Brand

Eindhoven, August 11th 2014
Abstract

Generalized LL parsing is a relatively new technique for parsing arbitrary context-free grammars while achieving a cubic bound on the running time. By overcoming the limitations of traditional LL recursive descent parsers, GLL parsers are capable to support a larger class of grammars using the same reasoning as in LL parsing. In contrast to alternative parsing algorithms, GLL parsers are known for their straightforward implementation and efficiency. Since this type of parsing is still in its infancy, there are still many existing techniques such as error recovery that have to be developed. The design of these techniques however often requires a detailed understanding of how the underlying parser deals with phenomena such as non-determinism and left recursion. Since the GLL algorithm consider multiple stacks in parallel, the actual control flow of the algorithm can become rather complex, thereby making it more difficult for engineers to enhance the algorithm with new extensions. In this project, we will try to overcome this issue by explaining how a GLL parser actually works using visualization techniques. By making the relationships between GLL parsing, LL parsing, and input data explicit, the learning curve of the algorithm can significantly be reduced, thereby making it easier to develop new extensions for GLL parsing. Furthermore, the step-by-step visualization of the algorithm can be used as a debug application to test the behaviour of grammars at early stages of language development.
Preface

This thesis is the result of my gradation project at Eindhoven University at Technology. As a preparation for this project, I studied Generalized LL parsing during an Honors project. Together with Josh Mengerink and Bram van der Sanden, we have investigated how GLL parsing can be implemented in object-oriented languages and how error handling can be realized by means of a plug-in architecture[18]. This project raised my enthusiasm for the topic and therefore became the foundation for this thesis.

I would like to thank Mark van den Brand for his guidance and valuable advice throughout both projects. Furthermore, I would like to thank Jarke van Wijk for his time and effort with respect to the visualization part of this project. Finally, I would like to thank committee members Adrian Johnston, Elizabeth Scott, and Jarke van Wijk for the evaluation of my thesis.

From the first day that I started at the university, I have followed the studies with great pleasure. This experience however would not have been so great without the friends that I have met throughout these years. In particular, I would like to thank Wouter van Heeswijk, Josh Mengerink and Bram van der Sanden for the all the time that we have spend together throughout these studies. Furthermore, I would like to thank my parents, my brother Dennis, and family for their endless support and encouragement throughout my academic career.

Bram Cappers
July 13th 2014
Eindhoven
Contents

1 Introduction 1
   1.1 Glossary and abbreviations .............................. 2

2 Preliminaries 4
   2.1 Preliminaries ........................................... 4

3 Generalized LL parsing 11
   3.1 Recursive descent parsing: the roots .................. 11
   3.2 Knowing where to go next: stacks ..................... 12
   3.3 Towards a parser ....................................... 14
   3.4 Issues LLRD ............................................ 15
   3.5 Explicit stack ......................................... 17
   3.6 Multiple derivations ................................. 25
   3.7 Towards a GLL parser ................................. 36
   3.8 GLL parser ............................................ 44
   3.9 OOGLL ................................................ 50

4 Visualizing GLL 54
   4.1 Requirements ........................................... 54
   4.2 Visualizing GLL: Walking grammars ................. 57
   4.3 Context-highlighting ................................. 73
   4.4 Constructing the visualization ..................... 77

5 Future work & Conclusions 103

Bibliography 107
Chapter 1

Introduction

In this thesis, the foundations and intuition behind Generalized LL parsing (GLL) [5] will be explored. The main goal of this project is to reduce the learning curve of the algorithm so that the development of new extensions such as error recovery and EBNF [1] support can be realized much faster. In order to do so, the algorithm is explored by means of two viewpoints. The first viewpoint will focus on how a GLL parser extends the underlying foundation of left recursive descent parsing (in short LLRD). This viewpoint allows us to reason about the structure of a GLL parser without having to cope with the abstract control-flow of the algorithm. Once the foundations have been explained, the second viewpoint will analyze the algorithm by means of its control-flow. For the explanation of the control-flow, we will use the concept of top-down parsing in general to explain the algorithm using visualization techniques. Although the second viewpoint of this project will focus on visualizing GLL parsing, the proposed visualization techniques are general enough to explain and visualize top-down parsing in general. To demonstrate the visualization techniques in practice, a demo application GLLExplorer is designed explaining GLL algorithm for non left-recursive grammars.

The remainder of this thesis is structured as follows. Chapter 1 already provided a short description of the topic that will be discussed in this project. Chapter 2 provides a general introduction to parsing and motivation for this research topic. Chapter 3 will focus on the first viewpoint by showing how a traditional LL parser can be incrementally transformed to a GLL parser. The visualization of the GLL algorithm and the difficulties that one may experience when designing this visualization will be discussed in Chapter 4.
1.1 Glossary and abbreviations

In this chapter notions and abbreviations are listed and explained. Throughout the project, these definitions will be referred to in by writing the definition in italic.

1.1.1 Abbreviations

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GSS</td>
<td>Graph Structured Stack [15]</td>
</tr>
<tr>
<td>SPPF</td>
<td>Shared Packed Parse Forest [6]</td>
</tr>
<tr>
<td>GLL</td>
<td>Generalized LL-Parsing [6]</td>
</tr>
<tr>
<td>CFG</td>
<td>Control-Flow Graph [12]</td>
</tr>
<tr>
<td>LL parsing</td>
<td>(Left to Right, Left-most derivation)</td>
</tr>
</tbody>
</table>

1.1.2 Glossary

<table>
<thead>
<tr>
<th>Term</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>User</td>
<td>Person interacting with the GLLExplore system.</td>
</tr>
<tr>
<td>Descriptor</td>
<td>A quadruple (Label, GSSNode, input position, SPPFNode) as defined by Scott and Johnstone [6].</td>
</tr>
<tr>
<td>Data structure $R$</td>
<td>A set of the descriptors that still have to be processed by the GLL algorithm.</td>
</tr>
<tr>
<td>Data structure $U$</td>
<td>A set of descriptors that have been processed by the GLL algorithm.</td>
</tr>
<tr>
<td>Data structure $S$</td>
<td>A set of SPPF nodes that have been created by the GLL algorithm.</td>
</tr>
<tr>
<td>Data structure $P$</td>
<td>A set of tuples (GSSNode, input position) for which the GLL algorithm has performed a pop() statement [5].</td>
</tr>
<tr>
<td>GSS node</td>
<td>Node in a GSS labelled with a tuple (Grammar slot, input position) as defined by Tomita [15].</td>
</tr>
<tr>
<td>SPPF symbol node</td>
<td>Node in an SPPF labelled with a triple (Grammar slot, left extent, right extent) as defined by Scott and Johnstone [6].</td>
</tr>
<tr>
<td>SPPF packed node</td>
<td>Node in an SPPF labelled with a tuple (Grammar slot, split) as defined by Scott and Johnstone [6].</td>
</tr>
<tr>
<td>GSS</td>
<td>(Cyclic) Graph where every path corresponds to a stack.</td>
</tr>
<tr>
<td>GLL generation algorithm</td>
<td>Algorithm to generate a GLL parser as defined by Scott and Johnstone [6].</td>
</tr>
<tr>
<td>GLL algorithm</td>
<td>Algorithm describing how to apply generalized LL-parsing for a given input string and grammar.</td>
</tr>
<tr>
<td>GLL parser</td>
<td>Parser as constructed by the GLL generation algorithm.</td>
</tr>
<tr>
<td>GLL recognizer</td>
<td>GLL parser algorithm without the SPPF data structure [5] and without labels on the GSS edges.</td>
</tr>
<tr>
<td>Recognizer</td>
<td>Algorithm to return whether a string $I$ can be derived from some grammar $G$.</td>
</tr>
<tr>
<td>Parser</td>
<td>Algorithm to return a parse forest for a string $I$ and grammar $G$.</td>
</tr>
<tr>
<td>GLLExplore system</td>
<td>An application for the at runtime visualization of a GLL recognizer and GLL parser.</td>
</tr>
<tr>
<td>State of the GLL algorithm</td>
<td>The input pointer, current GSS node, current SPPF node, current grammerslot, the sets $R, U, P$, the GSS, and SPPF at some point in time.</td>
</tr>
<tr>
<td>SPPF</td>
<td>Data structure for the efficient storage of a parse forest [17].</td>
</tr>
<tr>
<td>Parse forest</td>
<td>Collection of parse trees.</td>
</tr>
<tr>
<td>Parse tree</td>
<td>Hierarchical data structure representing the syntactic structure of an input string according to some grammar.</td>
</tr>
<tr>
<td>Grammar</td>
<td>Set of production rules defining how to construct strings that are valid according to the syntax of some formal language.</td>
</tr>
</tbody>
</table>
**CHAPTER 1. INTRODUCTION**

<table>
<thead>
<tr>
<th>Term</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Production Rule</td>
<td>Equation of the form $A ::= l_1 \ldots l_n$ specifying that nonterminal $A$ can be substituted to a string of literals. These rules can be recursively applied to generate new strings.</td>
</tr>
<tr>
<td>Derivation</td>
<td>The construction of a string through the application of one or more production rules.</td>
</tr>
<tr>
<td>Formal Language</td>
<td>Subset of strings generated by some alphabet.</td>
</tr>
<tr>
<td>Syntax</td>
<td>Set of rules specifying whether a combination of terminals is considered to be properly structured.</td>
</tr>
<tr>
<td>Semantics</td>
<td>Set of rules specifying whether a combination of terminals is considered to be meaningful.</td>
</tr>
<tr>
<td>Alphabet</td>
<td>Set of terminals from which a string can be constructed.</td>
</tr>
<tr>
<td>String</td>
<td>Sequence consisting of one or more terminals.</td>
</tr>
<tr>
<td>Literal</td>
<td>Element from the set of terminals and nonterminals.</td>
</tr>
<tr>
<td>Terminal</td>
<td>Symbol that cannot be replaced by other symbols using the production rules of a grammar.</td>
</tr>
<tr>
<td>$\varepsilon$</td>
<td>Empty string</td>
</tr>
<tr>
<td>Nonterminal</td>
<td>Variable that is replaced by one or more terminals using the production rules of a grammar.</td>
</tr>
<tr>
<td>Symbol</td>
<td>Token as defined by some regular expression [7].</td>
</tr>
<tr>
<td>Grammar slot</td>
<td>Production rule augmented with a pivot to indicate what part of the right hand side has been recognized so far. In LR parsing, this definition corresponds to a parse item (also known as dotted-rules) [8].</td>
</tr>
<tr>
<td>Character</td>
<td>Smallest distinguishable unit of information.</td>
</tr>
</tbody>
</table>
Chapter 2

Preliminaries

2.1 Preliminaries

2.1.1 Notation

Before going into any detail, some notation will be introduced for operations on strings. Let \( \#s \) denote the number of elements in a string \( s \). Let \( \text{head}(s) \) and \( \text{last}(s) \) denote the first and last element in string \( s \) respectively. Let \( \text{tail}(s) \) represent the string \( s \) without \( \text{head}(s) \). Finally, the notation \( s = t ++ [b] \) is used to represent string \( s \) consisting of substring \( t \) followed by element \( b \).

Throughout this report, Greek letters \( \alpha, \beta, \cdots \) will be used to denote strings.

2.1.2 Language engineering

A natural way for humans to interact with their environment is by means of a language, where language can vary from a series of predefined gestures to well structured sentences according to some grammar. In computer science, the focus of language engineering is to use formal languages as a way to achieve communication between man and machine.

A formal language \( \mathcal{L} \) describes a set of sentences that are considered to be structurally correct and desired. The main idea is to use the language as a higher abstraction level for specifying commands to enhance and ease communication between man and machine. The specified sentences in turn are analysed by the computer and translated into an equivalent set of instructions that have to be executed. How this analysis can be achieved is discussed in the next sections.

2.1.3 Grammars

In order for a sentence \( S \) to be analysed correctly, we need to determine whether the sentence is valid with respect to \( \mathcal{L} \). One way of verifying this validity is by checking whether the sentence can be constructed using a grammar \( G \). In computer science, a grammar is often described as a set of production rules, that allows a string of symbols to be substituted by another string of symbols.

For the type of grammars that are of interest in this project, these rules (also known as production rules) are typically of the form \( A ::= l_1 \cdots l_n \), where \( A \) represents a variable and \( l_1 \cdots l_n \) can represent an arbitrary string of symbols and variables. The main idea behind generating string using \( G \) is to start with some initial variable and repeatedly replace variables by means of the
production rules until the resulting string only consists of symbols. In language engineering, variables such as $A$ are referred to as nonterminals whereas symbols are also known as terminals. For the distinction between terminals and nonterminals, the textual convention is used to write terminals in lower case and nonterminals in upper case.

Listing $\Gamma_1$ shows an example of a grammar consisting of three production rules. Let $S$ represent the start nonterminal of the grammar. Note that $S$ consists two production rules, indicating that there is a choice how $S$ can be substituted. The string $aabdd$ is an example of a string that can be generated by means of the production rules in $G$. This is also illustrated in Listing 2.1. String $bd$ is an example that does not occur in $L$.

$$
\begin{align*}
S &::= ASd \mid b \\
A &::= a
\end{align*}
$$

Figure 2.1: A derivation can be constructed by means of a string of substitutions.

We say that $I$ is valid with respect to $L$ if and only if there exists a sequence of production rules in $G$ such that the application of these rules on nonterminal $S$ results to the string $I$. This sequence is also known as a derivation for $I$. Strings such as $aASdd$ are referred to as sentential forms, since they still contain nonterminals. A grammar is known to be ambiguous if there exists multiple derivations for to obtain $I$.

2.1.4 Context-free grammars

In formal language theory, grammars are used to formally describe the syntax of a language. The rules specified in a grammar form the building blocks for the construction of every string derivable from the language. More formally, a grammar $G$ is a 4-tuple $(N, T, R, S)$ where $N$ represents a set of nonterminals, $T$ a set of terminals, $R$ a set of production rules, and $S \in N$ a start nonterminal. Sets $N$ and $T$ are assumed to be disjoint. An element $l \in (T \cup N)$ will be referred to as a literal. Every terminal $t \in T$ is defined by a regular expression $R$. Let $\varepsilon \in T$ be a special symbol to denote the empty string and let $\$\$ represent an “end of string” symbol. We say that a nonterminal is nullable if it is possible for that nonterminal to eventually derive $\varepsilon$.

The rules in $R$ are of the form $A ::= \alpha_1 \mid \cdots \mid \alpha_n$, where $A \in N$ and $\alpha_1, \ldots, \alpha_n$ represent strings over $(T \cup N)^*$, also referred to as the alternates of $A$. These type of rules are considered “context free”, since their left hand side does not make any assumptions about literals that are preceded or followed by the nonterminal in question. A grammar is considered “context free” if and only if all productions rules in that grammar are context-free.

We assume that for every $sym \in N$, there is exactly one production rule $A ::= \alpha_0 \mid \cdots \mid \alpha_n \in R$ such that $sym = A$. One can easily satisfy this assumption by substituting rules $A ::= \alpha_0 \mid \cdots \mid \alpha_k$ and $A ::= \alpha_1 \mid \cdots \mid \alpha_m$ for the equivalent rule $A ::= \alpha_0 \mid \cdots \mid \alpha_k \mid \alpha_1 \mid \cdots \mid \alpha_m$. 
CHAPTER 2. PRELIMINARIES

First and Follow

For the explanation of generalized LL parsing, the so called first and follow set of a literal will be used. The first set for a literal \( l_i \) is the set of all terminals that can appear as the first terminal when deriving \( l_i \). More formally, given a grammar \( G = (N,T,R,S) \) and a nonterminal \( A \), we can define the first set of \( A \) as follows:

\[
\text{First}_T(A) = \{ t \in T \mid \exists \alpha \ (A \xrightarrow{*} t\alpha) \}
\]

\[
\text{First}(A) = \begin{cases} 
\text{First}_T(A) \cup \{\varepsilon\} & \text{if } A \text{ is nullable} \\
\text{First}_T(A) & \text{otherwise}
\end{cases}
\]

The follow set of a nonterminal \( A \) is the set of terminals that can appear after \( A \) in some sentential form. Given a grammar \( G = (N,T,R,S) \) and a nonterminal \( A \), the follow set of \( A \) is defined as:

\[
\text{Follow}_T(A) = \{ t \in T \mid \exists \alpha,\beta \ (S \xrightarrow{*} \alpha At\beta) \}
\]

\[
\text{Follow}(A) = \begin{cases} 
\text{Follow}_T(A) \cup \{$\} & \text{if } A \text{ is nullable} \\
\text{Follow}_T(A) & \text{otherwise}
\end{cases}
\]

Note that the follow set of a terminal is equivalent to \( \emptyset \) since by definition a terminal can not be substituted for other literals.

2.1.5 Parsing

The previous section showed how grammars can be used to generate strings with respect to a certain language \( L \). In the world of parsing, the problem is defined the other way around where the user provides an input string \( I \) and the question is how \( I \) can be constructed according to the production rules of a grammar. If the user is only interested whether there exists a derivation for \( I \), we refer to this problem as recognizing.

Why do we need parsing? Suppose that we would like to report the subject in the sentence “The dog ate my homework”. In order for a computer to know which words to report, it must know how an English sentence in general is constructed. As a consequence, the input string as it is presented does not provide enough structural information to perform this task. So what does a parser do? The goal of a parser is to obtain this structural information by figuring out how the input string can be constructed by means of a grammar. More formally, given a grammar \( G \) and some input string \( I \), the goal of a parser is to find a sequence of production rules such that the application of these rules on the start nonterminal of \( G \) results to the string \( I \). To illustrate the latter, consider example grammar \( \Gamma_2 \)
The main idea behind top-down parsing (the type of parsing that we will be looking into) is to repeatedly apply substitutions on a start nonterminal until either no more substitutions can be applied and the input string is obtained or it is certain that the input string can not be constructed by means of the grammar. As the substitution of a nonterminal can in turn result into multiple nonterminals, the output of the parser clearly has to identify for each nonterminal which production rule must be applied in order to obtain \( I \). This syntactic structure of \( I \) traditionally is represented as a tree where the leafs in the tree represent terminals, and internal nodes represent nonterminals. The children \( v_1 \ldots v_n \) of a node \( u \) represent the application of the production rule \( u ::= v_1 \ldots v_n \).

An example of such parse tree is also illustrated in Figure 2.2.

![Figure 2.2: Schematic overview of a parser in general.](image)

The parse tree now clearly illustrates that “the dog” was the subject in our example sentence. In the old days, people tend to write these parsers by hand. Since this work can become very tedious for large grammars, parsers nowadays are often generated by means of a generator.

### 2.1.6 Scanners

Before going into any detail with respect to GLL parsing, it is first important to know that when developing a parser in general there are two ways to deal with the provided input data: with or without the use of a scanner. A scanner is a separate module which decomposes the input string \( I \) into a stream of substring (also known as tokens) according to a set of predefined patterns. All irrelevant characters, most often characters involving layout, are discarded. The scanner in turn provides the tokens to the parser to perform the actual parsing process. This is also illustrated in Figure 2.3. A scannerless parser tries to perform this tokenization together with the actual parsing.
CHAPTER 2. PRELIMINARIES

Figure 2.3: Schematic overview of a parser with a scanner. Note that the string 10 is considered as one token.

From a model-driven point of view, scanners are often desired as they increase the modularity and therefore the understandability of the resulting parser. Further note that grammars do not have to be extended with layout constructs, since layout constructs are automatically discarded by the scanner. Scannerless parsing on the other hand can be easier to implement, because such a parser does not have to make design decisions with respect to for instance token classification. Nevertheless, since the goal of this project is to explain and visualize GLL parsing as clearly as possible, there is decided to explain GLL parsing by means of a scanner.

Similarly as proposed in earlier work [18], for the implementation of a scanner, a variable `inputString` is introduced containing the full string to be parsed. The variable `pos` is used to indicate how far the input string has been parsed so far. The scanner consists of the following five functions:

1. `hasNext(Regex r)`: Returns whether `InputString`, starting from position `pos`, matches regular expression `r`.
2. `hasNext(Regex[] rs)`: Returns whether `InputString`, starting from position `pos`, matches a regular expression in the set `rs`.
3. `next(Regex r)`: Increases `pos` by the length of the substring that matches expression `r`.
4. `peek(Regex r)`: Returns the substring that matches regular `r` without adjusting `pos`.
5. `setPosition(int i)`: Reverts the input position `pos` to position to `i`.

2.1.7 Why Generalized parsing?

Ever since the first program was written, scientists have come up with efficient techniques to make parsing suitable for large programs. So people rightly ask the question why to study “yet another parsing technique”? Although 15 years ago the latter argument would definitely hold, in the era of highly evolving multilingual information systems, not only the realization of a parser but also the maintenance of one becomes important. To illustrate the latter, suppose that we have developed some programming language `P` and we want `P` to support SQL syntax. Let us assume w.l.o.g. that `P` and `SQL` both have some overlap in syntax.

One solution would be to manually embed the latest SQL syntax in the grammar of `P`. Since traditional parsing techniques do not support ambiguities, all ambiguities between `P` and `SQL`
have to be resolved upfront when modifying the grammar. As it is likely for SQL to change over
time, the grammar of $P$ has to be modified whenever SQL comes with a new version. Note that
each time the grammar of $P$ is updated, there is a risk that the disambiguation decisions that were
made in the past conflict with new language constructs in the grammar. To avoid these conflicts
inside the parser, one idea is to perform the disambiguation of the language after the parsing has
been performed. This is where generalized parsing becomes interesting.

In contrast to traditional parsing, where every grammar is assumed to be unambiguous, generalized
parsers are capable in parsing input strings for which multiple derivations exist. Since generalized
parsers are able to parse context-free grammars in general, and the union of two context-free
grammars is again known to be context-free, languages such as $P$ can now easily be extended
with other languages by inserting them as separate modules. By performing the disambiguation
outside the scope of the grammar, disambiguation decisions can now be easily maintained without
having to rewrite the entire parser. Not only does this increase the modularity of the system, it
also reduces the complexity of the resulting parser as the grammar is no longer polluted with these
decisions. Since generalized parsers support the full class of context-free grammars, techniques
such as left-recursion elimination and left-factorization are also no longer required to make the
grammar suitable for the underlying parsing technique.

2.1.8 Motivation and Research Questions

Generalized LL parsing is relative new parsing technique developed by Johnstone and Scott [5]
which (as the name suggests) generalizes on the concept of LL parsing. In contrast to other
generalized parsing algorithms such as GLR by Tomita [16], GLL parsers are known for their
straightforward implementation and clean code structure. Their resemblance with traditional LL
parsers not only makes the structure of the parser easy to understand, but also makes them very
suitable to be generated by means of code templates.

During our Honors project, we have investigated the GLL algorithm to discover new error handling
techniques within GLL parsing. Although the global idea of the algorithm was clear, we experi-
enced that when designing new extensions for GLL, it was difficult to show that these extensions
would fully cover the control flow of the algorithm. The high learning curve of the algorithm boils
down to the following two points:

- **Sharing**
  In order to support arbitrary context-free grammars and multiple derivations, a GLL parser
  introduces several data structures to make sure that every possible interpretation of the
  input string is discovered. Although the purpose of every data structure is explainable and
  intuitive, when looking a the GLL parser as a whole it is difficult to relate the concepts that
  are happpening in an LL parsing back to these data structures. As a consequence, in order to
  fully understand how the algorithm works, it is important to see when and how certain data
  structures become important. Knowing why these data structures have to be introduced in
  the first place allows helps you to identify what kinds of problems one may run into when
designing extensions on top of GLL.

- **Progress and control flow**
  In contrast to traditional LL parsing, a GLL parser tries to parse multiple derivations in
  parallel. It does this so by dividing the parse work into chunks of work that derivations
  have in common. Although every chunk of work is understandable on its own, the resulting
  control flow of the algorithm is still rather abstract. This makes the progress of a GLL parser
  hard to follow over time. A GLL parser uses a set of operations to determine which chunk of
  work to parse next. Although the working of these statements by themselves are clear, when
  inspecting the implementation of these statements, you will notice that some occurrences of
  a statement are hard to explain from the viewpoint of these chunks.
The main objective of this thesis is to reduce the learning curve of the algorithm by presenting the previous two issues in a different way. In order to reach this goal, the following research questions have been defined.

**Question 1**

**How can we obtain a GLL parser by starting from a traditional left-recursive descent parser?**

The first question tries to discover what makes a Generalized LL parsing a generalization of LL parsing. Unfortunately, the current literature only describes the GLL parser as a whole which makes it difficult to reason about the underlying foundation of the algorithm. Understanding the relationship between the two parsing techniques helps us to understand which problems have to be tackled when designing new extensions and visualizations for GLL parsing.

**Question 2**

**Can we explain the control flow of a GLL parser by means of visualization techniques? If so, what are the difficulties one may encounter when designing such visualization?**

In order to understand how the algorithm can divide the parsing of the input string into chunks of work, the underlying control flow of the algorithm must be understood. By providing a step-by-step illustration of how a GLL parser deals with these chunks, the user is able to see why certain steps have to be performed in the first place. One can imagine that the visualization of the chunks that have been processed can provide the user an intuition how far every derivation is constructed with respect to the input string. In order to achieve this, however, we somehow have to know how to recombine these chunks together to obtain this information.

As a preparation towards solving the first question, we will first look at a very simple parsing technique known as recursive descent parsing. This type of parser is typically designed for LL grammars, indicating that (indirect) left-recursion and ambiguities are not supported by this technique. The relationship between LLRD parsing and GLL parsing will be explored by showing how a LLRD parser can be incrementally transformed to a GLL parser.
Chapter 3
Generalized LL parsing

3.1 Recursive descent parsing: the roots

The fundamental idea behind LLRD recognizing is the concept of divide and conquer, where the problem of parsing some input string $I$ using grammar $G$ is solved by the (recursive) invocation of one or more functions. In general, the problem an LLRD recognizer focuses on is:

(i) Given an input string $I = a_1 \cdots a_m$ and some starting position $i$ ($0 \leq i \leq m - 1$), does there exist a derivation for the substring $I' = I[i \ldots j]$ starting from some nonterminal $A$? ($i \leq j \leq m - 1$)

A recognizer is considered successful if it can solve (i) for $i = 0$ and $j = m - 1$. An LLRD recognizer solves this problem by creating a function $A()$ for every nonterminal $A$ in the grammar, where the goal of such function is to recognize the right hand side of the corresponding production rule $A ::= \alpha_1 | \cdots | \alpha_n$. The body of function $A()$ is generated by sequentially checking the occurrence of literals in $I$ according to alternates $\alpha_1, \ldots, \alpha_n$. Whenever the recognizer encounters a nonterminal $B$, it is aware that the recognition of $B$ is again a subproblem of the form (i). Hence function $B()$ is invoked to solve this subproblem after which the recognizer continues with the next literal in the alternate.

In the situation where $n > 1$, every alternate is considered sequentially. This is typically modelled by means of an else-if clause for every alternate. In the situation where multiple alternates are able to recognize a substring of $I$, the recognizer has to make sure that as soon as an alternate does not manage to recognize $I$ properly, all work performed by this alternate is reverted. This is also referred to as backtracking. Since a GLL parser solves backtracking in a different way, for the sake of simplicity, we will focus on the structure of an LLRD that does not require backtracking.

\[
S ::= bS \mid ASd \mid \varepsilon \\
A ::= a
\]

(Γ₃)

Listing 3.1 shows the resulting code of an LLRD recognizer for example Grammar Γ₃. Note that the recognizer does not require any backtracking, since every alternate starts with the recognition of a different terminal. Grammars that do not require backtracking and are not left-recursive are also known as LL(1) grammars, since they only require one symbol lookahead to find the only right alternate. The reason why left-recursive grammars are not LL(1) will become clear in Section 3.4.
Variable \texttt{scanner} represents a scanner as defined in section 2.1.6, globally maintaining the input position of the recognizer. Function \texttt{S'}() represents a goal function to test whether the recognizer managed to recognize \texttt{I} entirely. When looking at the execution of the recognizer, you will notice that every line of the program represents a certain state how much of a production rule has been considered. In the work by Scott and Johnstone [6], the state of a production rule is represented by a "·" and is also referred to as a grammar slot. Listing \texttt{Γ₃} shows how the different grammar slots correspond to locations in the recognizer code. Note that for the \(\varepsilon\) alternate, the statement \texttt{scanner.hasNext("")} is superfluous as it is always possible to detect the empty string in the input. But just to illustrate that an \(\varepsilon\) alternate can be treated the same as the recognition of any other alternate, the statement is not removed.

From the viewpoint of the code, the solution is very clean, as there is a one-to-one correspondence between the structure of the grammar and the resulting recognizer. In Section 3.4 we will however see that the algorithm as such is not sufficient enough to support arbitrary context-free grammars.

### 3.2 Knowing where to go next: stacks

Whenever an LLRD recognizer encounters a nonterminal \(B\) in a production rule \(A := αBβ\), an LLRD recognizer applies the production rule corresponding to \(B\) after which every literal in that production rule is recognized. Since the recognition of \(B\) is a new subproblem that in turn may require the recognition of other nonterminals in order to succeed, an LLRD recognizer solves this
problem in a separate function $B()$. Do note however that after $B$ has been recognized, the recognizer somehow has to know how to return to the original problem of recognizing $A$, as it is possible for the production rule $A$ to have literals after $B$ (i.e. $\beta \neq [ ]$). Since these literals have to be recognized as well, the recognizer has to remember the location in the production rule it should continue after $B$ has been recognized. This return location corresponds to the grammar slot $A ::= aB \cdot \beta$. Once the recognizer has managed to recognize $A$ successfully, it can use the grammar slot to jump to the right location in the code. Since the recognition of $B$ in turn could lead to new nonterminals, grammar slots are stored in a LIFO buffer (i.e. stack). Note that literals are considered in the order in which they occur in the production rule. As a consequence, whenever a nonterminal is substituted by some rule, those literals have to be considered first, thereby making a stack a suitable data structure to solve this problem. Figure 3.1 illustrates the working of an LLRD stack for input string $I = aad$ and grammar $\Gamma_3$. Note that in a top-down parser a stack is only used to remember the remaining literals that have to be considered after the recognizer has recognized a nonterminal. Therefore note that all grammar slots in the stacks have their \cdot occurring after a nonterminal.

Figure 3.1: Illustration of LLRD stack evolving over time. The gray stack entries represent the entries that are popped from the stack.

Listing $\Gamma_3$ illustrates that grammar slots before and after the execution of a nonterminal occur consecutive in the code. Since the recognition of a nonterminal is solved by means of a (recursive) invocation, for an LLRD recognizer, the code location of a grammar slot corresponds to the pop address that is pushed on top of the call-stack. This is also the main reason why an LLRD recognizer does not have to maintain a separate data structure for storing the literals that still have to be recognized. Besides, the great benefit of using the call-stack as the underlying data structure is that jumping from one code location to another can happen elegantly by means of function calls.
3.3 Towards a parser

In order to extend an LLRD recognizer into a parser, instead of reporting whether there exists a derivation for input $I$ given grammar $G$, the recognizer now also has to show how $I$ is structured with respect to that derivation. So, whenever the recognizer recognizes a literal in a production rule, it has to store which part of $I$ corresponds to that literal.

The solution is to augment the recognizer with an additional data structure, where the recognition of every terminal results into the creation of a leaf node and the recognition of a nonterminal results in the creation of an internal node. Whenever the recognizer manages to recognize a literal $l_i$ in production rule $A ::= l_1 \cdots l_n$, the parse node for $l_i$ is attached to the parse node corresponding to $A$. As soon as the recognizer managed to recognize an entire alternate of $A$, the root of the resulting parse tree is propagated to the original parsing problem by means of the stack. Listing 3.2 shows the code of the resulting LLRD parser.

```java
Scanner scanner = new Scanner(I);

SNode S'( ) {
    if (scanner.hasNext(test(S))) {
        SNode tree = S();
        if (scanner.hasNext("$")) {
            return tree;
        } else {
            error();
        }
    } else {
        error();
    }
}

SNode S() {
    SNode root = new SNode();
    if (scanner.hasNext("b")) {
        scanner.next("b");
        root.addChild(new bNode("b"));
        if (scanner.hasNext(test(S))) {
            root.addChild(S());
            return root;
        } else {
            error();
        }
    } else if (scanner.hasNext(test(A))) {
        root.addChild(A());
        if (scanner.hasNext(test(S))) {
            root.addChild(S());
            if (scanner.hasNext("d")) {
                root.addChild(new dNode("d"));
                return root;
            } else {
                error();
            }
        } else if (scanner.hasNext("a")) {
            return new ANode("a");
        } else {
            error();
        }
    }
}

ANode A() {
    ANode root = new ANode();
    if (scanner.hasNext("a")) {
```

Exploring and visualizing GLL parsing
CHAPTER 3. GENERALIZED LL PARSING

Listing 3.2: Pseudocode for an LLRD parser for grammar \( \Gamma_3 \)

```java
50 scanner.next("a");
51 root.addChild(new aNode("a"));
52 return root;
53 }
54 else{
55   error();
56 }
```

Since a function now returns a parse node instead of a boolean, we can no longer use the return value of a function to check whether a nonterminal is capable in parsing the next part of the input string. One way of solving this issue is to look ahead in the input string and perform this check before the invocation of the function can take place. In order to do so, the set \( \text{test}(A) \) is introduced which returns the set of terminals that can appear when deriving nonterminal \( A \). Figure 3.2 shows a more formal definition how the test set can be defined for any literal \( l \).

\[
\text{test}(l) = \begin{cases} 
\text{First}(l) \cup \text{Follow}(l) & \text{if } \varepsilon \in \text{First}(l) \\
\text{First}(l) & \text{if } \varepsilon \not\in \text{First}(l) 
\end{cases} \quad \text{for all } l \in (T \cup N)
\]

Figure 3.2: Definition for providing a 1-symbol lookahead check in the input string

First, note that a nonterminal \( A \) can only parse a part of the input string if and only if there is an alternate \( \alpha_i := l_1 \cdots l_n \) in \( A ::= \cdots | \alpha_i | \cdots \) such that \( \text{scanner.hasNext(test(l_1))} \) returns true. Second, note that the test set of a terminal \( t \) trivially equals \( \{t\} \). Hence, for the sake of readability, \( \text{test}(t) \) is immediately replaced by \( t \).

3.4 Issues LLRD

Although an LLRD parser is powerful enough to support quite some grammars, for context-free grammars in general, however, there are two main difficulties with this technique.

1. Call-stack
   The previous section illustrated how the call-stack can used to jump to the right code whenever a nonterminal has been parsed. The main disadvantage of using the call-stack however is that the data structure is acyclic implying that any repetitive patterns in the stack cannot be captured compactly by means of loops. To illustrate the latter, consider the LLRD recognizer in Listing 3.3 a) for example grammar \( \Gamma_4 \)

\[
E ::= E + E \mid 1 
\]  \( (\Gamma_4) \)
Although at first sight the code in Listing 3.3 a) looks innocent, when executing the problem for the input string 1 + 1 you will notice that the application does not terminate. In every recursive call of $E()$ the LLRD recognizer will push the return location $E \cdot +1$ on top of the stack thereby not noticing that there is no progress in the input string. Hence, in order to support these so called left-recursive grammars, the stack data structure has to be replaced by a graph thereby making the call-stack no longer suitable to perform the code jumping.

2. Non-determinism: Ambiguities and Backtracking

Section 3.3 illustrated how the stack can be used to propagate parse results of nonterminals so that they can be combined together in a parse tree. Since an LLRD parser only maintains one stack, it implicitly assumes that for every input string there exists at most one derivation. In the situation however where multiple alternates can be valid at the same time (such as in backtracking or ambiguities), the parser may be unable to decide which of these alternates will result in a valid parse tree. Hence, for the parsing of context-free grammars in general, the parser must be able to store the progress of multiple derivations at the same time. To illustrate the latter, consider the slightly different grammar $\Gamma_5$

$$S ::= aS \mid ASd \mid \varepsilon$$
$$A ::= a$$

$$(\Gamma_5)$$

Figure 3.4 shows an example input for which there exist multiple parse trees for the same input string. Grammar $\Gamma_5$ is an example of an ambiguous grammar. Since the focus of generalized parsing is to report all derivations, the LLRD parser has to be augmented with multiple stacks in order to report all parse trees for ambiguous grammars.

```
Scanner scanner = new Scanner(I);

boolean E(){
    if(E()){
        if(scanner.hasNext("+")){
            scanner.next("+");
            return true;
        }
    } else if(scanner.hasNext("1")){
        return true;
    }
    return false;
}
```

Figure 3.3: a) Pseudocode LLRD parser for Grammar $\Gamma_4$. b) Illustration of LLRD stack.

Figure 3.4: For the input string $aad$ and grammar $\Gamma_3$ multiple derivations are possible.
We will solve the first issue by replacing the call-stack by a custom data structure that allows cycles. For the second issue, we will present a way to compactly generate all resulting parse trees for some input string.

### 3.5 Explicit stack

A first step towards the removal of the call-stack is to introduce procedures for the operations that happen implicitly inside the call-stack. Whenever a function $p$ invokes a function $f$, the call-stack pushes the return address of $p$ on top of the stack. After the execution of the procedure has finished, the return address is removed from the stack and used to modify the instruction pointer of the recognizer. To make these operations explicit for the addition and removal of grammar slots, functions `create()` and `pop()` are introduced. To illustrate the concept, Listing 3.3 shows how the recognizer for grammar $\Gamma_3$ can be augmented to maintain grammar slots without implicitly storing it on the call-stack.

```java
Scanner scanner = new Scanner(1);
Stack s = new Stack();

boolean S'() {
    S();
    return scanner.hasNext("$");
}

void S() {
    if (scanner.hasNext("b")) {
        scanner.next("b");
        if (scanner.hasNext(test(S))) {
            create(S ::= b S·);
            S();
            pop();
        } else {
            error();
        }
    } else if (scanner.hasNext("a")) {
        create(S ::= A·S d);
        A();
        if (scanner.hasNext(test(S))) {
            create(S ::= A S·d);
            S();
            if (scanner.hasNext("d")) {
                scanner.next("d");
                pop();
            } else {
                error();
            }
        } else {
            error();
        }
    } else if (scanner.hasNext("")) {
        scanner.next("");
        pop();
    } else {
        error();
    }
}

void A() {
    scanner.next("a");
    pop();
}

void create(GrammarSlot g) {
    stack.push(g);
}
```

Exploring and visualizing GLL parsing
Since we no longer want to use the call-stack for the storage of data, propagating data by means of the return keyword is no longer possible. As a consequence, all functions in Listing 3.3 are replaced by methods. Since a method cannot be used to test the validity of an alternate, similarly as in section 3.3, a lookahead check by means of the \texttt{test()} function is introduced. After all, note that it is only useful to push a grammar slot on top of the stack if the recognizer knows upfront that the nonterminal has a chance of succeeding. To ensure that the recognizer terminates once a literal could not be recognized, the function \texttt{error()} is introduced.

Because of the fact that an LLRD recognizer does not have to construct a parse tree, the implementation of the \texttt{create statement} is simply realized by adding the desired grammar slot to the \texttt{LIFO queue}. The \texttt{pop()} statement however is slightly more complicated. A direct consequence of maintaining our own stack data structure outside the call-stack is that we are now responsible to \texttt{jump} to the right code locations whenever a \texttt{pop statement} is invoked. In order to realize this, a function \texttt{Goto()} is introduced that given a grammar slot $g$, jumps to the code location corresponding to that grammar slot. But how to implement this method?

Because of the fact that not all programming languages such a Java support \texttt{goto} statements, jumping to arbitrary locations in the code can be hard if not impossible to implement properly. Besides, the usage of such statement in practice is often considered to be a bad design decision as it can significantly decrease the understandability and readability of the resulting code. Hence, we have to come up with a different way to realize this code jumping. One solution as presented in earlier work is to divide the code of the recognizer into smaller functions such that the jump to a certain grammar slot corresponds to a function invocation. The realization of this decomposition will be explained in the next section.

### 3.5.1 GLL Blocks

Recall from Section 3.2 that a stack only pushes a grammar slot $B ::= \alpha A \cdot \beta$ on top of the stack before a nonterminal $B$ is about to be recognized. Since that grammar slot represents the code location after $B$, it is important to split the code in such a way that $\alpha B$ and $\beta$ occur in separate functions. This way, whenever grammar slot $B ::= \alpha A \cdot \beta$ is popped, the recognizer can simply return to the original problem by invoking the function for $\beta$. In order to define this decomposition properly, we introduce the notion of a GLL block [18].

**Definition** A GLL block in alternate $\alpha$ is a string $s$, such that:

\[
s = \begin{cases} 
\varepsilon & \text{if } \alpha = \varepsilon \\
\text{GLLBlock}(s) & \text{otherwise}
\end{cases}
\]

String $s$ satisfies \texttt{GLLBlock}(s) if and only if $s$ is of the shape $t \rightarrow \text{[last(s)]}$, and:

- $s$ is a substring of $\alpha$
- $\text{head}(s)$ is not preceded by a terminal.
• last(s) is either:
  - a nonterminal, or
  - a terminal that is *not* followed by any literal.

• t is a string of terminals.

To illustrate the concept of a GLL block consider Figure 3.5, showing the GLL blocks for alternate 
\[ E ::= abEFeE \]

Figure 3.5: Every alternate can be decomposed into one or more GLL blocks.

Here, the alternate consists of four GLL blocks, namely blocks “abE”, “F”, “eE”, and “” respectively. First, note that there is an empty block occurring at the end of the alternate, since the last literal is a nonterminal. Second, note that GLL blocks are defined in such a way that the concatenation of these blocks together form the whole alternate again.

By creating a separate function for every GLL block, the recognizer is able to jump to different positions within an alternate \( \alpha \) by means of function invocations. As a consequence, whenever a nonterminal \( A \) in \( \alpha \) was recognized successfully, the return statement is now realized by invoking the function that continues the recognition of \( \alpha \) after \( A \). Since a nonterminal always occurs at the end of a GLL block, this function corresponds to the next GLL block in \( \alpha \). For example, after GLL block “abE” was recognized successfully, the recognizer uses the stack to remember it must continue at grammar slot \( E ::= abE \cdot FeE \). Since every GLL block corresponds to a separate function, the recognizer can simply return to this grammar slot by invoking the function corresponding to GLL block “F”. Listing 3.4 illustrates the LLRD recognizer after applying the functional decomposition.

```java
Scanner scanner = new Scanner(I);
Stack s = new Stack();

boolean S() {
  S();
  return scanner.hasNext("S");
}

void S() {
  if (scanner.hasNext("b")) {
    Goto(S ::= b S);
  } else if (scanner.hasNext(test(A))) {
    Goto(S ::= A S d);
  } else if (scanner.hasNext("")){
    Goto(S ::= ε);
  } else {
    error();
  }
}

void b S() {
  scanner.next("b");
  if (scanner.hasNext(test(S))){
    create(S ::= b S ·);
    S();
  } else{
    error();
  }
}

void S ::= b S · () {
```

Exploring and visualizing GLL parsing

19

CHAPTER 3. GENERALIZED LL PARSING
void S ::= A · S d() {
    create(S ::= A · S d);
    A();
}

void A() {
    if (scanner.hasNext("a") ) {
        Goto(A ::= · a);
    } else {
        error();
    }
}

void A ::= · a() {
    scanner.next("a");
    pop();
}

void S ::= A · S d() {
    if (scanner.hasNext(test(S)) ) {
        create(S ::= A S · d);
        S();
    } else {
        error();
    }
}

void S ::= · ε() {
    scanner.next("a");
    pop();
}

void S ::= A S · d() {
    if (scanner.hasNext("d") ) {
        scanner.next("d");
        pop();
    } else {
        error();
    }
}

void create(String slot) {
    stack.push(slot);
}

void pop() {
    Goto(stack.pop());
}

void Goto(String slot) {
    switch(slot) {
    case S ::= A · S d:
        S ::= A · S d();
        break;
    case S ::= A S · d:
        S ::= A S · d();
        break;
    case S ::= b S · :
        S ::= b S · ();
        break;
    case A ::= · a:
        A ::= · a();
        break;
    case S ::= · b S:
        break;
    }
}

Exploring and visualizing GLL parsing
A direct consequence of the functional decomposition is that, in contrast to an LLRD recognizer, a function is no longer responsible for the recognition of an entire alternate. The recognition of for instance \( S ::= A \ S \ d \) must now be realized by invoking \( S ::= \cdot A \ S \ d() \), \( S ::= A \cdot S \ d() \), and \( S ::= A \ S \cdot d() \) sequentially (instead of just \( S() \)). The code decomposition however allow us to implement the Goto() function by means of a simple switch statement. Although the further decomposition of a GLL block is definitely possible, note that it can significantly increase the number of functions thereby creating more computational overhead in the algorithm. Earlier work [18] has shown that a more detailed decomposition however can become interesting when looking at error handling techniques within the GLL parsing [18].

Be aware that functions \( S() \) and \( A() \) do not longer perform any recognition with respect to the input string, but are used to test which of the alternates are able to recognize the input string. Since these functions focus on the validity of alternates rather than the recognition of one, functions \( A() \) and \( S() \) will be referred to as alternate functions. The functions for the grammar slots will be referred to as match functions, since they are responsible for the actual recognition of the input string with respect to parts of the grammar.

### 3.5.2 Labelling scheme

Because of the fact that most programming languages do not support spacing nor unicode symbols in their function names, a labelling scheme is required to rename the grammar slot functions properly. The labelling scheme that is proposed in earlier work [18] uses the following two conventions.

1. Alternate functions \( A ::= \alpha_0 \ | \cdots | \alpha_n \): these functions are renamed to \( A() \)

2. Match functions \( A ::= \beta \cdot \gamma \): let \( i \) denote the number of nonterminals in \( \beta \). let \( j \) denote the \( j \)th alternate in \( A ::= \alpha_0 \ | \alpha_j \ | \alpha_n \) such that \( \alpha_j = \beta \cdot \gamma \). These functions are renamed to \( \sigma \) by means of the function:

\[
\sigma = \begin{cases} 
    A_j() & \text{if } i = 0 \\
    A_j \cdot j() & \text{otherwise}
\end{cases}
\]

Table 3.1 shows the relationship between the function names in Listing 3.4 and their corresponding labels.

---

Listing 3.4: Pseudocode LLRD recognizer for Grammar \( \Gamma_3 \) after decomposing the code into GLL blocks.
CHAPTER 3. GENERALIZED LL PARSING

<table>
<thead>
<tr>
<th>Grammar slot</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>S ::= · b S</td>
<td>S0()</td>
</tr>
<tr>
<td>S ::= b S ·</td>
<td>S0_0()</td>
</tr>
<tr>
<td>S ::= · A S d</td>
<td>S1()</td>
</tr>
<tr>
<td>S ::= A · S d</td>
<td>S1_0()</td>
</tr>
<tr>
<td>S ::= A S d</td>
<td>S1_1()</td>
</tr>
<tr>
<td>S ::= · ε</td>
<td>S2()</td>
</tr>
<tr>
<td>A ::= · a</td>
<td>A0()</td>
</tr>
<tr>
<td>S ::= b S</td>
<td>A S d</td>
</tr>
<tr>
<td>A ::= a</td>
<td>A()</td>
</tr>
</tbody>
</table>

Table 3.1: The relationship between grammar slots and their corresponding function labels.

An application of the labelling is illustrated in Listing 3.5. Using the new labelling scheme, the alternate S ::= A S d is recognized by invoking functions S1(), S1_0() and S1_1() sequentially. The algorithm for generating the entire GLL parser using this labelling scheme is defined in Section 3.9.

3.5.3 Centralized invocation

Although the code in Listing 3.4 no longer requires the call-stack to push and pop grammar slots, there is still a fundamental issue with the recognizer. Note that the Goto() statement is always invoked within the scope of another function, indicating that the call-stack in a way is still interfering with the new control flow. Note that this interference does not alter the outcome of the recognizer, since by structure of the code, a Goto() statement is always the last statement that is performed. Be aware however that the nested invocation of Goto() and pop() statements will make the call-stack grow unnecessarily. Figure 3.6 shows an illustration of how the call-stack of the recognizer in Listing 3.4 can look like when trying to recognize the string ad$.

Whenever an alternate has been parsed successfully in Listing 3.4, the pop() statement recursively invokes a Goto() statement that continue with the parsing of the remaining alternate. Do note however that functions Goto(S ::= · A S d), A(), Goto(S ::= · a), and pop() are finished but remain on the call-stack until the Goto(S ::= A · S d) function has finished. So, ideally, we would like function S’() to invoke this Goto() statement so that the rest of the statements can be removed from the call-stack.
To make sure that the call-stack does not unnecessarily grow in the implementation, recursive
invocations of functions have to be reduced to a minimum. One way to solve this problem is to
centralize the invocation of Goto() in a separate loop which will be referred to as $S'$. In order to
know which grammar slot the Goto() function must jump to, a global variable $g$ is introduced.
Instead of invoking the Goto() statement inside pop(), the popped grammar slot is now stored
in $g$. Listing 3.5 shows the result after applying the labelling scheme for the functions and the
centralized invocation of the Goto() statement.

```java
Scanner scanner = new Scanner(1);
Stack s = new Stack();
GrammarSlot g;

boolean S'() {
    create(⊥);
    S();
    while (¬stack.isEmpty()) {
        Goto(g);
        return scanner.hasNext("$");
    }
}

void $S() {
    if (scanner.hasNext("b")) {
        create($b := b S$);
        pop();
    } else if (scanner.hasNext(test(A))) {
        create($S := A S d$);
        pop();
    } else if (scanner.hasNext("a")) {
        create($S := ε$);
        pop();
    } else {
        error();
    }
}

void $S_0() {
    scanner.next("b");
    if (scanner.hasNext(test(S))) {
        create($S := b S$);
        S();
    } else {
        error();
    }
}

void $S_00() { pop(); }

void $S_1() {
    create($S := A S d$);
    A();
}

void A() {
    if (scanner.hasNext("a")) {
        create(A := a);
        pop();
    } else {
        error();
    }
}

void A0() {
    scanner.next("a");
    pop();
}
```

Exploring and visualizing GLL parsing 23
Listing 3.5: Pseudocode LLRD recognizer for Grammar $\Gamma_3$ with centralized Goto() invocations.

Function $S'$() now contains a while loop which is responsible for the Goto() invocations. Since LL(1) grammars by definition can have at most one valid alternate at the same time, the Goto() statements in the alternate functions can simply be replaced by pushing the grammar slot of the valid alternate on top of the stack. By initially invoking the alternate function $S$, grammar slot $g$ is set to the beginning of the first valid alternate. To ensure that the application terminates properly, the stack initially contains a dummy element $\bot$ representing the bottom of the stack. From the viewpoint of a grammar slot, note that this element corresponds to the fictive grammar slot $S'$ ::= $S \cdot$. Since this element does not correspond to any function invocation, it is certain that the stack will no longer grow once this element has been popped. The fact that this element eventually is popped follows from the observation that at the end of every alternate a pop() statement is
invoked.

The correctness of the transformation follows from the fact that variable \( g \) is only modified by a \texttt{pop()} statement and a \texttt{pop()} statement always occurs at the end of a function. As a consequence, after a \texttt{pop()} statement has been executed, the code immediately returns to the while loop in \( S' \) after which the stored value in \( g \) is used by a \texttt{Goto()} statement. The result of the transformation is that the depth of the the call-stack no longer depends on the number of \texttt{Goto()} statement that are invoked.

Two small remarks with respect to the resulting code. First, note that every alternate function has a \texttt{create()} statement directly followed by a \texttt{pop()} statement. Since the sequence of a \texttt{create(slot)} and a \texttt{pop()} statement creates an element after which it is directly removed, this is equivalent to setting variable \( g = \text{slot} \). Second, note that line 42 in \( S1() \) does not state \texttt{scanner.hasNext(test(S))}. Be aware however that this check has already been performed at line 14 of function \( S() \). Keep in mind that function \( S() \) is the only function that will ever invoke \( S1() \). When presenting the algorithm for the generation of GLL parsers, this optimization step will be taken into account.

### 3.6 Multiple derivations

The previous section showed a step-by-step guide on how to replace the call-stack of an LLRD recognizer with our own stack data structure for LL(1) grammars. Towards the realization of a recognizer for arbitrary context-free grammars, however, ambiguous grammars have to be supported. Since grammar \( \Gamma_3 \) is not ambiguous, let’s consider the slight modified grammar \( \Gamma_5 \) from now on. In order to illustrate the changes that have to be made to support context-free grammars in general, let us consider the recognizer code for grammar \( \Gamma_5 \) after the call-stack has been removed. This corresponds to the code in Listing 3.5 where terminal \( b \) has been replaced with \( a \).

To make this recognizer work for context-free grammars, an extension is required that allows the recognizer to support the recognition of multiple derivations. One idea to realize this is to maintain a separate stack for every derivation that is likely to succeed. Depending on the derivation the recognizer wishes to look into, the stack data structure is replaced by the stack corresponding to that derivation. Listing 3.6 shows how this idea can be realized.

```java
Scanner scanner = new Scanner(I);
Stack stack;
Set< GrammarSlot , Stack , Int > R;
GrammarSlot g;

void init(){
    Stack s = new Stack(⊥);
    R.add(s, 0);
}

boolean S'() {
    init();
    while(!R.isEmpty()){
        (slot, st, i) = R.pop();
        stack = st;
        scanner.setPosition(i);
        g = slot;
        Goto(g);
    }
    return S([ ], #I) has been processed;
}
```

Exploring and visualizing GLL parsing 25
The main idea is to maintain a set of derivations that have a chance of succeeding. Since the progress of one derivation with respect to the input string can be different from the other, the idea is to store the input position $i$ along with every derivation. Similarly, in order to know where the algorithm must continue next time when considering this derivation, the popped grammar slot $g$ is also stored together with the derivation. In essence, we can summarize the progress of a derivation as a triple $(g, stack, i)$. In the work by Scott and Johnstone [5], this triple is referred to as an elementary descriptor.

Now that we are considering context-free grammars in general, note that it is possible for a nonterminal to have multiple alternates that are valid with respect to the input string. To capture this non-determinism, the else if-statements in the alternate functions are replaced by separate if-clauses. Since the recognizer does not know upfront which alternates will eventually lead to a successful derivation, it simply tries every alternate as if it will lead to a new successful derivation. Variable $R$ in the code stores all derivations that have to be considered. Once a derivation $(slot, s, i)$ is chosen and removed from this set, grammar slot $g$, the current stack and input position of the algorithm are set according to the values of $slot$, $s$, and $i$ respectively. This enables the algorithm to switch between derivations in a flexible way. Since every derivation stands on its own and has to be considered eventually, note that the order in which the derivations are considered does not matter.

As soon as the recognizer encounters a nonterminal $A$, the derivation in progress is duplicated for every alternate $\alpha$ that has a chance to recognize $A$. Since every alternate is assumed to lead to a successful derivation, non-determinism can be modelled by replacing the derivation $(slot, s, i)$ with a new triple $(\cdot, s, i)$ for every alternate $\alpha$. For the creation of a triple, the function $\text{add()}$ is introduced. Recall that whenever a nonterminal is recognized successfully, the $\text{pop()}$ statement is used to obtain the next grammar slot where the derivation must continue. Since a triple now

```java
void S() {
    if (scanner.hasNext("b")) {
        add(S ::= · b S);
    }
    if (scanner.hasNext(test(A))) {
        add(S ::= · A S d);
    }
    if (scanner.hasNext("")) {
        add(S ::= · ε);
    }
}

void A() {
    if (scanner.hasNext("a")) {
        add(A ::= · a);
    }
}

void add(String slot) {
    Stack s' = copyStack(stack);
    s'.push(slot);
    int i = scanner.getPosition();
    R.add((s', i));
}

void pop() {
    Grammarslot slot = stack.pop();
    add(slot);
}
```

Listing 3.6: Pseudocode LLRD recognizer for Grammar $\Gamma_3$ supporting multiple derivations.
describes the stack, input position, and grammar slot of a derivation, the derivation \((\text{slot}, s, i)\) has
to be updated with the popped grammar slot. As a consequence, the pop() statement now also
contains an add() statement to ensure that the recognizer will continue to recognize this derivation
in progress after \((\text{slot}, s, i)\) has been consumed.

If the recognizer did not manage to recognize the literals in grammar slot \(\text{slot}\) successfully, the
derivation in progress turned out to be invalid and the triple \((\text{slot}, s, i)\) is dropped. To ensure that
every derivation has its own stack, the function \text{copyStack()} is used to duplicate the active stack.
Be aware that the implementation of this function is no longer a problem, since the stack data
structure is no longer maintained inside the call-stack. Once all derivations are finished \((R = \emptyset)\),
the recognizer can report the success of the recognition by verifying whether there was a triple
that managed to recognize the whole input string and whose stack only contains the \(\bot\) element.

### 3.6.1 Optimizing GLL: GSS

The previous transformation illustrated how an LLRD recognizer can support multiple derivations
by summarizing the progress of a derivation as a descriptor. Although the methodology in general
works well, there are still two issues to be solved in order to obtain a full-fledged GLL recognizer:

1. **Stack duplication**
   The code in Listing 3.6 duplicates the stack data structure each time a descriptor is cre-
ated. Besides the fact that data structure copying in general is quite expensive, one can
imagine that the amount of overlap between stacks is often quite large. Why is this relev-
ant? Unfortunately, there are context-free grammars for which the number of stacks can
become exponential (e.g. Grammar \(\Gamma_6\)). In order to make GLL parsing practical for these
grammars as well, any form of sharing between the stacks is required to significantly reduce
the overhead in computation and memory when duplicating stacks.

   \[
   S ::= SS | a | \varepsilon \quad \text{(}\Gamma_6\text{)}
   \]

2. **Left-recursion**
   The current solution still uses stack data structures for the storage of grammar slots. Recall
however from Section 3.4 that the recognizer as such still does not provide support for left-
recursive grammars. In order to support left-recursive grammars, the stack data structures
have to be replaced with a graph so that any form of repetition can be discovered as early
as possible.

The GLL recognizer solves both issues by combining all stack data structures into one centralized
graph structure. This data structure is also known as a slight modified version of Tomita’s Graph
Structured Stack (in short GSS). Figure 3.7 shows the main idea behind this approach. Note
that in the resulting graph, the first stack can be represented by means of a reference to node
\(A ::= \alpha \cdot \beta\). The other two stacks both start at node \(C ::= \rho \cdot \omega\).
Although this solution already significantly reduces the amount of duplication, note that storing the grammar slots alone are not sufficient to capture left-recursion. In contrast to “regular” recursion, left-recursion pushes grammar slots on the stack without making any progress in the input string. Hence, for the distinction between the two, not only the grammar slot but also the input position in the string when the grammar slot is created is necessary to detect any repetitive patterns. Figure 3.8 illustrates how the actual GSS uses this information when considering regular-recursion versus left-recursion.

One interesting property of the GSS is that once an element has been added to the graph, it will never be removed. In contrast to the traditional stack data structure, where popped element are simply removed from the stack, the GSS rather returns a reference to the new top element of that stack. The main reason for doing so is that it may still be possible for the popped element to be used by a different stack when considering other derivations. Although this seems nothing more than an efficiency step, note that the reuse of GSS nodes and edges allows the GSS to detect any overlap between old and new stacks. In Section 3.6.3 we will show how this information can be used to avoid any duplicate work when parsing multiple derivations.
3.6.2 Descriptors

Now that we have managed to capture left-recursion inside our graph data structure, we will show how the recognizer can be augmented to use the GSS. Instead of storing an entire stack per descriptor, a descriptor now maintains a reference to a GSS node \( \langle g, i \rangle \) representing all stacks with their top element equal to grammar slot \( g \).

Note that the introduction of the GSS changes the meaning of a descriptor drastically, since a GSS node can refer to more than one stack. As a consequence, a descriptor no longer represents a point of continue for a single derivation in progress, but for a collection of them. Stated differently, a descriptor can now be seen as a “job description”, where the “job” is to recognize a string of literals starting from a certain position in the input string. The reference to the GSS node not only tells the recognizer which grammar slot to continue with, but also indicates which derivations are interested in the result of this work. The code after replacing the stack with the GSS data structure is listed in Listing 3.7.

```java
Scanner scanner = new Scanner(I); GSSNode currentNode;
Set<(GrammarSlot, Stack, Int)> R;
void init() {
    R.add(S' ::= S, ⟨⊥,0 ⟩,0);
}
boolean S'() {
    init();
    while (¬R.isEmpty()){
        (slot, node, i) = R.pop();
        currentNode = node;
        scanner.setPosition(i);
        Goto(slot);
    }
    return S, ⟨[],#⟩ has been processed;
}
void S() {
    if (scanner.hasNext("b")) {
        add(S ::= b S, Cu, scanner.getPosition());
    }
    if (scanner.hasNext(test(A))) {
        add(S ::= A S d, Cu, scanner.getPosition());
    }
    if (scanner.hasNext("")) {
        add(S ::= ε, Cu, scanner.getPosition());
    }
}
void A() {
    if (scanner.hasNext("a")) {
        add(A ::= a, Cu, scanner.getPosition());
    }
}
void create(GrammarSlot slot){
    if there is no GSS node (slot, scanner.getPosition()) then create one {
        if there is no GSS edge from (slot, scanner.getPosition()) to Cu then create one
        return (slot, scanner.getPosition());
    }
}
void pop() {
    if (Cu != ⟨⊥,0 ⟩) {
        for each edge from Cu to v{
        }
    }
}
```

...
Note that the stack data structure stack has now been replaced by a collection of stacks as defined by GSS node $C_u$. Since we are now maintaining a graph of stacks rather than just one, operations create() and pop() have to be adjusted. Each time a grammar slot $g$ is created, a check is required to see whether there already exists a GSS node $(g, \text{scanner.getPosition()})$ with an edge to $C_u$. Note that after the GSS node has been created, the create() statement updates the current stacks of the algorithm by assigning the new GSS node to $C_u$. As for the pop() statement, recall from Section 3.6 that a descriptor is dropped once it has been processed. If the recognition of a nonterminal turned out to be successful, the pop() statement creates a new descriptor for the derivation to ensure that the recognizer knows where to continue next. Now that a descriptor considers multiple stacks at the same time, the grammar slot $g$ in the GSS node can now be of interest for more than one derivation. In order to continue the work of all these derivations, a descriptor with the popped grammar slot must be created for each of the remaining substacks. Since a substack in the GSS can be found by looking at the outgoing edges of a GSS node, the pop() statement now performs an add() operation for every child node of $C_u$. This is also illustrated in Figure 3.9.

To illustrate the control flow of the algorithm, let us consider the first steps for the recognition of the input string $aadS$ using Grammar $\Gamma_5$. Initially, the recognizer starts with the descriptor $(S' ::= \cdot S, (\cdot, aadS), aadS)$. Informally, this descriptor states that the recognizer must recognize nonterminal $S$ starting from input string $aadS$. The $(\cdot, aadS)$ represents the empty stack, implying that if the descriptor is able to recognize $S$ entirely, the recognizer can terminate. Since alternates $aS$, $ASd$, and $\varepsilon$ are all able to recognize either $a$ or $\varepsilon$ in the input string, the invocation of $S()$ results in the creation of three new descriptors with grammar slots $S ::= \cdot aS$, $S ::= \cdot A S d$, and...
S ::= · ε respectively. Note that the GSS node and input pointer for these descriptors remain the same, since S() is an alternate function.

Let us consider the descriptor for (S ::= · A S d, ⟨⊥, aad$⟩, aad$). Since A is a nonterminal, before exploring the alternates of A, the GSS node (S ::= A · S d, aad$) is created. This node is used to remember that after the recognition of A, a descriptor has to be created to continue the recognition starting from S ::= A · S d. After the GSS node has been created, the function A() invokes which results into the creation of the descriptor (A ::= · a, (S ::= A · S d, aad$), aad$). After the A has been recognized using this descriptor, a pop() statement is executed which creates a descriptor with grammar slot S ::= A · S d for every remaining stack. Remember that aad$ of the GSS node is only used for the detection of repetitive patterns and is therefore not relevant when considering the pop() statement. In this particular case, there is only one stack which is empty.

3.6.3 Descriptors and GSS sharing

The introduction of the GSS enforces descriptors to parse literals for multiple stacks at the same time. In order to capture any sharing between multiple derivations, the GSS stores all GSS nodes and edges that have been created throughout the parsing process. As the recognizer is considering derivations that have not been explored yet, the GSS can detect any form of sharing by checking at the moment of creation whether the GSS node (or edge) already exists. Hence, it is important to remember that sharing between derivations can only be detected once the recognizer has considered them. But how is it possible for two derivations to obtain this sharing? Figure 3.10 shows an example of such situation for the input string aad$ and Grammar Γ_5. For the sake of readability, the stacks for every derivation are shown separately.

![Figure 3.10: Illustration where 2 separate derivations share the GSS node (S ::= A · S d, ad$).](image-url)
Note that both derivations have the GSS node \( S ::= A \cdot S \cdot d, ad \$ \) in common. The main intuition behind this is that both derivations are recognizing the same nonterminal \( A \) of the production rule \( S ::= A \cdot S \cdot d \), and both derivations are recognizing \( A \) starting from \( ad \$ \) in the input string.

In general, we can conclude any form of sharing between derivations if it is possible for these derivations to end up in parsing the same literals in a production rule, starting from the same input position in the input string. In order to illustrate how the recognizer deals with this sharing, we will first have to show how the GSS is constructed in our recognizer. If we would run the recognizer from Listing 3.7 for the same input string as in our previous example, the following descriptors would be created:

1. As stated in line 6, initially we have:

<table>
<thead>
<tr>
<th>( R )</th>
<th>Informal description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( S' ::= \cdot S, (\cdot, aad$), aad$ )</td>
<td>Recognize nonterminal ( S ) starting from ( aad$ ). Upon success, we are done with ( S ).</td>
</tr>
</tbody>
</table>

2. When processing this descriptor, the function Goto(\( S' ::= \cdot S \)) will lead the recognizer to \( S() \). Since alternates \( aS \), \( ASd \) and \( \epsilon \) all satisfy test(\( S \)) for the input string \( aad\$ \), for every alternate a descriptors is made. Set \( R \) now looks as follows.

<table>
<thead>
<tr>
<th>( R )</th>
<th>Informal description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( S ::= aS, (\cdot, aad$), aad$ )</td>
<td>Recognize ( S ) as ( aS ) starting from ( aad$ ). Upon success, done</td>
</tr>
<tr>
<td>( S ::= A \cdot S \cdot d, (\cdot, aad$), aad$ )</td>
<td>Recognize ( S ) as ( ASd ) starting from ( aad$ ). Upon success, done</td>
</tr>
<tr>
<td>( S ::= a\epsilon, (\cdot, aad$), aad$ )</td>
<td>Recognize ( S ) as ( \epsilon ) starting from ( aad$ ). Upon success, done</td>
</tr>
</tbody>
</table>

3. Consider the descriptor for \( S ::= \cdot aS \). The Goto(\( S ::= \cdot aS \)) will lead the recognizer to \( S0() \). After recognizing “a” and upon encountering nonterminal \( S \) of \( aS \), the create statement updates the GSS with \( (S ::= aS, ad\$) \) after which \( S() \) is invoked. Since alternates \( aS \), \( ASd \) and \( \epsilon \) again satisfy test(\( S \)) for the input string \( ad\$ \), for every alternate a a descriptor is made:

<table>
<thead>
<tr>
<th>( R )</th>
<th>Informal description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( S ::= A \cdot S \cdot d, (\cdot, aad$), aad$ )</td>
<td>Recognize ( S ) as ( ASd ) starting from ( aad$ ). Upon success, done</td>
</tr>
<tr>
<td>( S ::= \epsilon, (\cdot, aad$), aad$ )</td>
<td>Recognize ( S ) as ( \epsilon ) starting from ( aad$ ). Upon success, done</td>
</tr>
<tr>
<td>( S ::= aS \cdot d, (S ::= aS, ad$), ad$ )</td>
<td>Recognize ( S ) as ( ASd ) starting from ( ad$ ). Upon success, create a descriptor that continues with ( S ::= aS \cdot d ).</td>
</tr>
<tr>
<td>( S ::= a\epsilon, (S ::= a\epsilon, ad$), ad$ )</td>
<td>Recognize ( S ) as ( \epsilon ) starting from ( ad$ ). Upon success, create a descriptor that continues with ( S ::= \cdot a ).</td>
</tr>
</tbody>
</table>

4. Consider descriptor \( (S ::= \cdot A \cdot S \cdot d, (S ::= aS, ad\$), ad\$) \). Upon encountering nonterminal \( A \) of \( S ::= A \cdot S \cdot d \), the create statement updates the GSS with \( (A \cdot S \cdot d, ad\$) \) after which \( A() \) is invoked. Since alternate \( a \) satisfies test(\( A \)) for the input string \( ad\$ \), a descriptors for \( A ::= a \) is made:

<table>
<thead>
<tr>
<th>( R )</th>
<th>Informal description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( S ::= A \cdot S \cdot d, (\cdot, aad$), aad$ )</td>
<td>Recognize ( S ) as ( ASd ) starting from ( aad$ ). Upon success, done</td>
</tr>
<tr>
<td>( S ::= \epsilon, (\cdot, aad$), aad$ )</td>
<td>Recognize ( S ) as ( \epsilon ) starting from ( aad$ ). Upon success, done</td>
</tr>
<tr>
<td>( S ::= A \cdot S \cdot d, (S ::= aS, ad$), ad$ )</td>
<td>Recognize ( S ) as ( ASd ) starting from ( ad$ ). Upon success, create a descriptor that continues with ( S ::= aS \cdot d ).</td>
</tr>
<tr>
<td>( S ::= a\epsilon, (S ::= a\epsilon, ad$), ad$ )</td>
<td>Recognize ( S ) as ( \epsilon ) starting from ( ad$ ). Upon success, create a descriptor that continues with ( S ::= \cdot a ).</td>
</tr>
<tr>
<td>( A ::= a, (S ::= A \cdot S \cdot d, ad$), ad$ )</td>
<td>Recognize ( A ) as ( a ) starting from ( ad$ ). Upon success, create a descriptor that continues with ( S ::= A \cdot S \cdot d ).</td>
</tr>
</tbody>
</table>
Figure 3.11 a) shows the resulting GSS according to this execution trace. Figure 3.11 b) shows the GSS after the recognizer has considered both derivations in Figure 3.10. Convince yourself, using a similar approach as above, that this GSS trace is obtained after processing descriptors:

5. \( (S ::= \cdot A S d, \langle \bot, aad\$ \rangle, aad\$) \)
6. \( (A ::= \cdot a, \langle S ::= A \cdot S d, aad\$, aad\$) \)
7. \( (S ::= A \cdot S d, \langle \bot, aad\$ \rangle, aad\$) \)
8. \( (S ::= \cdot A S d, \langle S ::= A S \cdot d, aad\$, ad\$) \)

After executing descriptors 1 up to and including 4, the recognizer (at this point) does not know that the second derivation in Figure 3.10 will also lead to the creation of the GSS node \( (S ::= A \cdot S d, aad\$) \). Since both derivations create the same GSS node when considering grammar slot \( S ::= \cdot A S d \), we can see that the descriptor \( (A ::= \cdot a, \langle S ::= A \cdot S d, ad\$ \rangle, ad\$) \) is relevant for both derivations. But what happens if the recognizer processes this descriptor before this sharing is detected? Apparently, depending on the order in which descriptors \( (S ::= \cdot A S d, \langle \bot, aad\$ \rangle, aad\$) \) and \( (A ::= \cdot a, \langle S ::= A S \cdot d, ad\$, ad\$) \) are processed, the recognizer has to react differently. We will consider the meaning of every ordering separately:

1. \( (S ::= A S d, \langle \bot, aad\$ \rangle, aad\$) \) processed before \( (A ::= a, \langle S ::= A S d, ad\$ \rangle, ad\$) \)
   In this situation, the recognizer will notice that the descriptor for \( S \) will eventually lead to the creation of the GSS edge \( (S ::= A S d, aad\$) \to (S ::= A S \cdot d, ad\$) \). As a consequence, the GSS now contains all derivations that are interested in the descriptor for \( A \). The result of processing this descriptor will now lead to the creation of two descriptors, one for each derivation.

2. \( (A ::= a, \langle S ::= A S d, ad\$ \rangle, ad\$) \) processed before \( (S ::= A S d, \langle \bot, aad\$ \rangle, aad\$) \)
   Since the descriptor for \( S \) has not yet been processed, the GSS does not contain the edge \( (S ::= A S d, ad\$) \to (S ::= A S \cdot d, ad\$) \). This implies that the recognizer processes the descriptor for \( A \) while it is unaware of the sharing with the second derivation. Hence, the processing of this descriptor will now lead to the creation of only one descriptor. When considering the descriptor for \( S \) afterwards, the algorithm however will notice that this descriptor was processed too soon, since the GSS node \( (S ::= A S d, ad\$) \) has received a new edge that was not present at the moment the descriptor for \( A \) was processed.

Case 1 is the “ideal” situation where all relevant derivations are known before a descriptor is processed. Unfortunately, it is hard if not undecidable for the recognizer to know when all sharing of a GSS node has been detected. Case 2 is an example of a situation where both derivations have some work in common, but this sharing is detected after the descriptor has already been
processed. This is mainly due to the fact that it still possible for other derivations in the future to have some overlap with the ones that have been considered before.

The recognizer in Listing 3.7 does not distinguish between these situations 1 and 2 and solves the problem in Case 2 by naively processing the descriptor for $A$ a second time. Although at first descriptors multiple times seems harmless, in the next section we will see however why this is not such a great idea.

### 3.6.4 Fine-tuning: Duplicate descriptors

If we naively run the recognizer as presented in Listing 3.7, then you will notice that the processing of descriptors $(S ::= \cdot a S, \langle \bot, a a d$, $a d \rangle)$ and $(S ::= \cdot A S d, \langle \bot, a a d$, $a d \rangle)$ both will eventually lead to the creation of $(A ::= \cdot a, (S ::= A \cdot S d, a d, a d))$. Although it is possible to naively process the same descriptor twice, be aware that the GSS does not make a distinction between old and new edges. As a consequence, at the occurrence of a pop() statement, a duplicate descriptor will not only create descriptors for the new GSS edges, but also for the old ones that have been considered earlier.

Although this seems nothing more than an efficiency issue, be aware that in case of left-recursion, the recognizer will no longer terminate. Figure 3.12 illustrates how descriptors are created when a pop() statement is executed. Recall that the GSS uses loops to model left-recursive patterns. Since left-recursion does not make any progress in the input string, applying the pop() statement for the descriptor $(b, b, c)$ results in the creation of two descriptors, namely $(b, d, c)$ and descriptor $(b, b, c)$ again. Hence in order to avoid the infinite processing of $(b, b, c)$, the GLL recognizer must not be allowed to process a descriptor twice. Besides, when looking at duplicate descriptors from the viewpoint of a job description, recognizing the same literals twice seems rather redundant.

![Figure 3.12: In case of left-recursion, the pop() of a descriptor leads to a duplicate descriptor.](image)

If we want to disallow the creation of duplicate descriptors, we first have to find a solution for sharing that is detected afterwards. Recall from Case 2 that if descriptor for $A ::= \cdot a$ is processed before the descriptor for $S ::= \cdot A S d$, the recognizer only creates one descriptor as the second derivation of Figure 3.10 has not been considered yet. To ensure that the second derivation can continue once it has been detected, we have to make sure that the second descriptor is still created afterwards. So how does this look like in the recognizer?

The difference between Case 1 and Case 2 is that in Case 2 the GSS node $(S ::= A \cdot S d, a d)$ has already been involved in a pop() statement. So in order to detect any late sharing, we first
need to know whether a GSS node has already been popped before. For this we introduce a set \( P \). Whenever the recognizer wants to create a GSS node, we use this set to verify whether or not we still have to create descriptors for the derivations that have not been considered before. In order to know where these descriptor must start in the input string, \( P \) not only has to store the GSS nodes that have been popped, but also the input position at the moment the pop() has occurred. So, in essence, we can see that \( P \) is responsible for storing the recognizer results after a GLL block has been considered. The result of applying this last transformation results into the code in Listing 3.8. The code in red illustrates the additions that are made in order to avoid the creation of duplicate descriptors.

```
Scanner scanner = new Scanner(I);
GSSNode currentNode;
Set<(GrammarSlot, Stack, Int)> R;
Set<(GrammarSlot, Stack, Int)> U;
Set<(GSSNode, Int)> P;

void init() {
    R.add(S' ::= S, ⟨⊥, 0⟩, 0);
}

boolean S'() {
    init();
    while (∼R.isEmpty()){
        ⟨slot, node, i⟩ = R.pop();
        currentNode = node;
        scanner.setPosition(i);
        Goto(slot);
    }
    return S, ⟨[, #]⟩ has been processed;
}

void create(GrammarSlot slot){
    if there is no GSS node (slot, scanner.getPosition()) then create one
        if there is no GSS edge from (slot, scanner.getPosition()) to Cu then create one
            for all (s, z) ∈ P with s = (slot, scanner.getPosition()){
                add(slot, Cu, z);
            }
    return ⟨slot, scanner.getPosition()⟩;
}

void pop() {
    if (Cu != ⟨⊥, 0⟩) {
        P.add(Cu, scanner.getPosition());
        for each edge from Cu to v{
            add(slot, v, scanner.getPosition());
        }
    }
}

void add(GrammarSlot g, GSSNode s, Int pos) {
    Descriptor d = new Descriptor(g, s, pos);
    if (~U.contains(d)){
        R.add(d);
    }
}
```

Listing 3.8: Pseudocode GLL recognizer for grammar \( Γ_3 \)

In order to avoid descriptors for being created twice, the recognizer has to maintain a set \( U \) storing all descriptors that have been created throughout the recognition process. We can now easily solve this problem by checking in the add() statement whether a descriptor has already been processed in the algorithm.
If we now return to Case 2 of our example in Figure 3.10, the resulting control flow is now as follows. After the descriptor \((A ::= \cdot a, (S ::= A \cdot S \ d, ad$), ad$)\) has been processed, a `pop()` creates a descriptor which continues with \(S ::= A \cdot S \ d\). The fact that the recognizer managed to recognize \(S ::= A \cdot S \ d\) until \(ad$\) is stored in \(P\). As soon as the recognizer reaches the descriptor \((S ::= A \cdot S d, (S ::= A S \cdot d, ad$), ad$)\), the `create()` statement detects sharing at grammar slot \(S ::= A \cdot S d\). At this moment the recognizer is aware that at least one alternate of \(A\) that has been considered before by a different derivation. Hence instead of recognizing \(A\) again, the recognizer can use the result stored in \(P\) to skip the recognition of \(A\) by immediately creating a descriptor for the grammar slot \(S ::= A \cdot S d\). Using the set \(P\), recognizer knows that this descriptor must continue with \(ad$\). We have now obtained a fully functional GLL recognizer for Grammar \(\Gamma_5\).

### 3.7 Towards a GLL parser

To reduce the gap between the transformation from an GLL recognizer to a parser, let us first investigate how an LLRD recognizer with an explicit stack must be augmented to produce a parse tree. The LLRD parser in Section 3.3 uses the return keyword to pass the parse tree of a nonterminal \(A\) to a higher level of recursion (i.e. the production rule that initiated the parsing of \(A\)). Now that we have an LLRD recognizer with an explicit stack, however, this statement can no longer be used. Recall from Section 3.5 that the return keyword corresponds to a `pop()` statement that in fact initiates the transition to jump from one grammar slot to the other. So, if we would consider a stack as a graph, then we can model the data passing of a return statement on an edge between two stack entries. This is also illustrated in Figure 3.13.

```plaintext
Scanner scanner = new Scanner(I);
SNode S() {
    SNode root = new SNode();
    root.addChild(A());
    return root;
}
ANode A() {
    return new ANode("a");
}

(a) Return statement from A() to S().
(b) Data passing by means of an edge.
```

Figure 3.13: Data transfer by means of a `return` statement can be modeled as an edge between the two corresponding stack entries.

For the realization of the parser, similarly as in Section 3.3, we create a parse node whenever the parser has encountered a terminal. The function `getNodeT()` is responsible for this action. Once all the parse nodes of all literals in a production rule \(A ::= l_1 \cdots l_n\) have been created, a function `getNodeP()` is used to create a parse node for \(A\) after which all parse nodes of \(l_1 \cdots l_n\) are attached to this node. In order store the parse nodes of \(l_1 \cdots l_n\) throughout the parsing process, a variable \(C_N\) is introduced. Listing 3.9 shows the resulting code for the parser.

```plaintext
Scanner scanner = new Scanner(I);
Stack s = new Stack();
GrammarSlot g;
List<ParseNode> C_N;

boolean S'() {
    create(\(\cdot\));
    S();
}
```

Exploring and visualizing GLL parsing
while (!stack.isEmpty()) {
    Goto(g);
}
return scanner.hasNext("$");

void S() {
    if (scanner.hasNext("b")) {
        g = S ::= b S;
    } else if (scanner.hasNext(test(A))) {
        g = S ::= · A S d;
    } else if (scanner.hasNext("") ) {
        g = S ::= · ε;
    } else {
        error();
    }
}

void S0() {
    if (scanner.hasNext("b")) {
        CN.add(getNodeT("b"));
        scanner.next("b");
        create(S ::= b S ·);
        S();
    } else {
        error();
    }
}

void S0_0() { pop(); }

void S1() {
    if (scanner.hasNext(test(A))) {
        create(S ::= A · S d);
        A();
    } else {
        error();
    }
}

void A() {
    if (scanner.hasNext("a")) {
        g = A ::= · a;
    } else {
        error();
    }
}

void A0() {
    CN.add(getNodeT("a"));
    scanner.next("a");
    pop();
}

void S1_0() {
    if (scanner.hasNext(test(S))) {
        create(S ::= A S · d);
        S();
    } else {
        error();
    }
}

void S1_1() {
    if (scanner.hasNext("d")) {
        CN.add(getNodeT("d"));
        scanner.next("d");
    }
}
Listing 3.9: Pseudocode LLRD parser for grammar $\Gamma_3$ with explicit stack.

Whenever the parser has managed to parse a literal $l_i$ in alternate $A ::= l_1 \cdots l_i \cdots l_n$, the root of the resulting parse tree is added to variable $C_N$. After literal $l_n$ has been parsed, the recognizer is done with nonterminal $A$ and we can use the function $\text{getNodeP}()$ to attach all the nodes in $C_N$ to the parse node for $A$. Figure 3.14 illustrates how the $\text{getNodeP}()$ function is used when recognizing the string $ad$ with Grammar $\Gamma_3$.

Variable $C_N$, in essence, maintains a list of parse nodes that have to be attached to a nonterminal parse node for $B$. Be aware however that as soon as the parser encounters a new nonterminal $B$, variable $C_N$ has to be used to store all parse nodes for that nonterminal. Hence, we must temporarily store the content of $C_N$ somewhere else before the variable can be used for $B$. To ensure that we can continue with the old content of $C_N$ after $B$ has been parsed, the $\text{create}()$ statement stores the value of $C_N$ on the edge of the newly created grammar slot. After the parsing of $B$ has been successful, a $\text{pop}()$ statement is executed which attaches the new content of $C_N$ to the parse node of $B$ using the $\text{getNodeP}()$ function. The result of parsing $A$ can now be
stored by taking the old content of $C_N$ from the stack edge and add the parse node of $A$ to $C_N$. Figure 3.15 shows an illustration how variable $C_N$ and stack $s$ are affected by functions create() and pop().

![Diagram](image1)

Figure 3.15: The create() function stores the discovered parse nodes on the stack so that the pop() can obtain this information at a later stage.

### 3.7.1 SPPF

Section 3.7 illustrated earlier that the root of the parse tree discovered so far can be stored on an edge between two stack entries. In order to obtain a GLL parser, the recognizer has to be augmented with an additional data structure maintaining all valid parse trees that have been discovered. Since a GLL parser is capable in parsing ambiguous grammars, it is possible that there exist multiple derivations for the same input string. One way to realize a GLL parser is to create a separate parse tree for every derivation. This can be realized by augmenting the edges of the GSS data structure using the same strategy as shown in Section 3.7. Unfortunately, for highly ambiguous grammars (such as Grammar $\Gamma_6$), the number of parse trees can become infinite. This is also illustrated in Figure 3.16.

![Diagram](image2)

Figure 3.16: There are examples that can lead to infinite amount of parse trees.

Note that in this particular case it is possible for nonterminal $S$ to be rewritten to an infinitely large string of $S$ nonterminals without making any progress in the input string. Grammars such
as $\Gamma_6$ will be referred to as a SPPF cyclic, since they may require cycles to represent all parse trees for an input string finitely. In general, a grammar $G = (N, T, R, S)$ is SPPF cyclic if and only if:

- A nonterminal $A \in N$ can be rewritten in one or more steps to $AA\beta$, where $\beta$ can represent any string of literals, and
- $A$ can be rewritten in one or more steps to $\varepsilon$.

Be aware that grammars can only be SPPF cyclic due to the presence of nullable nonterminals, since they can be recognized without making any progress in the input string. In order to take these issues into account, again some form of sharing has to be introduced to compactly capture all possible parse trees into one data structure.

Intuitively, two parse trees share a parse node $n$ if and only if $n$ parses the same part of the input string in both parse trees. So we can combine multiple parse trees together by extending every parse node with two numbers $a$ and $b$ denoting the range of the input string that is parsed by that node. Variables $a$ and $b$ are also referred to as the extents of a parse node. Since these parse nodes either represent a terminal or a nonterminal, by the terminology of Scott and Johnstone, these nodes are also referred to as symbol nodes. Figure 3.17 shows how this information can be used to show any overlap between two or more parse trees for the input string $1+1+1$ and Grammar $\Gamma_4$.

Figure 3.17: Large amount of sharing can be detected by extending every parse node with left and right extents.

Figure 3.16 illustrates that every occurrence of $S$ can be used to generate an infinite amount of parse trees. So the new data structure has to be aware that the sharing between two parse trees can start at any position in a production rule. The colors in Figure 3.17 shows how much of this sharing can be present between two parse trees. In order to capture this overlap to its full extent, the new data structure must be able to express the parse trees in terms of these common subbranches. Note that in the original parse tree this is hard to realize, since a nonterminal node
may have references to multiple subbranches. Hence, we would like to have an intermediate node in the parse tree to represent each subbranch separately. An intermediate node \((A ::= \alpha \cdot \beta, a, b)\) represents the parsing of \(\alpha\) where \(a\) and \(b\) represent the part of the input string that was parsed by \(\alpha\). Figure 3.18 shows the result of introducing intermediate nodes to the original parse tree.

![Figure 3.18: Maximum sharing can be obtained by introducing intermediate nodes for subbranches.](image)

We can now for instance refer to the purple subbranch by means of the intermediate node \((E ::= E \cdot E, 0, 2)\). Note that intermediate nodes are introduced whenever the parser manages to parse a new literal without completing an alternate. After all literals of an alternate have been parsed, a nonterminal symbol node is used to represent the parsing of that alternate as a whole.

When looking closely to the resulting tree structures, you will notice that every node can now have at most two children. Do note however that this is a direct consequence of trying to obtain maximum sharing. Since sharing can start from any position in a production rule and the parser does not know in advance where sharing will take place, maximum sharing can only be obtained if every possible position in a production rule is considered separately in the parse tree. Hence, in worst case situation, the introduction of intermediate nodes becomes redundant if there is no sharing at all between two derivations. Since every parse node has of at most 2 children, the resulting parse trees are also referred to as binarized parse trees.

Now that we have a way to capture sharing to it finest granularity, it is time to look at combining multiple parse trees into one graph structure. Note that when combining multiple trees into one structure, every nonterminal or intermediate node can now correspond to more than one derivation. We can model this non-determinism in our graph structure by creating an additional
node for every non-deterministic choice that a parse node can provide. The additional nodes are also referred to as packed nodes, since they have the task to pack two branches together wherever necessary. Figure 3.19 shows how the introduction of packed nodes can be used to merge multiple parse trees together.

Note that for unambiguous grammars, the packed nodes in the SPPF are redundant, since by definition of an unambiguous grammar there is only one valid alternate for every nonterminal at
the time. Further note that many packed nodes in the resulting picture are merged together. From an algorithmic point of view, every packed node is a 2-tuple \((g = A ::= \alpha \cdot \beta, i)\), where \(g\) indicates which grammar slot is involved with the non-deterministic choice and \(i\) represents the position where the input string must be split to parse \(\alpha\). A more intuitive way of presenting this information is described in Section 4.4.6. Figure 3.20 shows the full data structure for the input string \(aad\$\) and Grammar \(\Gamma_5\).

The result is a graph data structure consisting of three types of nodes, namely symbol nodes, packed nodes, and intermediate nodes. Note that, apart from their representation, there is no semantical difference between a nonterminal node and an intermediate node. In fact, we can represent a nonterminal node as an intermediate node by placing the \(\cdot\) at the end of the production rule. This is also illustrated in Figure 3.21.

Now we can clearly see that a packed node represents a choice showing how the input string must be divided with respect to the part of the alternate that has to be parsed. In contrast to symbol and intermediate nodes, it is possible for a packed node to occur multiple times within an SPPF. Keep in mind however that a packed node is only used to locally distinguish one derivation from
CHAPTER 3. GENERALIZED LL PARSING

the other. The data structure as illustrated in Figure 3.20 is also known as a Shared Packed Parse Forest (in short SPPF).

3.8 GLL parser

Section 3.7 illustrated how functions getNodeT() and getNodeP() can be used to transform an LLRD recognizer (with explicit stack) into an LLRD parser. Even though we are now dealing with an SPPF data structure, the transformation from an GLL recognizer to a parser is very similar to the one as described in Section 3.7. Similarly as in the LLRD parser, a GLL parser maintains a variable $C_N$ to store SPPF nodes that have been discovered so far. Since the SPPF data structure is binarized, be aware that $C_N$ can now represent these nodes by means of a single SPPF node. This node represents the entire SPPF data structure that has been discovered so far. Figure 3.22 illustrates how the usage of $C_N$ looks like in a GLL parser. Note that with an SPPF, $C_N$ can use the intermediate node $(S := a b \cdot c,0,2)$ to store the parsing results for both $a$ and $b$. Note that $C_N$ can either be a symbol node or an intermediate node as packed nodes are merely used to store any non-determinism.

For the construction of the SPPF data structure, we can reuse the same concept as in a LLRD parser, by introducing functions getNodeT() and getNodeP() for the creation of leaf nodes and internal SPPF nodes respectively. Hence, we will use getNodeT($a,i,j$) to create an SPPF symbol node for terminal $a$ with extents $i$ and $j$ if it not already exists. The getNodeP(slot,$C_N,C_R$) will be used to attach an SPPF branch $C_R$ to $C_N$ either by means of an intermediate or a nonterminal symbol node. In case variable $C_N$ is empty (denoted with special symbol $\$), this function will only create a new SPPF node for the branch $C_R$. The actual implementation of this function will be discussed later on.

Recall that in order to obtain a binarized SPPF, intermediate nodes have to be created by the parser whenever a new literal of an alternate has been parsed. Since intermediate nodes and nonterminal symbol nodes are in essence the same, the getNodeP() function can be used for the creation of both of these nodes. The result is that getNodeP() is not only invoked inside the pop() statement, but also whenever a new literal has been parsed. The SPPF in $C_N$ can now be updated with new parse nodes by iteratively combining $C_N$ with the new parse nodes using getNodeP(). Figure 3.23 shows the difference between the construction of $C_N$ in an LLRD parser versus a GLL parser.

Figure 3.22: In a GLL parser, there is no need for $C_N$ to be an array anymore.
Important to note is that the GLL parser immediately assigns the value of getNodeT("a"). Similarly as in LLRD parser, this is because $C_N$ is always set to empty at the beginning of an alternate. If we would consider the function for grammar slot $S ::= a b c D \cdot e$ on the other hand, then we know that $C_N$ contains the SPPF tree for literals $abcD$ and the parsing of $e$ has to be added to $C_N$. Listing 3.10 shows how a GLL parser realizes this situation. Now note that getNodeT() is first assigned to variable $C_R$ after which it is combined with $C_N$.

Listing 3.10: Pseudocode how GLL parser constructs variable $C_N$

To ensure that the creation of intermediate nodes works well, there is one situation that the GLL parser has to consider separately. In the situation where a grammar slot only consists of one terminal (e.g. $S ::= \epsilon$), the GLL parser has to make sure that SPPF nodes for both $\epsilon$ as well as nonterminal $S$ are created before a pop() statement is executed. To illustrate why this is important, suppose that we are parsing $S ::= ASd$ and we have already managed to parse $A$. Recall from Section 3.7 that the parse tree for $A$ is put on an edge in the stack and we need the pop() statement to obtain these results again. Now suppose that we have managed to parse $S$ as $S ::= \epsilon$. In other words, using the getNodeT() function we have successfully created an SPPF node for $\epsilon$. Since this is the only literal in the alternate, a pop() statement is invoked which has to combine both the results of parsing $A$ and $S$ as $\epsilon$ together in one SPPF node. In order to know however that the intermediate node $S ::= AS \cdot d$ has to be made, the SPPF node of $\epsilon$ still needs to be attached to an SPPF node for $S$. Hence, in case of only one terminal, the GLL parser also has to invoke a getNodeP() function before the pop() statement is invoked. This is also illustrated in Listing 3.24a. Figure 3.24b graphically shows why $S$ must be created in order to obtain a valid SPPF.
(a) Pseudocode how GLL parser constructs variable \( C_N \) for an alternate consisting of one terminal.

(b) In order to create an intermediate SPPF node \( S ::= AS \cdot d \) SPPF nodes for both \( A \) and \( S \) must be known.

Figure 3.24: If an alternate consists of one terminal, an additional getNodeP() invocation is required to obtain a valid SPPF.

Since grammar slot \( \varepsilon \) represents the beginning of an alternate, note that \( C_N \) corresponds to \$. Hence the getNodeP() knows that it should only create a nonterminal SPPF node for \( C_R \).

### 3.8.1 Finalizing the parser: extending descriptors

Section 3.6.2 earlier showed that for the support of multiple derivations, the grammar slot and input position of a derivation are stored together in a descriptor. Similarly, since \( C_N \) is a global variable and the value of \( C_N \) depends on the derivations that the algorithm is considering, a descriptor is also extended with an SPPF node. This implies that the add() now also has to be extended with an additional parameter \( s \) representing the SPPF node to be attached to the descriptor.

The pop() statement of a GLL parser is very similar to that of an LLRD parser. Since a GSS node can represent multiple stacks, this time, the getNodeP() function is invoked for every child GSS node of \( C_u \). The result is stored in the descriptor so that the parser knows which part of the SPPF have already been created.

Recall from the GLL recognizer that, due to late sharing in the GSS, a set \( P \) had to be introduced storing the recognizer results whenever a GLL block was recognized. Since a GLL recognizer only has to report whether there exists a derivation, note that it was sufficient to store the input position after a GLL block has been considered. In case of a GLL parser however, we are no longer interested in only the existence of a derivation. So in order for a descriptor to know where it must continue in the SPPF in case late sharing is detected, \( P \) now has to store the SPPF graph that was obtained after parsing that GLL block. Since the SPPF is incrementally constructed in \( C_N \), luckily, we can solve this issue by storing \( C_N \) in \( P \) whenever a pop() statement is executed.

The create() statement can now reuse the SPPF fragments in \( P \) by combining these fragments to the final SPPF in \( C_N \) using the getNodeP() function. Once the parser has finished, it returns the root of the SPPF by returning the SPPF node for \( S \) whose left and right extents together cover the entire input string. If such node does not exist, then we know that the parser did not manage to parse the input string entirely. Listing 3.11 shows the resulting code after augmenting a GLL recognizer with an SPPF data structure.
void init()
{
  add(S' ::= · S, ⟨⊥, 0⟩, 0, $);
}

boolean S'()
{
  init();
  while(¬R.isEmpty()){
    (slot, node, i, s) = R.pop();
    C_N = s;
    currentNode = node;
    scanner.setPosition(i);
    Goto(slot);
  }
  return SPPFNode(S,0,#I) has been created;
}

void S()
{
  if (scanner.hasNext("b")){
    add(S ::= · b S, Cu, scanner.getPosition(), $);
  } else if (scanner.hasNext(test(A))){
    add(S ::= · A S d, Cu, scanner.getPosition(), $);
  } else if (scanner.hasNext("")){
    add(S ::= · ε, Cu, scanner.getPosition(), $);
  }
}

void S0()
{
  C_N = getNodeT("a");
  scanner.next("a");
  if (scanner.hasNext(test(S))){
    create(S ::= a S ·);
    S();
  }
}

void S0_0()
{
  pop();
}

void S1()
{
  create(S ::= A · S d);
  A();
}

void A()
{
  if (scanner.hasNext("a")){
    add(A ::= · a, Cu, scanner.getPosition(), $);
  }
}

void A0()
{
  SPPFNode C_R = getNodeT("a");
  scanner.next("a");
  C_N = getNodeP(A ::= a · C_N, C_R);
  pop();
}

void S2()
{
  SPPFNode C_R = getNodeT(""");
  scanner.next(""");
  C_N = getNodeP(S ::= ε · C_N, C_R);
  pop();
}

void S1_0()
{
  if (scanner.hasNext(test(S))){
    create(S ::= A S · d);
    S();
  }
void S1() {
  if (scanner.hasNext("d")) {
    scanner.next("d");
    pop();
  }
}

void create(GrammarSlot slot){
  if (there is no GSS node (slot, scanner.getPosition()) then create one
    if (there is no GSS edge from (slot, scanner.getPosition()) to C_u then create one
      for all (s, z) ∈ P with s = (slot, scanner.getPosition())
        SPPFNode y = getNodeP(slot,C_N,z);
        add (slot, C_u, x.getRightExtent(), y);
    return (slot, scanner.getPosition());
  }
}

void pop() {
  if (C_u != ⟨⊥, 0⟩)
    for each edge from C_u to v
      Let SPPFNode z be on the edge of (C_u,v)
      Let C_u be of the form ⟨slot,i⟩
      SPPFNode y = getNodeP(slot,C_N,z);
      add (slot, v, scanner.getPosition(), y);
}

SymbolSPPFNode getNodeT(Terminal t) {
  Int leftExtent = scanner.getPosition();
  Int rightExtent = scanner.peek(t).length;
  if(!S.contains(SymbolSPPFNode(t,leftExtent,rightExtent)))
    create one
  return SymbolSPPFNode(t, leftExtent, rightExtent);
}

SPPFNode getNodeP(Grammarslot slot, SPPFNode left, SPPFNode right) {
  Let slot be of the form A ::= α · β
  if(#α == 1 ∧ (head(α) is a terminal or a non-nullable nonterminal) ∧ β ≠ []){
    return right;
  } else{
    if(β ≠ []){ t = A; } else{ t = A ::= α · β }
    if(left == $){
      Let i and j be the left and right extents of SPPF node right
      if(~S.contains((t, i, j))) create one
        if((t, i, j)) does not have a packed node (A ::= α · β, i){
          create one with left child left and right child right
        }
    } else{
      Let j and the right extent of SPPF node right
      Let i and k be the left and right extents of SPPF node left
      if(~S.contains((t, i, j))) create one
        if((t, i, j)) does not have a packed node (A ::= α · β, k){
          create one with left child left and right child right
        }
    }
  }
}

return (t, i, j);
Now it only remains to show how the getNodeP() can be realized. The implementation of the function getNodeP() mainly consists of three parts. Be aware that this function has to create an intermediate node for the branches $C_N$ and $C_R$ if we have only managed to parse a part of an alternate. Note however that it only makes sense to create an intermediate node for a grammar slot $A::= l_1 l_2 \cdots$ if the SPPF branches for $l_1$ and $l_2$ are known beforehand. This is for instance not the case when parsing grammar slot $S::= ASd$. Note that after the $A$ has been parsed successfully, a pop() statement is executed after which grammar slot $S::= ASd$ is considered. Note however that the getNodeP() function in pop() now may not create an intermediate node, since the SPPF branch of $S$ is still unknown. The reason why this does not hold for a nullable nonterminals is because the substitution of that nonterminal by $\varepsilon$ may result in parsing the same problem instance. In Grammar $\Gamma_6$ one can easily obtain such cycle by substituting $S$ by $SS$ after which the first $S$ is again substituted by $\varepsilon$. The check whether an intermediate node must be created by the getNodeP() function can be easily performed by checking whether the dot appears at the end of an alternate (i.e. $\beta = \varepsilon$ in $A::= \alpha \cdot \beta$). This check corresponds to line 118 of Listing 3.11.

If the getNodeP() is required to create an SPPF node, the first check that must be performed is whether an intermediate node or a nonterminal symbol node must be created. We again can use the check $\beta = \varepsilon$ to test this. One can easily determine the left and right extent of the new SPPF node by taking the left extent of the left SPPF node and the right extent of the right SPPF node. In the situation where the left branch is empty (i.e. equal to $\$), then getNodeP() knows that grammar slot of the descriptor corresponds to the beginning of an alternate and an SPPF node must only be made for the $C_R$ branch. This is covered by the check in line 122. We have now obtained a fully functional GLL parser.
3.9 OOGLL

Now that we have seen how an LLRD recognizer can be transformed into a GLL parser, it is time to combine the results together in an algorithm also referred to as Object-Oriented GLL (in short OOGLL). Especially when designing parts of the visualization as described in Section 4.4.4, this algorithm will become important. In contrast to the traditional GLL algorithm, the algorithm that is presented in this section uses a scanner to read the input string. One of the main differences with respect to the approach by Scott and Johnstone is that \texttt{goto} labels are replaced by function calls. To illustrate these differences in more detail, we revisit the pseudocode for generating a GLL parser.

In general, the object oriented GLL algorithm (in short OOGLL) consists of the following four tasks for generating a parser for grammar $G$:

1. Generate functions for every production rule in $G$
2. Generate functions for every GLL block in $G$
3. Generate a function to simulate \texttt{goto}()
4. Create functions \texttt{init()}, \texttt{pop()}, \texttt{create()}, \texttt{add()}, \texttt{getNodeT()}, \texttt{getNodeP()}, and \texttt{test}()

Note that the functions mentioned in the last step are parser independent. Therefore we have decided to create an abstract parser which contains these functions. This abstract parser also contains functions to interact with the data structures used by GLL. Each parser that is generated extends from this abstract parser. By using this approach, the code of the generated parser is very compact, and specific to the grammar at hand. Since the GLL algorithm \textit{generates} a parser, there are two types of code present in the following fragments of pseudocode, namely code that needs to evaluated by the GLL parser generator (highlighted in gray), and code that is actually present in the generated parser. For instance, depending on the rule $R$ for which a GLL function must be generated, the outcome of $A$ will be different. Hence $A$ is evaluated during the generation of the parser. As a starter, the pseudocode for generating an alternate parse function can be found in Listing 3.12.

```
For every rule $A ::= \alpha_1|\cdots|\alpha_m \in G$:

\begin{verbatim}
void A() {
  \textbf{if} (scanner.hasNext(test(A, A_1)) \{ add(A_1, e_u, j) \});
  \ldots
  \textbf{if} (scanner.hasNext(test(A, A_m)) \{ add(A_m, e_u, j) \});
}
\end{verbatim}
```

Listing 3.12: Code template for the generation of alternate rule functions

Here, $A_1, \ldots, A_m$ represent the function labels of alternates $\alpha_1, \ldots, \alpha_m$ respectively, variable $e_u$ represents the current GSS node and $j$ the current input pointer according to the original algorithm.

Now that we can generate alternate functions, let's look at match functions. The pseudocode for their generation can be found in Listing 3.13.

```
For every rule $A ::= a_1|\cdots|a_m \in G$, for every alternate $a_i$ in that rule
for every GLL block $g_k$ in $a_i$:

\begin{verbatim}
  if (k = 0){
  \textbf{void } A_i() {
    \textbf{void } A_{i,k-1}() {
    \}
    code(g_k A, k-1, k = 0, k = m-1, 1);
  \}
}\end{verbatim}
```

50 Exploring and visualizing GLL parsing
Listing 3.13: Pseudocode for the generation of GLL block functions

```plaintext
void code(g, func, id, firstSymbol, addPop, pos)
{
    if(#g != 0)
    {
        Let f := head(g);
        if(f ∈ T)
        {
            if(firstSymbol ∧ #g ≠ 1)
            {
                CN := getNodeT(f, scanner.getPosition());
                scanner.next(f);
            }
        }
        else
        {
            if(firstSymbol) if(scanner.hasNext(f))
            {
                CR := getNodeT(f, scanner.getPosition());
                scanner.next(f);
            }
            CN := getNodeP(L2G(func, id, pos), CN, CR);
        }
    }
    else
    {
        if(firstSymbol)
        {
            if(scanner.hasNext(test(func, f)))
            {
                Cu = create(func.id + 1, Cu, scanner.getPosition(), CN);
                goto(f);
            }
            code(tail(g), func, id, false, addPop, pos + 1);
        }
    }
    if(addPop){ pop(Cu, scanner.getPosition(), CN); }
}
```

Listing 3.14: Content of a GLL block function for match rules

The `firstSymbol` variable is used to denote whether the first literal of an alternate is being considered. In case the first literal is a terminal, the `scanner.hasNext()` statement is not required, since this check has already been performed by the alternate parse function who created the descriptor for the current function. Variables `func` and `id` together represent the name of the function that invoked `code`. Listing 3.14 line 13 also shows a function `L2G(A, k, pos)`, where `A` is the nonterminal of a grammar rule, `k` is the GLL block instance number (i.e. the `k`th GLL block in `A`), and `pos` is the literal position in the GLL block. The literal position is indexed relative to the start of the GLL block. Given the name of the function, it returns the position in the grammar to which this label corresponds.

Because of the fact that GLL block `g` is a string of literals, variable `f` can either be terminal or a nonterminal. In case `f` is a terminal, the scanner has to verify whether this terminal occurs in the input string. Since a GLL block may consist of one or more sequential terminals, it is important that these checks are nested. In case the `f` is a nonterminal, the scanner verifies whether an element in the test set of `f` occurs in the input string.

Since every function label corresponds to a function, we will use the `goto` function to invoke the function corresponding to label `f`. The variable `func_id + 1` in line 17 denotes the function where the algorithm should return to after function `f()` has been successfully processed. This return label is obtained from the active descriptor and exactly corresponds to the `R_{A_k}` label in the original algorithm.

The simulation of the `goto` in the original algorithm can be implemented by means of a switch case. Every function is identified by an integer, which can be passed to the `goto()` function. Let `i, ..., j` represent the identifiers for functions `f_i, ..., f_j` respectively. The code for the `goto` routine...
can be found in Listing 3.15. The pseudocode for the functions main, pop, create, add, getNodeP, and getNodeT is omitted here because their definitions stay the same.

```c
void Goto(f m){
    switch (m){
        case i: f_i(); break;
        ...
        case j: f_j(); break;
    }
}
```

Listing 3.15: Pseudocode for the generation of the goto functionality

To illustrate the differences between a parser generated by the original algorithm versus a parser generated by the OOGLL algorithm, consider Grammar Γ_7:

\[ S ::= aS | \varepsilon; \quad (\Gamma_7) \]

The pseudocode described in Listing 3.16 corresponds to the pseudocode according the original algorithm, whereas Listing 3.17 shows the code according to our algorithm. To ease the comparison, the order in which the labels occur in Listing 3.16 corresponds to the order in which the functions occur in Listing 3.17.
create GSS nodes $u_0 := (L_0,0)$
$\mathcal{R} := \{ (L_0,u_0,0); \mathcal{P} := \emptyset; C_u := u_0; C_N := \emptyset; C_I := 0$ for $0 \leq j \leq m$ \{ $U_j = \emptyset$ \}
$\text{goto } L_S$
\(L_0: \text{if}(\mathcal{R} \neq \emptyset) \{ \text{remove } (L,u,i,\omega) \text{ from } \mathcal{R} \}$
\(C_u := u; C_I := i; C_N := \omega; \text{goto } L \}$
\(\text{else if}(\text{there exists an SPPF node labelled } (S,0,m)) \text{ report success }$
\(\text{else report failure }$
$L_S: \text{if}(\text{test}(I[C],S,aS)) \text{ add }(L_{S_1},C_u,C_I,S)$
$\text{if}(\text{test}(I[C],S,\epsilon)) \text{ add }(L_{S_2},C_u,C_I,\emptyset)$
$L_{S_1}: C_N := \text{getNodeT}(a,C_I); C_I := C_I + 1;$
\(\text{if}(\text{test}(I[C],S,S)) \{ C_u := \text{create}(R_{S_1},C_u,C_I,C_N); \text{goto } L_S \}$
\(R_{S_1}: \text{pop}(C_u,C_I,C_N); \text{goto } L_S$
$L_{S_2}: C_R := \text{getNodeP}(\epsilon,C_I); C_N := \text{getNodeP}(\epsilon,C_N,C_R); \text{pop}(C_u,C_I,C_N);$
\(\text{goto } L_0$

Listing 3.16: "GLL parser using gotos for Grammar $\Gamma_5$"

\begin{verbatim}
S'() \{ 
create GSS nodes $u_0 := (\perp,0); \mathcal{R} := \{ (S,u_0,0); \mathcal{P} := \emptyset; C_u := u_0; C_N := \emptyset; C_I := 0$ for $0 \leq j \leq m$ \{ $U_j = \emptyset$ \}
while $\mathcal{R} \neq \emptyset$ \{ \text{remove } (id,u,i,\omega) \text{ from } \mathcal{R} \}$
\(C_u := u; \text{scanner.nextPosition}(C_I); C_N := \omega; \text{goto(id); }$
\} \text{ if } \text{there exists an SPPF node labelled } (S,0,m) \{ \text{report success }$
\} \text{ else } \{ \text{report failure }$
\} S() \{
\text{if}(\text{scanner.hasNext}(\text{test}(S0))) \text{ add }(S0,C_u,\text{scanner.getPosition}());$
\text{if}(\text{scanner.hasNext}(\text{test}(S1))) \text{ add }(S1,C_u,\text{scanner.getPosition}());$
\} S0() \{
\text{if}(\text{scanner.hasNext}(\text{test}(a))) \text{ add }(S0,C_u,\text{scanner.getPosition}());$
\text{if}(\text{scanner.hasNext}(\text{test}(S))) \{
\text{create}(S0,0,C_u,\text{scanner.getPosition}(),C_N);$\text{goto(S);}$
\} S0_0() \{
\text{pop}(C_u,\text{scanner.getPosition}(),C_N);$\}
S1() \{ 
\text{create}(S1,0,C_R,\text{scanner.getPosition}());$
\text{scanner.nextPosition}();$
\text{create}(S1,C_N,\text{getNodeT}(\epsilon,C_I));$
\text{scanner.nextPosition}();$
\text{create}(S1,C_N,\text{getNodeP}(\epsilon,C_N,C_R));$
\text{pop}(C_u,\text{scanner.getPosition}(),C_N);$\}
Goto(int 1) \{
\text{switch}(1) \{
\text{case } 0: S(); \text{break}; \text{case } 1: S1(); \text{break};$
\text{case } 2: S2(); \text{break}; \text{case } 3: S1_0(); \text{break};$
\} \}
\end{verbatim}

Listing 3.17: "OOGLL parser for Grammar $\Gamma_5$"
Chapter 4

Visualizing GLL

Chapter 3 illustrated how a GLL parser extends an LLRD parser by means of incremental program transformations. Although the transformations by themselves are relatively easy to understand once the difficulties are known, when trying to understand the algorithm from the viewpoint of a descriptor, you will notice that the control flow can become quite abstract. When developing new techniques for GLL parsers such as error handling or the support of EBNF format, however, the control flow of the algorithm is determinative for the design of these techniques. In the next sections, we will propose a technique to illustrate the progress of a GLL parser by visualizing the underlying control flow of the algorithm. Section 4.1 describes the set of requirements that must be taken into account when constructing the visualization. The goal of the system to be designed is to increase the understanding of the GLL algorithm by creating a multi-view system showing the relationships between descriptors, the GSS, and SPPF.

4.1 Requirements

This chapter contains the user requirements for the realization of the GLLExplorer system. The purpose of this chapter is to describe the functional requirements that must be taken into account when designing the visualization for the GLL algorithm. These requirements in turn are used as a guideline towards the realization of the system. The GLLExplorer software system creates an at runtime visualization of the GLL algorithm as defined by Scott and Johnstone [6]. From an educational point of view, the application should provide the user an overview of how the GLL algorithm performs any parsing by means of descriptors. This viewpoint focusses on a relatively small grammars and small input strings to gain understanding of the underlying data structures in the algorithm. By means of interaction, the user is able to decide which parts of the algorithm he is interested in. It is therefore desired to have a stateless visualization that can be constructed by looking at a snapshot of the data structures at a certain moment in time.

4.1.1 Data flow and views

This section describes the data flow and components that have been introduced in preparation of the implementation of the GLLExplorer system. The dataflow with respect to the GLLExplorer system is illustrated in Figure 4.1. Initially, the user specifies an input grammar for which a GLL parser must be created. After the system has generated the parser using the OOGGLL generation algorithm, the parser is augmented with code fragments that allows the visualization to extract data from the algorithm. The visualization in turn can use this data to create a step-by-step
visualization.

Figure 4.1: Data flow diagram of GLLExplorer system.

For visualization of the entire GLL algorithm, the GLLExplorer visualization is divided into several components. Depending on the data structures the user is interested in, he can decide which components to activate. For every component, the purpose and functionality is described in this chapter. Figure 4.2 shows a screenshot of the designed system.

Figure 4.2: Screenshot of GLLExplorer system.

### GLL Viewer 1

The GLL viewer is responsible for the visualization of the steps that the algorithm has to perform in order to create the right descriptors over time. The main goal of this component is to show how the algorithm uses different data structures to realize the final outcome of the parse. In that sense, this component is the most important part of the application.
CHAPTER 4. VISUALIZING GLL

GSS Viewer

The GSS viewer is responsible for the visualization of the GSS data structure. In particular for the understanding of the algorithm, it is important to see how the GSS evolves over time. By visualizing the current position in the GSS, the user knows which production rules have recursively been applied.

SPPF Viewer

This component is responsible for the understanding of the SPPF data structure. After parsing an input string \( I \), the GLL parser returns an SPPF containing all possible parse trees that are valid for \( I \). Since the SPPF is constructed in a bottom up fashion, it can be hard to understand how the data structure evolves over time. The SPPF viewer solves this issue by visualizing the parts of the SPPF that have been constructed so far.

Set, Grammar, and Descriptor Viewer

The set viewer is responsible for the visualization of the data structures \( R \), and \( U \) as defined in Section 3.6.3. Besides showing the actual content of these sets, the goal of this component is also to visualize how the algorithm uses these data structures during parsing. The descriptor view provides a short textual summary of the job that the current descriptor has to process. These descriptions are of the same format as defined in Section 3.6.3. The grammar view not only provides an overview of the grammar that is considered by the parser, but also shows the result of computing the test() of all nonterminals in that grammar.

Navigation Panel

The navigation panel allows the user to control the step-by-step visualization by means of “tape-recording” functionality such as back, stop, forward, and fast-forward. Depending on the level of detail the user is interested, the GLLExplorer system can visualize the execution of the algorithm after an arbitrary number of steps. Since descriptors are the main building blocks of a GLL parser, the application also allows the algorithm to be visualized at the level of a descriptor rather than the level of a statement.

4.1.2 General Requirements

This section contains a list of requirements that were taken into account when designing the visualization. For every requirement, an explanation is provided.

R.001: The GLLExplorer system provides mechanisms to suspend the execution of the GLL algorithm at any point in time.

Be aware that the execution of a GLL parser can involve a large number of statements. One can imagine however that the user is only interested in certain parts of the execution.

R.002: When processing a descriptor \( d \), the user is able to see the part of the input string that is parsed by \( d \).

Because of the fact that descriptors play a central role in the visualization, it is important to see the effect of executing a descriptor. Since obtaining a parse tree for the input string is the main goal of the algorithm, the GLLExplorer system clearly has to visualize the impact of processing such descriptor.
R_003: The user is able to skip the execution of descriptors where the user is not interested in. This requirement follows the line of Requirement R_001. If at a certain point the user becomes interested in the execution of the GLL parser, it should not be forced to view every descriptor that is created during this execution.

R_004: The GLLExplorer system provides a plug-and-play mechanism that allows GLL parsers to make easy use of the GLLExplorer system.

Note that a GLL parser is build on grammar-independent statements such as add(), create(), pop(), etc. Hence, if we can augment the visualization code within the scope of these statements, the GLLExplorer system can visualize the progress of a GLL parser without first having to transform them into a suitable format.

R_005: Whenever the user selects an descriptor d, the set viewer provides an informal description for d describing its purpose in the algorithm. Although the a descriptor by itself is rather abstract, the work that must be performed according to the descriptor is relatively easy to explain by means of an informal description. This allows the user to reason about descriptors in terms of these descriptions without looking at the content of it.

R_006: The user is be able to steer the GLL algorithm by changing the order in which descriptors are processed.

Since the order in which descriptors are processed by a GLL parser does not matter, the idea is to let the user choose its own ordering. This allows the user to only look at the parsing of specific derivations without first having to explore other derivations first.

R_007: The user is able to revert the algorithm to a state where it has been before (i.e. there is a notion of history).

A downside of the multi-view approach is that one step in the algorithm can result into changes in multiple views at the same time. Since the user at first may not know where these changes can occur, the GLLExplorer system must provide the user the ability to revert a step in the visualization.

R_008: After selecting a descriptor d, the user is able to skip the execution of the GLL algorithm until the next n steps have been performed. When no specific ordering has been defined by the user, descriptors are processed in the order the algorithm creates them.

This requirement states that it should not only be possible to hide the execution of an entire descriptor, but also hide the execution of specific steps within the scope of a descriptor.

R_010: The GLLExplorer system should provide the user a clear understanding how grammar slots and stacks are used to determine the control flow of a GLL parser.

Grammar slots and stacks play a fundamental role in the control flow of the algorithm. By visualizing how grammar slots are used to jump from one location to the other, the relationship between GLL parsing and multiple stacks can be explained.

4.2 Visualizing GLL: Walking grammars

The transformation of an LLRD parser to a GLL parser as described in Chapter 3 shows that we can obtain a GLL parser by maintaining a stack for all derivations that have to be discovered. Although other transformations such as the removal of the call-stack had to be performed as well, note that these transformations do not affect the idea behind LLRD parsing. Because of the close resemblance to LLRD parsing, for the visualization, the decision was made to view a GLL parser as a collection of LLRD parsers. To illustrate the process of a GLL parser, the concept of top-down parsing in general is used where production rules are “walked” from left to right and nonterminals
are parsed by trying alternates that are likely to result in a successful derivation. The main idea of
the visualization is to represent the instruction pointer of the algorithm as a walking figure, trying
to find derivations by exploring all possibilities for the nonterminals it encounters. The result of
exploring all alternates of a nonterminal results in a tree visualization as illustrated in Figure 4.3.

The visualization creates a node for every alternate that the parser encounters when exploring all
possible choices for a nonterminal. Since a nonterminal can have more than one valid alternate,
edges are used to denote this non-determinism. More specifically, the solid edges between a nonter-
minal and a node reflects a non-deterministic choice that the parser has made when considering
a certain derivation. Dashed edges represent alternates that the parser still has to explore when
considering other derivations.

We can visualize the progress of the parser with respect to a derivation by highlighting the literals
of every GLL node that have already been parsed. The difference between a successful and a
unsuccessful derivation is illustrated by colouring the progress of an alternate according to this
status. To illustrate the user that literals of alternates are considered from left to right, arrow
shaped blocks are used to denote this ordering. The steps that are performed in the GLL parser can
now directly be explained in terms of actions that the walking figure “intuitively” has to perform
in order to continue the parsing process. In Section 4.2.2 we will discuss how this mapping can be
realized.

Although the concept of the visualization is surprisingly simple, note that almost every aspect in
LL parsing can be explained by means of this paradigm. The stack for a certain derivation can
for example be visualized as the path of nonterminals that the parser is currently considering.
This is also illustrated in Figure 4.4. Furthermore, note that the progress inside an alternate in
fact corresponds to a grammar slot, where the arrows in white still remain to be considered. As
soon as the parser has managed to find a valid derivation, the parse tree of that derivation can be
obtained by simply looking at all the literals in the visualization that were parsed successfully.
4.2.1 GLL node

Every node in the so called GLL tree is constructed in such a way that it serves two purposes. From the viewpoint of a derivation, a GLL node summarizes the status how far an alternate has been parsed. From the viewpoint of the control flow, every node corresponds to a collection of functions that the GLL parser is able to jump to depending on the grammar slot it is interested in. This way, we can represent code jumping from one grammar slot to another as a “jump” performed by the walking figure. A GLL node mainly consists of four parts that are illustrated in red in Figure 4.5.

1. Alternate label [1]: The alternate label of a GLL node indicates which alternate of a production rule is considered by the algorithm. The label for an alternate is defined according to the labelling scheme as described in Section 3.5.2. Especially for situations where production rule are allowed to have multiple alternates consisting of the same literals (e.g. S ::= a | a), this label indicates which of two alternates is considered by the parser.

2. Block arrows [2]: In Section 3.5 we saw that for the realization of an explicit stack, the parsing of an alternate has to be decomposed into GLL blocks. We can represent this decomposition by splitting the literals of an alternate into separate blocks as defined by the
function decomposition in Section 3.5. Figure 4.6 shows an example how this decomposition is illustrated in the visualization for alternates $S ::= aS$ and $S ::= ASd$ respectively.

![Figure 4.6: Illustration how GLL blocks of alternates $S ::= aS$ and $S ::= ASd$ can be represented by means of block arrows.](image)

In order to show the relationship between the block arrows and the corresponding functions that are invoked, every block arrow is provided with its corresponding function label. In case of grammar slot $S ::= A \cdot S d$, according to Table 3.1, this corresponds to parsing the block arrow with label $S1_0()$.

3. Walking figure [3]: As mentioned before, when the GLL parser is considering a certain alternate, we can reflect the current position of the parser with respect to that alternate as a walking figure. In case of unambiguous grammars, there is always one walking figure visible at the same time. Together with the progress that is shown in the block arrows, the walking figure in Figure 4.7 clearly illustrates that the parser has parsed $a$ and is about to encounter nonterminal $S$.

![Figure 4.7: Illustration how a walking figure can be used to denote the progress of a GLL parser.](image)

4. Input string [4]: The input string illustrates which part of the input string still remains to be parsed. As the parser manages to parse new literals, the status of a string is updated by emphasizing the literals that have been parsed so far. This way, the user can see where the parser remained in the input string. The first valid question one may ask is why to place the input string inside a GLL node? Why not visualizing the input string as a whole outside the scope of the GLL tree? The main reason behind this decision is that for the visualization of a descriptor, the part of the input string that still remains to be parsed plays a very important role.

60 Exploring and visualizing GLL parsing
So now the main question remains is how a descriptor can be represented in this paradigm? Recall from Section 3.6 a descriptor represents a point of continue for one or more derivations. If we translate the components of a descriptor \((g,c,i,s)\) into the concepts that have been introduced, we can see that:

- The grammar slot of a descriptor corresponds to the beginning of a block arrow in a GLL node. From the viewpoint of the walking figure, this is the position in the alternate where the parser must start. Figure 4.5 illustrates the start position of the walking figure in case it must parse grammar slot \(S ::= A \cdot S \cdot d\).

- The GSS node represents the top node of one or more stacks. As illustrated in Figure 4.4, we can represent a stack a path from the root of the GLL tree to the GLL node corresponding to the top of that stack. Hence, the GSS node represents the collection of GLL nodes for which the parser has to parse grammar slot \(g\). Figure 4.8 a) shows an example how the GSS node \((S ::= A \cdot S \cdot d, ad\$)\) as defined in Figure 4.8 b) corresponds to two locations in the GLL tree (indicated in yellow), one for every derivation.

![Figure 4.8: Visualization of a GSS node consisting of multiple stacks. This figure uses colouring to illustrate the resemblance between a GSS trace and a trace in the GLL tree.](image)

Note that the input string in both GLL nodes does not start from the beginning, but corresponds to \(ad\$\) as stored in the GSS node. The first reason for doing so is that for large input strings, visualizing the entire input string in a GLL node is not scalable. Besides, note that the result of parsing of the literals that were parsed earlier is also reflected in the GLL tree that was constructed so far. A second reason for only visualizing the remaining part of the input string is that we can use this information to determine whether two derivations have some overlap. In Section 4.2.3 we will see that the visualization of the input string in this manner, allows us to detect any sharing by checking whether the status of two GLL nodes are visually equal to each other.

- The input position indicates how far the input string is currently parsed and where the parser must continue. As illustrated earlier, we can visualize this progress for instance by emphasizing the literals that the parser already managed to parse successfully.
Hence, for every stack represented by \( c \), we can summarize a descriptor as a location in a GLL node where the walking figure must jump to starting from the input position as denoted by \( i \). Figure 4.9 shows how we can visualize the descriptor \((S ::= a S \cdot, \langle S ::= A S \cdot d, aad\$, d$\rangle, d$\)) in case \( c \) consists of one stack. The walking figure in gray here is used to denote the return location of the parser as defined by the GSS node.

Figure 4.9: Visualization of a descriptor. Note that we can construct a visualization of a descriptor without having to reason where it is going to be used for.
4.2.2 GLL Tasks: lifetime of a descriptor

In order to visualize the control flow of the GLL algorithm, it is important to understand the steps that a GLL parser has to perform in order to process a descriptor. When looking at the resulting code of the GLL parser, you will notice that there is a high level structure of statements that are happening when invoking a certain function. By showing how the GLL parser responds to these functions, the user can see why these functions are necessary in the first place. In order to give the user an idea where the parser resides in the processing of a descriptor, the visualization shows a list of all steps that the algorithm performs. As described in the OOGLL algorithm, the most important tasks inside the GLL parser are the functions Goto(), test(), add(), create(), pop(), getNodeT(), and getNodeP(). Note that the last two statements are in particular relevant for the parser, since they are used to construct the SPPF data structure.

Figure 4.10: The introduction of GLL tasks can say much about the work that currently been performed by the descriptor in progress.

Figure 4.10 shows how this task set can be used to visualize the current state of the parser. Note that depending on the outcome of evaluations such scanner.hasNext(test(S)) or \( U \).contains(\( d \)), the control flow of the algorithm can become significantly different. To illustrate these changes in control flow, every task is provided with a status symbol, where a check sign indicates the successful execution of a statement and a cross the exact opposite. The task set now clearly illustrates that the parser is almost finished with the current descriptor and that apparently the first alternate of \( S \) was unable to able the next part of the input string.

Since every statement affects the parser in a certain way, it is important that the result of performing a statement is clearly reflected in the visualization. In order to illustrate how the result of a GLL task can be visualized, we will explain the execution of every statement when processing the descriptor \( S ::= A \cdot S \ d, \langle \bot, aad\$ \rangle, aad\$ \). For the illustration of a pop() and scan() statement, we will look at the processing of \( A ::= a, \langle S ::= A \cdot S \ d, aad\$,aad\$ \). For the illustration of a pop() and scan() statement, we will look at the processing of \( A ::= a, \langle S ::= A \cdot S \ d, aad\$,aad\$ \).

1. Goto(): As mentioned before, we can model the execution of a Goto() literally as a jump of the walking figure from one location to another. The first statement of a descriptors
always corresponds to a Goto() which should take the parser to the right parsing function. To illustrate that before the execution of this statement, the parser has not performed this jump yet, initially, the walking figure is position outside the GLL node that is of interest. If we consider the first task of our example descriptor. The result of performing a Goto(S::= A · S d) statement can for instance now be translated to a jump of the walking figure to the beginning of the block arrow for S. This is also illustrated in Figure 4.11.

Figure 4.11: The execution of a Goto() statement to a match function.

When looking closer to the execution of a GLL parser, we can see that the Goto() statement can be performed at most twice when processing a certain descriptor. The first Goto() invocation ensures that the parser starts at the right match function, whereas a second Goto() invocation can be necessary to jump to an alternate function for the creation of descriptors. Since our example descriptor tries to parse nonterminal S, the second Goto() statement becomes necessary to model non-determinism. We can model the execution of this statement again as a jump, but now from within the scope of the function for S ::= A · S d. This is also illustrated in Figure 4.12.

Figure 4.12: The execution of a Goto() statement to an alternate function.

2. test(): The test() function performs a lookahead check in the input string to see whether the creation of a descriptor is useful. Since this is merely a check that determines how the GLL parser must proceed, we can visualize the result of this check by adding a status symbol to
the GLL task and provide a brief motivation why this check was successful or not. Figure 4.13 illustrates the two scenarios that the test() statement can return.

Figure 4.13: The execution of a test() statement either successful or unsuccessful. Note that the tree visualization in both cases remains the same.

3. Create(): The create() statement is responsible for the creation of a GSS node (and edge) and has to make sure that derivations for which descriptors have been processed before can continue their work anyway. Since the creation of such node is merely an administrative step in the algorithm, visualizing this creation in the GLL tree is rather difficult. To show the result of this statement anyway, there was decided to create a separate visualization of the GSS where we can represent the current GSS node in the parser by means of a pointer. Figure 4.14 shows the result before and after executing a create() statement for our example descriptor.

Figure 4.14: The execution of a create() statement can be illustrated by means a GSS visualization.
In the situation where the GSS node already exists, we can inform the user of this event by means of a popup. More about this in Section 4.4.7. In the next section we will discuss how the visualization handles sharing in the GSS.

4. Add(): The add() statement is responsible for the creation of a new descriptor. Note however that this descriptor is added to the set $R$ to be processed later on. To denote that this descriptor still has to be processed in the future, a sitting figure is added to the GLL node(s) corresponding to this descriptor. The user can now select a certain alternate by clicking on one of the sitting figures. Figure 4.15 illustrates the add() operation for our example descriptor. When the GLL tree has not considered a certain alternate before, the add statement creates a new GLL node to represent this alternate.

![Figure 4.15: We can visualize the creation of a descriptor by means of a sitting walking figure.](image)

Since a descriptor can be relevant for multiple derivations at the same time, recall that it is possible that the visualization of a descriptor corresponds to one or more GLL nodes in the visualization. To visualize that these derivations have this descriptor in common, such walking figure has to be added to all these GLL nodes. Figure 4.16 shows an example where two sitting figure together represent the processing of one descriptor.

![Figure 4.16: Depending on the derivations that are interested in a descriptor, the execution of a add() statement can result into one or more sitting walkers.](image)
5. **Scan()**: The scan() operation corresponds to the scanner.next() statements in the algorithm and is responsible for reading the actual input string. We can easily visualize the effect of this statement by adjusting the progress inside an alternate. To indicate that the algorithm is now considering the next literal, we can move the walking figure to the next literal in a GLL block. The result whether the scan() was successful can be visualized. Figure 4.17 shows the result of applying a scan() operation.

6. **Pop()**: As mentioned before, after a nonterminal has been parsed successfully, the pop() statement uses the GSS to obtain the next grammar slot where the parser must continue with the current derivations. The visualization of a pop() can be demonstrated by taking the grammar slot from the stack after which an add() statement uses this information to create the next descriptor. To indicate that the parser has finished an alternate, the walking figure is positioned at the end of the last GLL block. Figure 4.18 shows the result of applying a pop() statement under the assumption that there is only one derivation that is interested in the descriptor for $S ::= A \cdot S$.

---

**Figure 4.17**: We can visualize the progress of a scan() by adjusting the status of a GLL node.

**Figure 4.18**: A GSS visualization is used to show the interaction between GLL parser and GSS.
7. Return(): The return statement represents the end of processing a descriptor.

8. GetNodeT() and getNodeP(): The getNodeT() and getNodeP() are responsible for building the resulting SPPF. Since the execution of these statements does not affect the control flow of the GLL parser, visualizing this step in the GLL tree is rather difficult. Similarly, for the visualization of the create() statement, a separate view is used to visualize the construction of the SPPF over time. Again by means of pointers we can represent variables $C_N$ and $C_R$ in the resulting visualization. More about this in Section 4.4.6. Note that if we omit the operations getNodeT() and getNodeP() from the visualization, the resulting visualization can also be used to illustrate how a GLL recognizer works.

4.2.3 Visualizing sharing

Recall from Section 3.6.3 that it is possible for a GLL parser to process descriptors before all derivations that are interested in this descriptor have been discovered. If we return to the execution trace in this section, we discovered that depending on the order in which descriptors ($S ::= \cdot A S \ d, \langle \bot, aad$\rangle, aad$)$ and ($A ::= \cdot a, \langle S ::= A \cdot S \ d, ad$\rangle, ad$)$ are processed, the GLL parser was able to detect this sharing beforehand. In order to demonstrate how sharing can be reflected in the visualization, let us consider both orderings separately.

1. ($S ::= \cdot A S \ d, \langle \bot, aad$\rangle, aad$)$ processed before ($A ::= \cdot a, \langle S ::= A \cdot S \ d, ad$\rangle, ad$)$

Recall from Section 3.6.3 that in this situation sharing in the GSS is detected before the descriptor for $A$ is processed. Hence we know that when we are processing the descriptor for $A$, all interested derivations are known beforehand. From the viewpoint of the walking figure, we can visualize the processing of this descriptor as walking both derivations in parallel. This is also illustrated in Figure 4.19. Note that we can clearly see that both derivations have some overlap, since the yellow GLL node are visually identical to each other.

Figure 4.19: Walking derivations in parallel.

Since both derivations are interested in the same descriptor, we know that both derivations have to parse the same literals starting from the same grammar slot. The steps that the walking figures will perform are therefore completely identical to each other.
2. \( (A::=\cdot a, \langle S::=A \cdot S d, \text{ad}$ \rangle, \text{ad}$) \) processed before \( (S::=\cdot A S d, \langle \perp, \text{aad}$ \rangle, \text{aad}$) \)

Recall in this particular case that the descriptor for \( A \) is processed before the second derivation that is also interested in this descriptor has been discovered. To illustrate this situation, the visualization will show the processing of descriptor \( A \) as if all derivations have been discovered. Since there is only one derivation known at the time this descriptor is processed, the visualization will look like as if the grammar is unambiguous. Figure 4.20 shows the visualization when processing the descriptor for \( A \).

Figure 4.20: Parse the descriptor for \( A \) as if all derivations have been discovered.

Let us now consider the situation where the GLL algorithm is about to detect late sharing. Recall from Section 3.6.3 that this is detected when processing descriptor \( (S::=\cdot A S d, \langle S ::= A S \cdot d, \text{ad}$ \rangle, \text{ad}$) \). At the occurrence of the create() statement, the GLL parser will notice that it has seen the parsing of \( S ::= A S d \) against the string \text{ad}$ before in another derivation. The visualization illustrates this late sharing by introducing a shadow walker showing the steps that the algorithm took when it was processing the descriptor for \( A \). By walking again both derivations in parallel we can illustrate which parts of GLL tree both derivations have in common. This is also depicted in Figure 4.21 b). The goal of this shadow walking in essence is to synchronize the results of both derivations thereby implicitly explaining the usage of the \( P \) data structure. Figure 4.21 shows how the visualization explains late sharing from the moment it is detected until the end of processing the descriptor.

After the descriptor has been added to the visualization (Figure 4.21 c)), the algorithm will continue with the two walkers when scheduling new descriptors using \( S() \). Since the descriptor for \( A \) has already been processed, it is skipped by the algorithm. We can illustrate the latter by indicating that the node for \( A0 \) has already been considered by the shadow walker. This is also illustrated in Figure 4.21 d).
Figure 4.21: a) The moment in the visualization where sharing is detected.  b) Use a shadow walker to view the parse result $A$ that has been discovered in the past.  c) By copying the parse result to $A$ the user is aware that this work has been done before.  d) descriptor for $A$ is duplicate, since the GLL nodes $A_0$ have already been processed.
4.2.4 Descriptors paradox

The main idea of the requirements in Section 4.1 is that the at runtime visualization of the GLL algorithm serves two main purposes. On one hand it should provide the user a clear meaning of what a descriptor represents while on the other hand it must provide the user an overview of the parsing that the algorithm has already performed. Although the visualization as presented in the previous sections works well for the explanation of the steps in the algorithm, there is something fundamentally wrong when visualizing the notion of a descriptor. Recall from Section 3.6.3 that a descriptor represents an independent piece of work where one or more stacks are interested in. In Section 3.6.3 however we noticed that, due to late sharing in the GSS, a descriptor can be processed before all the interested stacks have been discovered. But how can we visualize a descriptor at runtime if the GLL parser has not detected this late sharing? And what happens to the visualization of a descriptor when this late sharing is detected? Be aware that in a step-by-step visualization of the algorithm at runtime, we can only visualize the the usage of a descriptor based on the knowledge that the parser has obtained thus far. To illustrate the problem, let us return to the execution trace as discussed in Section 3.6.3 and consider the visualization of the descriptor \((A ::= \cdot a, (S ::= A \cdot S d, ad$), ad$)\) before and after sharing is detected. These situations are illustrated in Figure 4.22 a) and Figure 4.22 b) respectively.

Before any sharing is detected, based on the knowledge that is known at that time, the descriptor for \(A\) was only relevant for one stack. If we would look at the same descriptor after this sharing was detected, then you will notice that the visualization has become different. Although this is perfectly understandable from the execution of the algorithm, from the viewpoint of a descriptor as an independent piece of work, this is rather counter-intuitive. So, in order to obtain a visualization of a descriptor that is consistent over time, all sharing between the stacks that are interested in this descriptor has to be known in advance. This is however contrary to the explanation of the control flow of a GLL parser, as it is its task to discover these derivations in the first place. So on one hand we would like to reason about descriptors as chunks of work that have to be recognized independent from the derivations that the algorithm is considering, but on the other hand we can only show the usage of a descriptor at runtime for the derivations that have been discovered so far. We will refer to this problem as the descriptors paradox. This main reason for this contrast has to do with the fact that a GLL parser tries to detect any sharing while it is still exploring new
derivations in the algorithm. The fact that we want to build a visualization at runtime however makes it impossible to know these derivations beforehand for the visualization of a descriptor.

If we would look at a descriptor before the GSS was introduced, the visualization as proposed in Section 3.6 can visualize the notion of a descriptor perfectly, since we know that such descriptor is only used by one stack. It is due to the detection of late sharing in the GSS that the visualization of a descriptor can change over time. The result of this paradox is that when designing the visualization, we have to take into account that the visualization of a descriptor depends on the state of the GSS and can make the resulting algorithm rather complex and tedious to program. Note that this remark does not necessary only have to hold in just the area of generalized parsing. In fact, any exploration algorithm that tries to obtain on-the-fly sharing by means of a centralized data structure is likely to end up with the same paradox when explaining the control flow of the algorithm. The fact that the algorithm may discover new sharing as it is still exploring other parts of the solution space, makes it difficult to visualize the result of processing a chunk of work without knowing all the parts where this chunk may occur.

4.2.5 Limitations and difficulties

Note that the visualization shows a lot of resemblance with the step-by-step illustration of an LLRD recognizer in Figure 3.1 which makes the concept very suitable to explain the bridge between LLRD and GLL parsing. There are however also a few limitations to this visualization approach. The first limitation of this approach is that unfolding every non-deterministic choice in the grammar can result in a large tree structure. The width of the resulting tree in fact grows linearly in the number of alternates in the grammar whereas the depth of the tree is linear in the size of the input string. As a consequence, do note that for large grammars and input strings the scalability of the visualization can definitely become an issue. Although there are some solutions presented in Chapter 5 to reduce the size of the tree, more future work will be necessary to realize these techniques properly.

A second limitation is that the concept may not be finite for certain context-free grammars. The ground assumptions behind the visualization is that every derivation can be represented as a tree structure and the number of derivations to be visualized is finite. Note however that for SPPF cyclic grammars such as $\Gamma_6$, the last assumption is violated. Since the explanation of parsing a SPPF cyclic grammar is no different from the explanation of generalized parsing in general, there was decided to only focus on the visualization of grammars that produce acyclic SPPFs. The effect of this assumption is that the resulting SPPF is always a DAG structure which allows us to summarize the progress of a GLL parser by means of a finite tree structure. The difficulties that one may encounter when visualizing left-recursive grammars is discussed in Section 4.4.9.

Be aware that the main difficulty of the visualization does not lie in the concept, but rather in the explanation how the GLL algorithm can be shown in this format. For instance how do we know what derivations a GLL parser is currently discovering? And how can we illustrate for instance sharing between two or more derivations? In fact, it turns out that the translation of a GLL parser to this format brings the necessary difficulties. These will be discussed throughout Section 4.4.
4.3 Context-highlighting

The concept as proposed in Section 4.2 uses a single tree to visualize the status of a GLL parser. Unfortunately, the use of such a simple concept does come with a price when considering (highly) ambiguous grammars. Although the ambiguities in Figure 4.19 are well separated from each other, one can imagine that this is not always the case. To illustrate the latter, consider the following two partial derivations in Figure 4.23.

If we want to visualize these derivations into one GLL tree, you will notice that the GLL nodes S1, A0, and S2 correspond to the same nodes in the tree. This is mainly due to the fact that both derivations have made the same non-deterministic choices except for the S in S ::= A S d. So now the main question arise, should we visualize the progress of S1 according to derivation 4.23 a) or b)? The answer of this question depends on the descriptor that the algorithm is currently processing. Recall from Section 3.6.3 that both derivations correspond to different descriptors, since derivation a) has already recognized a larger part of the input string compared to derivation b). Hence, depending on the descriptor the algorithm is looking into, the status of a GLL node can become different. Since it is impossible to present all derivations at once in the GLL tree, we have to make sure that the visualization only shows those derivations that are of interest when processing the current descriptor. One way of realizing this is to highlight all GLL nodes of the derivations that are involved when processing this descriptor. Figure 4.24 shows how we can use highlighting to show the progress of both derivations separately in the resulting GLL tree.
Figure 4.24: By means of highlighting we can show the progress of a certain derivation.

Note that the visualization of every partial derivation in a separate tree structure is not an option, as the number of derivations already can become large for small grammars such as Grammar $\Gamma_5$. Not only does the size of the visualization become impractical, the user then has to consider even more visualizations in parallel, making the execution of the algorithm significantly harder to follow. So what do we do with the GLL nodes that are not highlighted? In general, there are several possibilities. The first option is to hide any progress that has been made with respect to that alternate. Although this is the most simple option to realize, note that the user will no longer have an intuition of how far the GLL parser managed to parse other derivations. Figure 4.25 show the result of applying this strategy.

Figure 4.25: Although hiding the progress of a GLL node is easy to realize, it does not provide the user any knowledge about the other derivation.
The second option is to determine the status of a GLL node based on the derivations that have been parsed so far. In the situation where a GLL node only belongs to one specific derivation, the problem can be easily solved by assigning the GLL node the status with respect to that derivation. Unfortunately, since a GLL node in general can be used by multiple derivations, some strategy has to be used to assign the most appropriate status. To illustrate the concept, we will discuss two possible strategies, namely LIFO and MostComplete. The LIFO strategy assigns a non-active GLL node its status according to the last derivation where the GLL node participated in. The main advantage of this strategy is that when the GLL parser processes a different descriptor, the work that has been performed by the previous descriptor is still clearly visible. A downside however is that when the previous descriptor did not manage to parse a literal successfully, the progress of a GLL node is coloured red even when it still can be part of a successful derivation. Figure 4.26 shows a scenario in which the LIFO strategy may not be so desired.

Figure 4.26: There are situations where the LIFO strategy may not result in the best highlighting.

Note that in this example, if we assume that the derivation $S \rightarrow ASD \rightarrow aSd \rightarrow aASDd$ is considered last, the first GLL node $S_1$ is colored red, since the $d$ has already been parsed by the $S$ in $A S d$. The derivation $S \rightarrow ASD \rightarrow aSd \rightarrow aaSd \cdots$ on the other hand is clearly an example where $S_1$ has been parsed successfully. The MostComplete strategy tries to overcome the issue of the LIFO strategy by assigning a GLL node its status according to the derivation that managed to parse most of the input string. Note however that in case a GLL node belongs to multiple derivations, this strategy becomes non-deterministic. Figure 4.28 show an example of such situation.
$S ::= ASd \mid aS \mid \varepsilon$  
$A ::= a \mid aa$  

\[ \Gamma_8 \]

Figure 4.27: In the situation where two derivations parsed the same input string using different alternates, the MostComplete strategy may provide a deterministic answer for Grammar $\Gamma_8$.

The MostComplete strategy in this case can not decide whether block arrow $S$ must be highlighted in the resulting GLL tree, since both derivations managed to parse “aa”. When designing the algorithm for context-highlighting, we will discuss how both strategies can be realized. Since a descriptor can be used to perform parse work for more than one derivation, be aware that due to sharing the GLL tree may have to highlight multiple derivations at the same time. Figure 4.28 even shows an example of situation where two derivations can have a GLL node in common while they are both interested in the same descriptor.

Figure 4.28: Depending on the derivations the user is interested, highlighting can be different.

Note that there are only two valid ways of highlighting the nodes such that they represent derivations. The choice whether nodes must be highlighted according to Figure 4.28 a) or b) does not matter for the outcome of the parser, since both partial derivations represent the parsing of the same input string. Hence, we can see that some point user input is required to obtain the desired highlighting. This is also discussed in Section 4.4.8.
4.4 Constructing the visualization

This section describes the algorithms and data structures that have been introduced for the realization of the visualization as discussed in Section 4.2. Furthermore, this section will discuss some of the difficulties that one may encounter when realizing certain parts of the visualization.

4.4.1 Data management

One important requirement for the visualization is that the generation of the resulting image is state-less, meaning that the generation of the visualization can be realized by means of a snapshot of the algorithm’s data structures. In order to construct the GLL tree for an arbitrary GLL parser, we need to find a way to extract the progress of each derivation out of the information that is provided by the algorithm. The first data structure that seems to be a suitable candidate to perform this extraction is the GSS. Keep in mind that every path in the GSS corresponds to a stack that in turn can be visualized as a path in the resulting GLL tree. Although the progress of a derivation can directly be derived from the stack, note that a stack only stores the literals that remain to be parsed. As soon as a nonterminal was parsed successfully, the data is removed from the stack and the knowledge of that nonterminal is gone. Although a GSS does not remove any nodes from the data structure, it can still be difficult to obtain this data. Especially in situations where two derivations have GLL nodes in common, the structure of the GSS is not sufficient to derive the GLL tree. Figure 4.29 shows an example of such situation. This GLL tree corresponds to the one as illustrated in Section 4.28.

![Figure 4.29](image)

Figure 4.29: In case of sharing within the GLL tree, the GSS data structure is not very suitable for the generation of the GLL tree.

Note that number of GSS nodes does not correspond to the nodes in the GLL tree. Be aware however that the GSS only stores grammar slots that still remain to be parsed after a nonterminal has been considered. Hence, whether nonterminal A of $S ::= A S d$ is parsed as $A_0$ or as $A_1$ does
not matter for return location of the parser. Both alternates eventually have to jump to $S ::= A \cdot S \cdot d$. The main problem with the GSS data structure is that the recognition of $S ::= A \cdot S \cdot d$ and $S ::= A S \cdot d$ occur as siblings in the GSS graph. Not only will the generation algorithm be not that pleasant to design, note that in this particular situation it is difficult to determine which parts of the GSS correspond to a certain derivation. Besides, note that the GSS cannot be used to generate GLL nodes as a result of an add() statement (e.g. GLL node $S_0$ in Figure 4.29), since these nodes still have to be considered by the algorithm. Generating the GLL tree by means of the SPPF data structure is also not going to work, since this data structure only stores the derivations that were actually successful with respect to the input string. Hence we will have to find a different way of obtaining this information.

Recall from Section 3.6 that a descriptor is created whenever there is a non-deterministic choice for a nonterminal. Hence, if we can store the ordering in which these descriptors are created, we are able to derive how the discovery of new derivations are realized by the parser. Figure 4.30 shows the resemblance between the proposed hierarchy (Figure 4.30 a)) and the GLL tree (Figure 4.30 b)). For the sake of readability and space, let us consider descriptors without their SPPF nodes and represent the grammar slots of a descriptor by means of the labelling scheme as defined in Section 3.5.2.

![Diagram](image)

Figure 4.30: A hierarchy of how descriptors are created in the algorithm (depicted in a)) can be useful for the construction of the GLL tree (depicted in b).

Note that in contrast to the GSS data structure, the parsing of GLL blocks within the same alternate are stored as a chain in the data structure of Figure 4.30 a), thereby making it more easy to distinguish one derivation from another. If we look closer to the mapping between a descriptor node in the data structure of Figure 4.30 a) and a GLL node in Figure 4.30 b), then we see that all descriptors with the same GSS node and prefix in their label (e.g. $(S_1, \langle \bot, 0 \rangle, 0)$ vs. $(S_1, \langle \bot, 0 \rangle, 1)$) correspond to the same GLL node. Intuitively this makes sense, since every descriptor satisfying this condition is responsible for parsing a specific GLL block within the scope of the same alternate. What makes this property so interesting is that we can now easily determine the subbranches of a GLL node by looking at the descriptors that are encapsulated by the descriptors with the same colouring. To illustrate the latter, Let us consider the nodes $(S_1, \langle \bot, 0 \rangle, 0)$ and $(S_1, \langle \bot, 0 \rangle, 1)$ in Figure 4.30 a). Recall from Section 3.5.2 that in order for a GLL parser to create a descriptor $S_1,0$, first the descriptor for $S_1$ has to be processed successfully. Since in between the processing of $S_1$ and $S_1,0$ nonterminal $A$ is parsed, we know that all nodes
that occur in between \((S1, (\bot,0), 0)\) and \((S1, (\bot,0), 1)\) represent the parsing of \(A\). Similarly, if we look at the subgraph encapsulated by the nodes \((S1, (\bot,0), 1)\), \((S1, (\bot,0), 1)\), and \((S1, (\bot,0), 2)\), then we can see that for the parsing of \(S\) there are apparently two possibilities. Note that the construction of these subbranches is again an instance of the original tree generation problem and can be solved recursively.

The data structure as proposed in Figure 4.30 a) will be referred to a descriptor graph. A descriptor graph is a graph structure where every node corresponds to a descriptor and there is an edge from \(u\) to \(v\) if and only if descriptor \(u\) invokes an add() statement to create a descriptor for \(v\). Even when it turns out that \(v\) has already been created before, the descriptor graph will still store this information as if \(v\) did not exist before. The reason for doing so will become clear if we consider late sharing in the GSS. Recall that if the GLL algorithm discovers sharing between derivations after the corresponding descriptor has been processed, it uses the set \(P\) to avoid parsing the same descriptor twice. If we would consider the late sharing example of Section 3.6.3 again, then the resulting descriptor graph would look like as depicted in Figure 4.31.

![Figure 4.31: Descriptor graph for the input string aad$ and Grammar $\Gamma_5$. The nodes in red represent locations where duplicate descriptors are created.](image)

Be aware that the descriptors nodes \((A0, (S1_0,1), 1)\), \((S2, (S1_1,2), 2)\), \((S2, (S0,2), 2)\) occur twice in the descriptor graph. These are all descriptors for the example input \(aad$\) for which at some point sharing has been detected. The nodes in green represent the first time that these descriptors have been created by the algorithm. The nodes in red are the descriptors for which the statement \(U\).contains(d) in the algorithm returned true.

If we zoom in on the resulting graph (Figure 4.32), then we can see that due to late sharing the descriptor for \((S1, (S0,1), 1)\) will immediately create a descriptor for \((S1, (S0,0), 1)\), 2). In order to know however how the GLL tree for \(S1\) must look like, we must have some kind of reference to the GLL tree that would have been created if this sharing would not have happened. Hence, at every place in the algorithm where a descriptor is about to be created twice the descriptor graph will create a so called duplicate descriptor. By looking at the graph that was created by
the corresponding green descriptor, we are able to construct the GLL tree for S1.

One of the interesting properties of a descriptor graph is that the order in which the descriptors are processed does not influence the structure of the graph. Note that the ordering only determines which of the descriptor nodes are considered duplicate. Further note that the leaf nodes of a descriptor graph either represent descriptors that occur in $\mathcal{R}$, or descriptors that already have been processed but did not result in the creation of new descriptors, and duplicate descriptors. If we ignore the last category, then we can see that every path from leaf to root in essence shows the course of a certain derivation. This property in particular becomes interesting when considering for instance context-highlighting in Section 4.4.8.

**GLL tree generation**

The algorithm to generate the resulting GLL tree is defined in Listing 4.1. The main idea is fairly simple. The generation of the GLL tree corresponds to a depth first search traversal of the descriptor graph where there is a GLL node created whenever a descriptor with grammar slot $S ::= \alpha \beta$ is encountered. Note that if the grammar slot of a descriptor does not correspond to the first GLL block, we know that the GLL nodes for this descriptor have already been created before by another descriptor. This follows directly from the fact that a descriptor with grammar slot $S ::= \alpha \cdot \beta$ is only created whenever $\alpha$ has successfully been parsed by other descriptors.

As soon as the algorithm encounters a descriptor that has one or more children that are duplicate, then the algorithm first has to generate a GLL tree corresponding to these descriptors. It does this so by going to the equivalent descriptor node that was not marked as a duplicate and finds the part of the descriptor graph that corresponds to this node. As illustrated in Figure 4.30 before, this subgraph is encapsulated between descriptor nodes whose GSS node and prefix label are equal to each other. Listing 4.1 shows the resulting code of the algorithm.
private Descriptor createGLLGraph(Descriptor descriptor, DescriptorGraph graph, GLLNode currentNode) {
    Let descriptor be of the form \((g, c, i, s)\) and \(g\) of the form \(S ::= \alpha \cdot \beta\);
    Let currentBlock denote the GLL block corresponding to \(g\)
    Descriptor pointer = descriptor;

    for every child \(d\) of \(\text{descriptor}\) in \(\text{graph}\) {
        if \((d\) is a duplicate descriptor) {
            Let \(d'\) be the descriptor node of which \(d\) is a duplicate
            DescriptorGraph subGraph = new DescriptorGraph(descriptor);
            GSSNode gssn = d'.getParent().getGSSNode();
            GrammarSlot gram = d'.getParent().getGrammarSlot();
            getSubGraph(d', graph, gssn, gram, subGraph);
            subGraph.addNode(descriptor);
            subGraph.addEdge(descriptor, subGraph.getRoot());
            createGLLGraph(descriptor, subGraph, currentNode);
        }
        if (\(\alpha == [\]\)) {
            Create GLL node \(\text{gllNode}\) with parent \(\text{currentNode}\).
            Create and edge from GLL block \(\text{currentBlock}\) to \(\text{gllNode}\).
            pointer = createGLLGraph(child, graph, gllNode);
        } else {
            gllNode n = currentNode;
            while (!n was created by descriptor \(e = (g', c', i', s')\) with
                \(g' = S ::= \cdot \alpha \beta\), and \(c' = c\) ) {
                n = n.getParent();
                pointer = createGLLGraph(child, graph, n);
            }
            if (\(\mathcal{R}.\text{contains}(\text{child})\)) {
                add sitting walker to \(\text{gllNode}\)
            }
            return pointer;
        }
    }

    private getSubGraph(Descriptor descriptor, DescriptorGraph graph, GSSNode n, GrammarSlot g, DescriptorGraph acc) {
        Let \(g\) be of the form \(S ::= \alpha \cdot \beta\)
        for every child \(d\) of \(\text{descriptor}\) in \(\text{graph}\) {
            if (! (\(d\).getGrammarSlot() \(== S ::= \cdot \alpha \beta\)) || ! (\(d\).getGSSNode() \(== n\)) { \(\text{acc}\).addNode(d);
                acc.addEdge(descriptor, d);
                getSubGraph(d, graph, n, g, acc);
            }
        }
    }
}

Listing 4.1: Pseudocode generation GLL tree

The algorithm initially starts at the root of the descriptor graph for which a GLL node \(S'\) has already been created. Whenever the algorithm encounters a descriptor \(d'\) whose grammar slot does not represent the beginning of an alternate, we know that the GLL node for this descriptor has already been created before. In order to know where the GLL branches of this descriptor must be attached to, the algorithm traces the descriptor graph back to the root until the GLL node corresponding to \(d'\) in that path has been found.

Even though a descriptor can represent multiple GLL nodes at once, note that for non left-recursive grammars it can never be the case that two GLL nodes of the same descriptor occur along a trace...
in the descriptor graph. This can explained by means of Figure 4.33.

Recall from Section 3.4 that for non left-recursive grammars, the GSS data structures does not contain any cycles. Hence we know that every GSS node can only occur at most once in every path in the GSS. Recall from Section 4.2 that there is a one-to-one correspondence between a stack and a path in the GLL tree. Hence we know that two GLL nodes representing the same alternate can only occur below one another if at least one character in the input string has been parsed/recognizer. This means that we can always find the GLL node for a descriptor node by looking at the descriptors’ GSS node that created it. This is also one of the reasons why visualizing left-recursion can become an issue using this approach. More about this in Section 4.4.9.

To ensure that the algorithm knows where to continue in the descriptor graph after the GLL tree for the duplicate descriptor has been generated, a pointer variable is introduced. The function getSubGraph() finds the subgraph in the descriptor graph that corresponds to the parse result of processing the duplicate descriptor. As soon as the algorithm encounters a descriptor that still has to be processed, the GLL nodes corresponding to this descriptors are accommodated with a sitting walking figure.

Figure 4.33: For non left-recursive grammars, in order for two GLL nodes to occur below one another, the must at least differ in the inputstring (indicated by red boxes)
4.4.2 Step-by-Step visualization

For the visualization of GLL tasks, we noticed that for some cases a separate visualization of the GSS and SPPF are desired. A disadvantage of the multiple views in the application is that the execution of a GLL task can now result into multiple changes in the visualization. Especially when these changes are unexpected, the user may not be able to grasp all these changes at once. In order to increase the usability and understandability, it can be desired to reverse steps and restart from at an earlier point in the algorithm. This way, the user can quickly revisit the parts of the visualization that were missed without having to view the execution of the algorithm from the start. But how to realize this "debugging" functionality at runtime?

The GLLExplorer system creates an at runtime visualization by performing the rendering of the algorithm in parallel with the execution of the algorithm. Whenever the visualization of the parser (or recognizer) is desired, the GLLExplorer system suspends the thread of the GLL algorithm after which the visualization gathers all data that is necessary to generate an image. As soon as the user wants to revisit a certain step, The GLLExplorer system uses the history to obtain the data structures with respect to that step. If the user decides to take a step that the parser did not reach yet, the execution of the algorithm is resumed until a certain condition has been met. Depending on the level of detail the user is interested, this condition can vary from the occurrence of the next GLL task to the moment in which a new descriptor is processed. Since the smallest step in the algorithm corresponds to the execution of a GLL task, there was decided to set the step size in the algorithm at the level of a GLL task.

If we would execute the GLL parser for Grammar $\Gamma_5$ for the input string $aadB$, then it would approximately require the execution of 150 GLL tasks to complete the parsing. One can imagine that storing the data structures $R$, $U$, $P$, GSS, and SPPF for every step completely even becomes impractical for small GLL parsers. Hence instead of copying data structures we now maintain a list of all descriptors, node, and edges that are created and removed throughout the execution of the algorithm. We can now model the state of the algorithm by means of a set of references to these lists (i.e. a record). Figure 4.34 illustrates how we can use one shadow copy of the data structures to represent the state of a data structure over time.

Unfortunately, the data structures as they are presented in the GLL parser are not suitable to store...
the data at this level of detail. Hence we will have to augment the GLL parser with additional
data structures to store this shadow administration. In Section 4.4.3 we will show how we can
maintain this shadow administration without obfuscating the code of the GLL parser.

We can have the user to select a certain descriptor by modifying the ordering in which the
descriptors are stored in \( R \). If the user decides to the change this ordering while it is looking
at historic data of the GLL parser, we have to take into account that all steps after this point of
execution were based on a different ordering of \( R \). Although the ordering in which descriptors are
processed do not influence the outcome of the parser, recall from Section 3.6.3 that this ordering
can seriously affect the way in which a GLL parser handles sharing. Hence in order to avoid
any inconsistencies in the administration, the state of the GLL parser is reverted to the moment
where \( R \) was modified. We can realize this by suspending the GLL parser before it is about to
process a new descriptor and replacing all data structures with the ones according to our shadow
administration at the desired moment in time.

When designing the shadow administration for the GLL algorithm, it is very important that
the reconstruction of the data structures and reverting mechanism are properly tested. Since
descriptors are always dropped once they have been processed, errors that are made in the re-
version are propagated implicitly throughout the creation/processing of descriptors. The result
of executing for instance a descriptor with an outdated reference will be very hard to discover in
the control flow, since it will likely result in the creation of other (faulty) descriptors. The fact
that a GLL parser determines its own control flow and uses centralized data structures to store
the state of the algorithm makes it very difficult to discover these errors. It is therefore highly
recommended that for the realization of the data management a database system is used.

4.4.3 Plug and Play

One of the main requirements of the GLLExplorer system is to make the usage of the visualization
flexible for any GLL parser. Keep in mind however that the visualization requires some shadow
administration in order to work properly. So instead of rewriting the OOG l algorithm to support
this administration, we will show how we can perform this administration inside the functionality
of the basic statements add(), pop(), create(), getNodeP(), getNodeT(), Goto(), init(), and \( S'() \).
Recall from Section 3.9 that the implementation of these statements are independent from the
input grammar that is provided by the user. As a consequence, we introduced an “AbstractParser”
class that contains the implementation of these statements for any GLL parser. As discussed in
Section 4.4.2, the introduction of a practical debugging mechanism requires the visualization to
maintain shadow administration for every data structure in the algorithm. If we can maintain
this administration within the scope of the abstract parser, then we are able to visualize arbitrary
GLL parsers without transforming them first to make them suitable for the visualization.

In order to realize the shadow administration, the abstract parser is augmented with data structures
for the storage of nodes, edges, and descriptors that have been added or removed by the GLL
parser over time. Note that for the the GSS, SPPF, and the descriptor graph only the addition
of nodes and edges have to be maintained, since they never remove data. As for the step-by-step
execution of the algorithm, the algorithm will be augmented with a suspend function to stop the
execution of the parser until the user decides to continue. Listing 1 shows the modified abstract
parser after applying these modifications.

```
1 public class AbstractParser {
  
  public Descriptor targetDescriptor;
  public AtomicInteger countDown;
  private List<Descriptor> addToR;
  private List<Descriptor> removeFromR;
  private Map<GSSNode, List<GSSNode>> addedGSSEdges;
```
private List<SPPFNode> addedSPPFNodes;
private Map<SPPFNode, List<SPPFNode>> addedSPPFEdges;
private Map<Descriptor, List<Descriptor>> addedDescriptorGraphEdges;

boolean S() {
    init();
    while(!stop){
        while(!R.isEmpty()){
            if (!U.isEmpty()){
                waitForResume();
            }
            (slot, node, i, s) = R.pop();
            waitForResume();
            CN = s;
            currentNode = node;
            scanner.setPosition(i);
            Goto(slot);
        }
    }
    return SPPFNode(S,0,\#I) has been created;
}

void create(GrammarSlot slot) {
    if there is no GSS node (slot, scanner.getPosition()) then create one{
        if there is no GSS edge from (slot, scanner.getPosition()) to Cu then create one
        add edge to addedGSSEdges
        waitForResume();
        for all (s, z) ∈ P with s = (slot, scanner.getPosition()){
            SPPFNode y = getNodeP(slot,CN,z);
            add(slot,Cu,z.getRightExtent(),y);
        }
    }
    return (slot, scanner.getPosition());
}

void pop() {
    waitForResume();
    if (Cu != ⟨⊥,0⟩) {
        P.add(Cu, scanner.getPosition());
        for each edge from Cu to v{
            Let SPPFNode z be on the edge of (Cu,v)
            Let Cu be of the form {slot,i}
            SPPFNode y = getNodeP(slot,CN,z);
            add(slot,v,scanner.getPosition(),y);
        }
    }
}

SymbolSPPFNode getNodeT(Terminal t) {
    Int leftExtent = scanner.getPosition();
    Int rightExtent = scanner.peek(t).length;
    waitForResume();
    if (!S.contains(SymbolSPPFNode(t, leftExtent, rightExtent))){
        create one
    }
    return SymbolSPPFNode(t, leftExtent, rightExtent);
}

SPPFNode getNodeP(Grammarslot slot, SPPFNode left, SPPFNode right) {
    waitForResume();
    Let slot be of the form A ::= α · β
    if (#α == 1 ∧ (head(α) is a terminal or a non–nullable nonterminal) ∧ β ≠ []){
        return right;
    } else{
        if (β ≠ []) { t = A; } else { t = A ::= α · β }
        if (left == $){
            ...
Let $i$ and $j$ be the left and right extents of SPPF node right
if ($\neg S$. contains $((t, i, j))$) {create one
if ($((t, i, j)$ does not have a packed node $(A \bydef \alpha \cdot \beta, i)$ {create one with child right
add $(t, i, j)$ and packed node to addedSPPFNodes
add edge from $(t, i, j)$ to packed node to addedSPPFEdges
add edge from packed node to right to addedSPPFEdges
}
else {
    Let $j$ and the right extent of SPPF node right
    Let $i$ and $k$ be the left and right extents of SPPF node left
    if ($\neg S$. contains $((t, i, j))$) {create one
    if ($((t, i, j)$ does not have a packed node $(A \bydef \alpha \cdot \beta, k)$ {create one with left child left and right child right
    add $(t, i, j)$ and packed nodes to addedSPPFNodes
    add edge from $(t, i, j)$ to packed node to addedSPPFEdges
    add edge from packed node to left to addedSPPFEdges
    add edge from packed node to right to addedSPPFEdges
}
    else {
        return $(t, i, j)$;
    }
}
}

void Goto(GrammarSlot slot) {
    waitForResume();
    ...
}

void add(GrammarSlot g, SPPFNode s) {
    Descriptor d = (g, $C_u$, scanner.getPosition(), s);
    waitForResume();
    if ($\neg U$. contains $(d)$) {
        $R$. add $(d)$;
        addedToR.add$(d)$;
        add edge from currentDescriptor to d to addedDescriptorGraphEdges
    } else {
        removedFromR.add$(d)$;
        DuplicateDescriptor $d'$ = d.copy();
        add edge from currentDescriptor to $d'$ to addedDescriptorGraphEdges
    }
}

protected void waitForResume() {
    if (waitForDescriptor) {
        if (currentDescriptor == targetDescriptor) {
            waitForDescriptor = false;
            Thread.suspend();
        }
    } else {
        if (countDown == 0) {
            Thread.suspend();
        } else {
            countDown--;  
        }
    }
}

@Override
public Set<Terminal> test(Literal l) {
    waitForResume();
    return super.test(l);
Listing 4.2 shows the strategic places where the parser thread suspends itself until the visualization gives permission to continue the execution. The waitForResume() function is responsible for this functionality and uses two variables countDown and targetDescriptor to determine whether suspension is desired. The countDown variable is an atomic integer that is decremented whenever the waitForResume() function is invoked. As long as the integer is larger than 0, the abstract parser will not invoke the Thread.suspend() function. The visualization can use this variable to suspend the parser after executing \( n \) GLL tasks by setting countDown = \( n \). In order to satisfy Requirement \( R_{003} \), the variable targetDescriptor is used to determine whether the algorithm is processing a certain descriptor. Note that if we want to skip the execution of the GLL parser until the next descriptor is processed, we can assign this variable the last descriptor that was added to \( R \). To ensure that the visualization is still usable after the GLL algorithm has terminated, an additional while loop with a stop variable is introduced. This way, whenever the user decides to revert to an earlier point of execution, the GLL parser is still capable in processing these descriptors.

Note that the waitForResume() function in the create() occurs after the GSS node and edge have been created by the parser. This way, when suspending the parser, the visualization can already show the user what the impact is of executing create() statement before it is finished. This is in particular useful if you want to notify the user beforehand whether sharing is about to occur. Note that the return statement as it is visualized by a GLL task does not happen explicitly inside a GLL parser, but is implicitly executed inside the call-stack. We know however that the result of this return statement always ends in the function \( S'(\cdot) \). Hence, by adding a second waitForResume() function in \( S'(\cdot) \) we can visualize the end of processing a descriptor without immediately starting with the next one. In order to visualize the execution of the test() statement, a waitForResume() function also has to be added inside the scanner.

### 4.4.4 GLL Tasks

The visualization of GLL tasks allows the user to reason how the operations such as add(), create(), pop(), etc. affect the control flow of the algorithm. In order to visualize such task set, for every grammar slot we must know which statements are invoked by the parser. In order to give the user an idea what he can be expecting in the future, the GLLExplorer system also visualizes the statements that still have to be performed. For the realization of this task set, the algorithm for obtaining this data is split in three parts. The first part extracts the task set of the parser by looking at the statements as they occur in the GLL parser. The second part maintains a list of all GLL tasks that have currently been executed by the GLL parser. The third part of the algorithm combines the data of the first two parts together into the desired task set. This is also shown in Figure 4.35.
Figure 4.35: Construction of GLL tasks.

Since this algorithm does not take the control flow the parser into account, every conditional statement in the algorithm has to be modelled as an non-deterministic choice. The result of applying this algorithm is a control flow graph [12] (in short CFG) for every function in the GLL parser. Note that these graphs can be easily precomputed when generating the parser using to the OOGLL algorithm. Rather that writing the generated statements to a text file, the OOGLL now has to maintain a graph for every function where every generated statement corresponds to a node and there is an edge between two statements \(a\) and \(b\) if and only if \(b\) is subsequently executed after \(a\). The if-statements in the code are responsible for any non-determinism in the graph as it can have two outcomes. Since the resulting pseudocode hardly differs from the OOGLL algorithm, for the sake of brevity, it has been omitted. Figure 4.1 shows the control flow graphs that are generated for our GLL parser for Grammar \(\Gamma_5\).

<table>
<thead>
<tr>
<th>function</th>
<th>CFG</th>
</tr>
</thead>
<tbody>
<tr>
<td>S()</td>
<td>![CFG diagram for S()]</td>
</tr>
<tr>
<td>A()</td>
<td>![CFG diagram for A()]</td>
</tr>
<tr>
<td>S0()</td>
<td>![CFG diagram for S0()]</td>
</tr>
<tr>
<td>S1()</td>
<td>![CFG diagram for S1()]</td>
</tr>
<tr>
<td>S2()</td>
<td>![CFG diagram for S2()]</td>
</tr>
</tbody>
</table>
The second part of the algorithm maintains the task set as it is currently executed by the GLL parser. This can be obtained by augmenting all statements in the abstract parser class with a function createTask(name, value, status, depth). This function simply creates a new task with name name, parameters as defined in value, with status status and indentation level depth. The name variable refers to the type of statement. Depending on the type of statement, additional information can be desired. For instance, knowing what part of the input string is scanned when performing a scan() statement is helpful to understand what is going on in the parser. The data that is maintained in value corresponds to the parameters that were passed along when executing the statement. The variable status can be empty, success, fail, or todo (indicating that a walking figure must be positioned with that task). We can use the depth of the call-stack to determine the level of indentation for every GLL task. Listing 4.3 illustrates how GLL tasks and the status of these tasks can be stored inside the abstract parser.

```java
1 public class AbstractParser {
2     :
3     private List<Task> performedTasks;
4     private int depth;
5     
6     boolean S() {
7         init();
8         depth = callStack.depth();
9         while (!stop){
10             while (!R.isEmpty()){
11                 if (!t.isEmpty()) {
12                     Task t = createTask("return", nil, "todo", 0);
13                     waitForResume();
14                 } else {
15                     (slot, node, i, s) = R.pop();
16                     scanner.setPosition(i);
17                     Goto(slot);
18                     C_N = s;
19                     currentNode = node;
20                 }
21             waitForResume();
22         }
23     }
24 }
```

Table 4.1: Control flow graphs for the parse functions as defined by the GLL parser for Grammar Γ₅.
CHAPTER 4. VISUALIZING GLL

Exploring and visualizing GLL parsing

```java
21 return SPPFNode (S, 0, \#1) has been created;
22 }
23 }
24 return SPPFNode (S, 0, \#1) has been created;
25 }
26
27 void create (GrammarSlot slot) {
28 Task t;
29 if there is no GSS node (slot, scanner.getPosition()) then create one {
30 if there is no GSS edge from (slot, scanner.getPosition()) to Cu then create one 
31 add edge to addedGSEdges
32 t = createTask("create", (slot, scanner.getPosition()), "todo", callStack.depth() - depth;
33 waitForResume();
34 t.setStatus("success")
35 for all (s, z) \in P with s = (slot, scanner.getPosition()) {
36 SPPFNode y = getNodeP(slot, Cu, z);
37 add(slot, Cu, z, getRightExtent(), y);
38 }
39 }
40 }
41 return (slot, scanner.getPosition());
42 }
43
44 void pop() {
45 Task t = createTask("pop", nil, "success", callStack.depth() - depth);
46 waitForResume();
47 if (Cu != \langle \bot, 0 \rangle) {
48 P.add(Cu, scanner.getPosition());
49 for each edge from Cu to v {
50 Let SPPFNode z be on the edge of (Cu, v)
51 Let Cu be of the form A ::= \alpha \cdot \beta
52 SPPFNode y = getNodeP(slot, Cu, z);
53 add(slot, v, scanner.getPosition(), y);
54 }
55 }
56 }
57 }
58 SymbolSPPFNode getNodeT(Terminal t) {
59 Int leftExtent = scanner.getPosition();
60 Int rightExtent = scanner.peek(t).length;
61 Task t = createTask("getnodet", t, "todo", callStack.depth() - depth);
62 waitForResume();
63 if (\neg S contains(SymbolSPPFNode(t, leftExtent, rightExtent))) {
64 create one
65 t.setStatus("success");
66 return SymbolSPPFNode(t, leftExtent, rightExtent);
67 }
68 }
69 SPPFNode getNodeP(Grammarslot slot, SPPFNode left, SPPFNode right) {
70 Task t = createTask("getnodep", CN, "todo", callStack.depth() - depth);
71 waitForResume();
72 Let slot be of the form A ::= \alpha \cdot \beta
73 if (#\alpha = 1 \land (head(\alpha) is a terminal or a non-nullable nonterminal) \land \beta \neq []) {
74 t.setStatus("success");
75 return right;
76 } else {
77 if (\beta \neq []) { t = A; } else { t = A ::= \alpha \cdot \beta }
78 if (left == $) {
79 Let i and j be the left and right extents of SPPF node right
80 if (\neg S contains((t, i, j)) create one
81 if (\neg S contains((t, i, j)) does not have a packed node (A ::= \alpha \cdot \beta, i) {
82 create one with child right
83 add(t, i, j) and packed node to addedSPPFNodes
84 add edge from (t, i, j) to packed node to addedSPPFEdges
85 add edge from packed node to right to addedSPPFEdges
86 }
87 }
88 ```
Let \( j \) and the right extent of SPPF node \( \text{right} \).
Let \( i \) and \( k \) be the left and right extents of SPPF node \( \text{left} \).

\begin{align*}
\text{if} (\neg S. \text{contains} ((t, i, j))) \{ \text{create one} \\
\text{if} ((t, i, j)) \text{ does not have a packed node} (A ::= \alpha \cdot \beta, k) \{ \\
\text{create one with left child} \text{ left} \text{ and right child} \text{ right} \\
\text{add} ((t, i, j)) \text{ and packed nodes to addedSPPFNodes} \\
\text{add edge from} (t,i,j) \text{ to packed node to addedSPPFEdges} \\
\text{add edge from packed node to right to addedSPPFEdges} \\
\text{add edge from packed node to left to addedSPPFEdges}
\}
\}
\}
\}
\}
\text{t.setStatus("success");}
\text{return} ((t,i,j));
\}
\]

\begin{align*}
\text{void} \text{ Goto(Grammarslot} s) \{ \\
\text{Task} t = \text{createTask("goto", s, "todo", callStack.\text{depth}() - depth);} \\
\text{waitForResume();} \\
\text{t.setStatus("success");}
\}
\]

\begin{align*}
\text{void} \text{ add(GrammarSlot} g, \text{ SPPFNode} s) \{ \\
\text{Descriptor} d = (g, C_u, \text{scanner.\text{getPosition}(), s}); \\
\text{Task} t = \text{createTask("add", d, "todo", callStack.\text{depth}() - depth);} \\
\text{waitForResume();} \\
\text{if} (\neg U. \text{contains}(d)) \{ \\
\text{R. add}(d); \\
\text{addedToR. add}(d); \\
\text{add edge from current\text{Descriptor} to} d \text{ to added\text{DescriptorGraphEdges}} \\
\text{t.setStatus("fail");} \\
\} \text{else} \{ \\
\text{removedFromR. add}(d); \\
\text{Duplicate\text{Descriptor} d'} = d. \text{copy}(); \\
\text{add edge from current\text{Descriptor} to} d' \text{ to added\text{DescriptorGraphEdges}} \\
\text{t.setStatus("success");}
\}
\}
\]

\begin{align*}
\text{protected} \text{ void} \text{ waitForResume()} \{ \\
\text{if} (\text{waitForDescriptor}) \{ \\
\text{if} (\text{current\text{Descriptor} == target\text{Descriptor}}) \{ \\
\text{waitFor\text{Descriptor} = false;} \\
\text{Thread.\text{suspend}();} \\
\} \text{else} \{ \\
\text{if} (\text{countDown ==} 0) \{ \\
\text{Thread.\text{suspend}();} \\
\} \text{else} \{ \\
\text{countDown--;} \\
\}
\}
\}
\]

\begin{align*}
@\text{Override} \\
\text{public} \text{ Set<Terminal>} \text{ test(Literal} l) \{ \\
\text{Task} t = \text{createTask("test", l, "todo", callStack.\text{depth}() - depth);} \\
\text{waitForResume() ;} \\
\text{Set<Terminal>} \text{ testSet = super.test(l);} \\
\text{boolean result} = \text{scanner.hasNext(testSet);} \\
\}
\]
CHAPTER 4. VISUALIZING GLL

Listing 4.3: Pseudocode Abstract parser after extending it with the storage of executed GLL tasks.

The `createTask()` function creates a GLL task after which it is added to the variable `performedTasks`. After a statement has been executed, the status of such task is modified using the `setStatus()` function. Note that the outcome of executing the `add()` and `test()` statements can either fail or succeed. In order to avoid adding any visualization code in the parser, the `test()` function has to be evaluated before it is executed.

As for the third algorithm, note that depending on the outcome of a statement, the control flow of a GLL parser can become different. In order to make visualization as stable as possible, the visualization of statements that in the end may not be executed by the parser have to be avoided.

The algorithm for merging the actual GLL task set with the precomputed control flow graphs is listed in Listing 4.4. Note that the algorithm shows much resemblance with the implementation of union operation in set and logic theory.

Listing 4.4: Algorithm for merging the actual GLL task set with the precomputed control flow graphs.

```java
Task actual; CFGNode precom;
List<Task> executed;
List<Task> result = [];
void createTaskSet(GrammarSlot g){
    if (executed.isEmpty()){
        result.add("Goto", g, "todo", 0);
    } else{
        repeat{
            actual = executed.removeFirst();
            Let c be the CFG for g:
            precom = c.root();
            result.add(actual.getTask(), actual.getValue(), actual.getStatus(), actual.getDepth());
            if (actual.getTask() == Goto){
                Let c be the CFG of actual.getValue()
                precom = c.root();
            } else{
                Let d be a CFG node such that precom → d and d = actual.getTask()
                precom = d;
            }
        } until (executed.isEmpty())
        boolean first = true;
        while (precom.hasChildren()){
            if (precom.getOutDegree() > 1){
                Let d be a CFG node such that precom false → d
            } else{
                Let d be a CFG node such that precom → d
            }
            if (first){
                result.add(precom.getTask(), parser.getValues(), "todo", 0);
            } else {
                first = false;
                result.add(precom.getTask(), null, "empty", 0);
            }
        }
        return result;
    }
}
```
CHAPTER 4. VISUALIZING GLL

Listing 4.4: Pseudocode for the synchronization of the actual task set versus the precomputed one.

Variables $actual$ and $precom$ initially represent the top elements of the actual GLL task set and the control flow graph of grammar slot $g$. The main idea of the algorithm is to relate every performed statement in $executed$ to a node in the control flow graphs so that in the end we know which statements in these graphs have been executed. Note that by definition of a CFG, the execution of a statement can be related back in the CFG by means of a transition. Once we have obtained the last position in the CFGs, we can generate the tasks that still have to be executed by traversing the parts of the CFGs that the parser has not considered yet. To ensure that we do not visualize tasks that may not have to be executed, whenever we encounter a $test()$ statement, the algorithm generates tasks as if the evaluation of this statement would lead to false. One small remark, note that we maintain control flow graphs per grammar slot. Hence, whenever a $Goto(g)$ statement is encountered, we have to make sure that the merge algorithm continues with the CFG corresponding to $g()$. The function $parser.getValues()$ assigns the parameters that were passed along with the current statement to the GLL task.

4.4.5 GSS

When visualizing the GLL tasks of the algorithm, for the illustration of functions $create()$ and $pop()$ we saw that a separate visualization of the GSS data structure can be desired. Not only does this view allows the system to directly visualize the variable $C_u$ in the algorithm, it also allows the user to experience the evolution and usage of the GSS data structure over time.

To show the resemblance between the GLL tree and the structure of the GSS, for the visualization of both trees, the GraphViz graph layout algorithm [10] has been used. As long as the visualization does not detect any sharing, the structure of both trees are in essence the same. When looking at the notion of grammar slots in a GSS node, we experienced that this data at first was difficult to relate back to the data that is stored on the call-stack of an LLRD parser. This is however mainly due to the fact that the traditional visualization of a call-stack as a list of nested function invocations does not make this return data explicit. In order to see the resemblance however between a GSS trace and an LLRD call-stack, we can visually augment every GSS node with the nonterminal where it was created. In a GLL parser one can obtain this data by looking at the grammar slot of a GSS node and take the nonterminal that is preceded by “·” symbol. Figure 4.36 illustrates the result of visualizing the GSS with this additional data.
Figure 4.36: We can represent the LLRD stack of a derivation by augmenting every GSS node with the nonterminal where it was created. This figure illustrates the resemblance between this information and the GLL tree in red.

4.4.6 SPPF

The interaction between functions getNodeT(), getNodeP() and the algorithm can be improved by showing what the impact is of executing these operations. One way to do so is to show which SPPF nodes and edges are created when executing these statements. The main problem with this approach however is that not every SPPF branch that is constructed by the parser ends up in the resulting SPPF structure. Since it is possible for a partial derivation to fail parsing the input string at some point, the SPPF branch that was constructed along the way becomes useless. Hence, depending on whether a derivation is going to be successful in the future, the constructed SPPF branch will be part of the final SPPF. Why is this a problem? Unfortunately, it turns out that even for small input strings and grammars (such as aad$ and Grammar Γ₅) the number of useless SPPF branches is as large as the number of SPPF nodes and edges in the final SPPF. Figure 4.37 shows a visualization of all SPPF nodes that are created by our example GLL parser and the input string aad$.
Not only do the useless branches significantly clutter the visualization, without the presence of the final SPPF, it is difficult to see what the part of the input string is currently considered. This is especially the case when parsing input strings containing repetitive patterns (e.g. $aaaaaaad$$S$). Besides, one can ask themselves whether the visualization of the useless branches can be fruitful for the understandability of the parser. Since the visualization of the SPPF as a whole is not an option, there was decided to only visualize the result of function getNodeT() and getNodeP() if the resulting SPPF branch ends up in the final SPPF. In order to see however when an SPPF branch is useful, knowledge about the final SPPF has to be known in advance. Figure 4.38 shows how we can use highlighting and the final SPPF to provide the user some global context about
the part of the SPPF that is currently constructed.

Figure 4.38: By means of highlighting techniques we can show the progress of the SPPF over time.

By means of two pointers $CN$ and $CR$, we can illustrate the left and right branch of the SPPF the algorithm is currently considering. We can improve the readability of the SPPF by visualizing the extents of an SPPF node as a region, representing the substring that is encapsulated by the extents. By emphasizing the literals in the alternate that are responsible for parsing this string, we can summarize the purpose of an SPPF branch by looking at the root node of that branch. To indicate that the packed nodes represent non-deterministic choices, there was decided to visualize the edge to a packed node differently. Figure 4.39 shows the resulting SPPF after applying this visualization scheme.

Figure 4.39: a) Traditional representation of an SPPF b) Alternative representation to ease readability.

Recall from Section 3.7.1 that a packed node represents a choice indicating how an alternate must
divide the parsing of the input string. The extent of a packed node, represents the position in the input string where this split occurs when constructing the SPPF. Note however that in the new representation this split is now also clearly reflected by the SPPF nodes surrounding a packed node. So, in order to avoid any redundancy in the visualization, there was decided to visualize a packed node as a split showing how an alternate is divided when parsing the remaining input string.

4.4.7 Popup messaging

A downside of providing an at runtime visualization of the GLL parser is that we never know how the next step of the algorithm is exactly going to look like. For the understanding of the algorithm however, it is important that at some point in time the user is able to predict what the next step is going to be. To assist the user in making these predictions, the visualization provides the user an informal description whenever something interesting is happening in the algorithm. In Figure 4.21 we already saw an example of how messaging by means of “thinking balloons” can be convenient to introduce phenomenon such as sharing in the visualization.

Several issues that one may encounter when implementing this messaging is that the occurrence of a popup not only depends on the statement the parser is currently considering, but also on the statements that have been performed in the past. For instance, depending on whether an add() statement is invoked by an alternate function or the create() statement can significantly change the explanation why this statement is performed in the first place. Tables 4.2 and 4.3 show the conditions and messages that are used throughout the GLLExplorer system to increase understandability of the parser. The first table show the messages that are generated when encountering a specific GLL task. The second table show the messages after a certain GLL task has been executed. The variables in the messages depend on the descriptor and parameters that the parser is currently using. Note that is possible for multiple messages to occur at the same time.

<table>
<thead>
<tr>
<th>currentTask</th>
<th>Condition</th>
<th>Message</th>
<th>Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>add()</td>
<td>invoked by a test()</td>
<td>This path seems right: ( a \in test(A) ). Lets create a descriptor to take this route in the future.</td>
<td>GLL viewer</td>
</tr>
<tr>
<td>pop()</td>
<td>none</td>
<td>I am done with nonterminal ( A ). Thanks to the GSS I know that I should continue exploring ( B := \alpha \cdot C \beta ). String ( E ) remains to be parsed.</td>
<td>GLL viewer</td>
</tr>
<tr>
<td>create()</td>
<td>true</td>
<td>Ready to apply ( A() )</td>
<td>GSS viewer</td>
</tr>
<tr>
<td>create()</td>
<td>true</td>
<td>Hmmm... ( A ) is a nonterminal and could match anything. So I have to explore the possibilities for ( A ) first. After finishing ( A ), I must remember to create a descriptor which leads me back to ( S := \alpha A \cdot \beta ). Let's store this in the GSS.</td>
<td>GLL viewer</td>
</tr>
<tr>
<td>test((A))</td>
<td>number of alternates for ( A ) larger than one</td>
<td>There seems to be multiple paths</td>
<td>GLL viewer</td>
</tr>
<tr>
<td>test()</td>
<td>previous statement is a Goto()</td>
<td>Let see whether parsing ( S ) is fruitful</td>
<td>GLL viewer</td>
</tr>
</tbody>
</table>
CHAPTER 4. VISUALIZING GLL

<table>
<thead>
<tr>
<th>add()</th>
<th>invoked by a create() statement</th>
<th>I have seen this path before</th>
<th>GLL viewer</th>
</tr>
</thead>
<tbody>
<tr>
<td>getNodeP()</td>
<td>alternate consists of one literal</td>
<td>Alternate α consists of one literal, lets first create a nonterminal node for this alternate before we can attach it to the SPPF</td>
<td>GLL viewer</td>
</tr>
</tbody>
</table>

Table 4.2: Conditions that are checked for the realization of popup messaging in the GLLExplorer system.

<table>
<thead>
<tr>
<th>previousTask</th>
<th>Condition</th>
<th>Message</th>
<th>Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>add()</td>
<td>descriptor already exists</td>
<td>I have already processed this descriptor before!</td>
<td>GLL viewer</td>
</tr>
<tr>
<td>create()</td>
<td>current task is add()</td>
<td>Apparently, I have walked this path before. Let's continue with $A_0()$ next time.</td>
<td>GLL viewer</td>
</tr>
</tbody>
</table>

Table 4.3: Popup messages after a certain GLL task has been executed.

A second difficulty that one may encounter is that when designing this messaging is that the size of a popup depends on the data that currently processed by the GLL algorithm. Hence, when designing the visualization, one must take into account that the size of these popup messages can vary significantly. To avoid the popup messages from cluttering the visualization, in the GLLExplorer system, messages can be moved and hidden whenever necessary.

4.4.8 Context-highlighting

As illustrated in Section 4.3, in case of ambiguities visualization techniques have to be applied to distinguish the progress of one derivation from the other. The algorithm for context-highlighting consists of two parts. The first part highlights the GLL nodes in the visualization according to the derivations that are interested in the descriptor. The second part of the algorithm is responsible for assigning the right status to the block arrows and input string. For the realization of both parts we can reuse the concept of a descriptor graph to perform this highlighting. Recall from Section 4.4.1 that every path from root to non-duplicate leaf node corresponds to a derivation.

The algorithm as proposed in Listing 4.5 illustrates the main idea to highlight the GLL nodes in the GLL tree corresponding to one path in the descriptor graph. If it turns out that the algorithm detects sharing at the occurrence of a create() statement, every derivations that participates in this sharing can be highlighted by repeatedly invoking Listing 4.5. By reporting every path in a descriptor graph in a listbox, the user can decide for himself which highlighting it is interested in.

```c
void highlight(GLLNode root, Descriptor leaf, DescriptorGraph graph){
    GLLNode currentRoot = root;
    Descriptor currentDescriptor = graph.getRoot();

    while (currentDescriptor != leaf){
        if (currentDescriptor.getGSSNode() ∈ gsst)
            highlight e with fat box and GLL block corresponding to d.getGrammarslot()
        else
            highlight root with fat box and GLL block.
            set input string of root to currentDescriptor.getGSSNode().getInputPosition().
            with progress until leaf.getInputposition();
    }
    let d be the next descriptor that is on path from currentDescriptor to leaf.
    let e be the GLL node that was created by d with parent currentRoot.
    let gsst be the GSS trace from graph.getRoot().getGSSNode() to d.getGSSNode()
    Highlight root with fat box and GLL block.
}
```

98 Exploring and visualizing GLL parsing
15 }  
16 Set edge from currentRoot to e solid  
17 Set input string of e to d.getGSSNode().getInputPosition()  
18 with progress until leaf.getInputPosition();  
19 
20 if(leaf $\notin R \land$ leaf.getInputPosition() $\neq \#I$){  
21 Color all GLL blocks of e red.  
22 } else{  
23 Color GLL blocks upto d.getGrammarslot() of e green.  
24 }  
25 currentDescriptor = d;  
26 
28 i f ( sharing ){  
29 Let duplicates be the set of all duplicated descriptors corresponding to leaf  
30 for every duplicate in duplicates{  
31 for every path in graph from graph.getRoot() to duplicate.getParent(){  
32 highlight(root, duplicate, graph);  
33 }  
34 }  
35 }  
36 

Listing 4.5: Pseudocode for highlighting derivations.

The algorithm initially starts at the root of the descriptor graph. For every descriptor in the path from root to leaf, the GLL node corresponding to this transition is highlighted. If at some point the current descriptor of the algorithm detects sharing, all derivations that are involved in this sharing have to be visualized as well. For this we can use the locations of the duplicate descriptors in the descriptor graph. Since sharing is detected by the descriptor who is about to create the duplicate descriptor, line 31 looks duplicate.getParent() rather than duplicate.

To represent the stack of a derivation, a GLL node and GLL block are emphasize with a fat border whenever it is part of that stack. We can use the one-to-one correspondence between a GLL trace and a GSS trace to discover which GLL nodes have to be emphasized. The status of a GLL node can be adjusted depending on the grammar slots that occur along the path. The color of the GLL blocks depend on whether leaf still has to be processed. If variable leaf for instance has been processed but did not managed to parse until the end of the string I, then we know that GLL blocks of the GLL nodes along this path have to be colored red.

The main difficulty of the algorithm is that a descriptor can correspond to more than one GLL node. So in order to know which GLL node must be chosen in this traversal, a variable currentRoot has to be maintained storing the last GLL node that has been considered. Note that the algorithm for colouring the status of GLL nodes that are not highlighted is very similar to the code in Listing 4.5 without the highlighting part. By applying this algorithm for every path in the descriptor graph, every GLL node will obtain a status. There are of course several optimizations possible to make this procedure efficient. The ordering in which the descriptor paths are coloured depends on the desired strategy (e.g. LIFO or MostComplete).

4.4.9 Left-Recursion

As shown earlier, the algorithms for the generation of the GLL tree and context-highlighting are only applicable in case of non left-recursive grammars. The main reason why this is case is that for the visualization of left-recursion, there are two fundamental issues with the proposed visualization. The first problem of visualizing left-recursion at runtime is that it in general is impossible to know beforehand how many times left-recursion has to be unfolded in the visualization in order to parse the input string successfully. This is because at runtime, the GLL parser is still trying to discover how many times left-recursion has to be applied. When looking a this problem from the
viewpoint of the descriptor graph or the GSS, we can see that the algorithm introduces cycles whenever left-recursion is detected. So unless we know how the resulting SPPF is going to look like, we are in general unable to construct a visualization a GLL tree that covers the desired level of left-recursion. This is also illustrated in Figure 4.40.

$$E ::= E + 1 | 1 \quad (\Gamma_9)$$

Figure 4.40: Unless we know the resulting SPPF, in general, we do not know how many time the GLL tree has to be unfolded.

To illustrate the second problem of visualizing left-recursion, let use consider Grammar $\Gamma_9$ and the input string $1 + 1$. Let us assume for a moment that with the use of future knowledge we know how many times the visualization has to be unfolded. After the parser has discovered that parsing $E$ as $E_1$ is not sufficient to recognizer the entire input string, it will continue processing the descriptor for $E_0$. As soon as the parser reaches the descriptor $(E_1, \langle E ::= E \cdot + 1, 1+1$ $\rangle, 1+1$ $)$, it will notice that the application of $E$ as $E_1$ is left-recursive. This is also illustrated in Figure 4.41.

Figure 4.41: Processing of descriptor $(E_0, \langle E ::= E \cdot + 1, 1+1$ $\rangle, 1+1$ $)$. 

100 Exploring and visualizing GLL parsing
Similarly as in Section 4.2.3 the parser now creates a descriptor \((E ::= E \cdot + 1, (E ::= E \cdot + 1, 1+1\$), 1+1\$)\) to capture this sharing. When processing this descriptor however, we can see that the progress of derivations are going to interfere with each other. This is also illustrated in Figure 4.42.

Figure 4.42: Processing of descriptor \((E_1, (E ::= E \cdot + 1, 1+1\$), 1+1\$)\). Note that the status of the GLL nodes differ from each other.

Be aware that due to the self-loop in the GSS, there are officially infinitely many derivations that are interested in this descriptor. If we only look at two of them, then we can see that the yellow GLL nodes represent the parse work that both derivations have in common. Note that from the viewpoint of a descriptor, the execution of the \texttt{Scan()} operation is successful, since in both derivations, the input string corresponds to “$+1$”. Note however that if we look at the visualization from the viewpoint of a derivation, performing this operation should only holds for one of the two yellow GLL nodes. This is again an example of the descriptors paradox, where the at runtime visualization of the progress of a GLL parser interferes with the notion of a descriptor.
4.4.10 Summary

In chapter 4 we have discussed a concept of grammar walking to visualize the execution and progress of a GLL parser. By representing the progress of a derivation as a route in the resulting parse forest, we can summarize the progress of a GLL parser as a collection of routes that have been walked. The fact that a descriptor can be relevant for more than one derivation, can be modelled as walking these routes in parallel. To give the user an overview of the steps that still need to be performed, the GLLExplorer system visualizes the GLL tasks that still have to be performed. One of the main difficulties of constructing such task set is that at runtime the visualization somehow has to know which statements have to be executed. Luckily we can obtain this information by looking beforehand at the structure of the generated parser code.

Because of the fact that in general the execution of a GLL parser requires a large number of GLL tasks before it can return an SPPF, simply running the algorithm step-by-step can become time-consuming. Especially if the user is only interested in specific parts of execution (e.g. sharing), running the entire algorithm step-by-step is not an option. The fact that we are dealing with a multi-view system also makes it difficult to spot every change in the application when multiple views change in parallel. In order to avoid having to restart the entire visualization once such change is overlooked, the GLLExplorer system maintains history to revert the algorithm to an earlier state. Unfortunately, storing all data structures as a whole after the execution of a GLL task already becomes impractical for small grammars and input strings. To solve this problem, we introduced some shadow administration for every data structure to keep track how it evolves over time.

Section 4.3 already argued that the visualization of every derivation in progress as a separate tree will significantly increase the size of the application. Hence, in order to make the visualization practical, derivations are summarized into one unified tree structure. This sharing however comes with the price that in case of ambiguities context-highlighting has to be applied to distinguish between one or more derivations. It even turned out that in order to obtain such tree structure, data structures such as GSS and SPPF may contain insufficient information.

As mentioned before, the main difficulty of the visualization does not lie in its concept, but in the realization of it. One important property of visualizing data in general is that the visualization is capable in explaining all situations (both normal and special) in an algorithm consistently. In order to show how this consistency is achieved in the visualization, in Section 4.4.4 we have defined the construction of the visualization as a set of rules that must be applied after a certain step has been executed. One advantage of this approach is that, similarly as in a model transformation, one can construct a visualization of a GLL parser by starting with an initial figure and incrementally applying the set of rules. A downside of this approach however is that the simplicity of the concept becomes blurred because of the detailed level in which the rules are specified.

The main issue with the explanation of the GLL visualization is that the quality of the proposed concept can only be measured by looking at the tiniest details in the execution of a GLL parser. This is mainly because the set of conditions that have to apply in order to obtain for instance sharing is quite large and specific. So in order to warn the user when something special is happening in the algorithm, we somehow have to show what makes this set of conditions at a certain point so special. The fact that this sharing phenomenon is so special also reflected in the execution of our example GLL parser. Even for the input string `aad$` it requires about 149 steps (out of approximately 155) before it actually detects any sharing. It is however this kind of situations where the visualization has to show what makes the concept so intuitive.
Chapter 5

Future work & Conclusions

In this thesis, we have looked at different viewpoints to explore (the foundations behind) Generalized LL parsing. The main goal of this project was to discuss and visualize the working of a generalized LL parser in order to reduce the learning curve of the algorithm. In the first viewpoint as described in Chapter 3, we have investigated how we can obtain a GLL parser starting from an LLRD recognizer. Although the concept behind the transformations are relatively easy, the introduction of an explicit call-stack and for instance support for left-recursion causes the necessary problems. For the realization of the visualization, we have seen that we can use the concept of “walking” to explain how the GLL algorithm performs any parsing through the use of descriptors. By means of context-highlighting and colouring, we can summarize the progress of the relevant derivations without having to create a separate visualization for every derivation separately. Unfortunately, due to insufficient knowledge about the derivations that a the parser may discover in the future, the proposed at-runtime visualization is unable to visualize the notion of a descriptor consistently. The descriptors paradox illustrated that visualizing the exploratory character of the GLL parser on one hand and explaining the notion of a stateless descriptor on the other hand do not mix well in an at-runtime visualization. The lack of future knowledge is also the main reason why left-recursion cannot be visualized at all without having some post-parse knowledge about the resulting parse forest.

5.0.11 Contribution to research questions

How can we obtain a GLL parser by starting from a traditional left-recursive descent parser?

Chapter 3 illustrated that in order to overcome the limitations of a traditional LLRD parser, the return locations of the parser had to be maintained and controlled outside the scope of the call-stack. For programming languages that do not support the use of goto statements, this required us to decompose the parser code into smaller fragments such that code jumping can be realized through function invocations. For the support of multiple derivations, the LLRD parser were extended with multiple stacks by introducing the notion of a descriptor. Since the number of stacks can become exponential for certain context-free grammars, stack are stored together in a GSS. Left-recursion can be modelled in the GSS by means of cycles. Since a GLL parser can produce multiple parse trees for the same input string and the number of parse trees can become infinitely large, an SPPF datastructure had to be introduced to capture these trees finitely. Based on these transformations we can see that a GLL parser in essence represents a collection of LLRD parsers together and uses centralized data structures to model the results of all these parsers compactly. We also saw that for context-free grammars that are not cyclic nor contain left-recursion the introduction of a GSS is not necessarily required.
Can we explain the control-flow of a GLL parser by means of visualization techniques? Chapter 4 described how we can use the notion of grammar walking to illustrate the working of GLL parsing for non left-recursive grammars. The main idea of the visualization was to show the progress of a GLL parser as if it were a collection of LLRD parsers. Although the proposed concept provides a nice overview why and when the steps in the GLL parser are executed, the fact that the visualization is created at run-time makes it hard to consistently visualize the concept of a descriptor. The paradox that was discovered when designing the visualization illustrated that the visualization of descriptors and the progress of the parser in general do not mix well. Whether the use of post-parse data will entirely solve this paradox is still an open question. Note that although the concept works well for small input strings and grammars, visualizing ambiguities by means of context-highlighting can however become problematic for large and highly ambiguous grammars. It even turned out that the visualization of left-recursive grammars at runtime in general is impossible to construct a proper visualization without knowing the outcome of the parser upfront.

5.1 Future work

The first part of the thesis explains the purpose and main idea behind the introduction of all data structures and functions in a GLL parser. The visualization of the GLL algorithm on the other hand still requires the necessary research in order to make it practical for larger applications. We can identify the following two main area in which future work is possible:

5.1.1 GLLExplorer

Due to the difficulties that were discovered along the way with respect to generating the visualization, the demo application as provided still requires the necessary work to become of practical use. Especially with respect to the interaction between views and realization of left-recursion, the application still requires some additional work and research to do this properly. The goal of Chapter 4 was not only to introduce the concept, but also to illustrate the difficulties that one may encounter when realizing such visualization. Although the main concepts are implemented, with respect to the visualization side of the application there are still many things to be improved. The concept as provided. In Chapter 4.2 illustrates the working of a GLL parser for small input grammars and small input strings. One can imagine however that even with the use of context-highlighting, for practical grammars, the size of the visualization becomes too large. One idea would be to look at different ways of hiding certain branches or to summarize parts of the derivations into smaller units as depicted in Figure 5.1. The realization of these compaction methods with respect to multiple derivations, context-highlighting, and user-interaction still may cause the necessary issues to be solved.

5.1.2 Post-parse visualization

As mentioned earlier, the visualization as described in Chapter 4 is designed based on the state of the algorithm at run-time. The descriptors paradox however made us realize that we cannot visualize both the control-flow and the concept of a descriptor by only using local data. Especially for the visualization of descriptors and left-recursion, knowing all derivations in advance can significantly change the way in which the execution of a GLL parser can be explained. The main difficulty however is how to use and combine this future knowledge into a practical visualization that is both suitable for the explanation of a descriptor and control-flow of the algorithm.
Figure 5.1: Summarizing the parts that have already been parsed can significantly reduce the size of the GLL tree.
CHAPTER 5. FUTURE WORK & CONCLUSIONS

5.2 Related Work

Visualizing parsing algorithms have been studied in many various settings. The first and probably most common way of visualizing parsing algorithms is by providing a post-parse visualization of the data structures that were used during parsing. Well known parser generation tools such as ANTLR [11] and Bison [3] use this type of visualization to aid programmers with debugging their parser specification. These applications typically have the ability to export their data structures to visualization APIs such as the [10] API [10]. Although such type of visualizations are nice for debugging purposes, the execution of the parser along the way still remains unexplained.

There are several tools known in the literature that try to visualize the execution of a parser at runtime. Visual Yacc [4], GYacc [9], CUPV visualization [2], LRParse [14], are examples of tooling that were specifically designed to help students understanding (Yacc’s) LALR and LR parsing respectively. Visual Yacc explains the execution of the parser by showing the effect of an LALR reduction in the stack and parse tree the algorithm has constructed so far. The application however does not provide the user any detail of the shift reductions that the parser still has to perform. CUPV improves on this application by providing a multi-view system where both the progress of the abstract syntax tree as well as the reduction steps are visualized in parallel. Together with GYacc they are both examples of applications that are separate from the generated parser. Examples of visualization tools for LL parsing are JFLAP [13] and LLParse [14]. Most of these applications however specifically only target LL(1) parsing and therefore try to visualize the execution of the algorithm by means of incremental parse table transformations. The LLParse application provides a text-based explanation of an LL(1) parser already showing how the use of informal descriptions can be valuable for the explanation of a parser.

In contrast the visualization techniques as described in 4, the previous tooling is mostly based on the concept of parse tables and LR parsing, whose concept significantly differs from (G)LL parsing. Because of the that LALR and LR parsing do not support ambiguous grammars, all these applications do not have to cope with the visualization of multiple parse trees and stacks. Since the ordering in which an (LA)LR parser performs it steps is deterministic, none of the applications have to cope with reversion techniques or parallelism to illustrate how a GLL parser deals with stack sharing. One thing that all these visualizations have in common is that the visualization of parse trees, stacks, and abstract syntax trees are all based on node-link diagrams. Many techniques as described in the literature focus on the concept one stack and one parse tree to visualize the main concept behind a parser. Techniques to explain the execution of a generalized parser however are yet still unexplored in the literature. Probably the closest research that has been performed in this area of research is the visualization of data structures for the parsing probabilistic context-free grammars. Because of the non-deterministic behaviour of the probabilistic grammars, they had to introduce the notion of a hyper graph to visualize all choices that were made by the underlying decoder. The GLLEXplorer system tries to overcome the scalability issues of such hyper graph by summarizing overlap between multiple derivations in one tree structure. Although the current literature has explained the generation and concept of a GLL parser extensively, the problems that a generalized LL parser has to cope with in contrast to a traditional LL parser have not been explored.
Bibliography


