MASTER

On-parse disambiguation in generalized LL parsing using attributes

Mengerink, J.G.M.

Award date:
2014

Department of Mathematics and Computer Science
Model Driven Software Engineering Group

On-parse Disambiguation in Generalized LL Parsing Using Attributes

Master Thesis

Josh Mengerink

Preliminary Version [1]

Supervisor:
Mark van den Brand
Committee Members:
Mark van den Brand
Tim Willemse
Adrian Johnstone
Elizabeth Scott

Eindhoven, August 2014

[1]Due to time constraints, some textual errors may still occur in the text. These will be fixed in a future version.

“Thanks to my solid academic training, today I can write hundreds of words on virtually any topic without possessing a shred of information, which is how I got a good job in journalism.”

Dave Barry

EINDHOVEN UNIVERSITY OF TECHNOLOGY

Abstract

Faculty of Mathematics and Computer Science
Model Driven Software Engineering Department

Master of Science

Parse-time Disambiguation in Generalized LL Parsers Using Attributes

by Josh Mengerink

When developing a new Domain Specific Language (DSL), writing a parser for your new language is either too dull, or too difficult. Parser generation based on grammars greatly improves the ease and speed with which parsers can be created. However, most parser generators suffer from deficiencies such as only supporting regular languages or grammars that are left-recursion free. Generalized LL parsing solves these issues, as the entire class of context-free grammars are supported. However, as we are dealing with generalized parsing, ambiguous grammars are also supported an GLL parsers will yield every correct derivation. Code generators on the other hand, usually require a single derivation to work on. We thus see the need to be able to reduce the amount of yielded derivations (i.e. disambiguate). Many different techniques for disambiguation exist, but only few (such as grammar rewriting) focus on incorporating these techniques in the generated parser. The downside to many of these techniques is that as a result of their work, the yielded derivations are often (slightly) different than the original derivation that was intended. In this thesis, we will attempt to extends grammars with an attribute system, such that the original structure of the grammar (and thus the derivation) is preserved, but we are intuitively able to define which derivations should be excluded from our final result.

Acknowledgements

This thesis is the result of my graduation project at the Eindhoven University of Technology. Before starting my graduation, I already dove into the subject of GLL during an honors project together with Bram Cappers and Bram van der Sanden [1]. This project was very energizing and fueled my already present intensest in semantic analysis, which I hope to further during my PhD.
I would like to thank Mark van den Brand for supervising this project, in addition to my committee members Adrian Johnston, Elizabeth Scott, and Tim Willemse.
During my time at the university, most of which was enjoyable, I have made some great new friends. In particular, I would like to thank Bram Cappers and Bram van der Sanden for the moments (both fun and aggravating) that we have shared.
Furthermore I would like to thank my house-mates for their infinite stream of distraction in whichever form (be it mental or liquid), without whom this thesis would have been finished a lot sooner.
Last, but certainly not least, I would like to thank my parents for their never ending support and inspiration, which I could not imagine myself living without.

Those that know me, know I am a great fan of putting inspiring quotes into my work, and I would like to start this thesis with one of my own:

> "Assumptions are the worst enemy of a computer scientist... But they are great for performance!"

Contents

Chapter 1

Introduction

In this thesis, the feasibility of extending Generalized LL (GLL) [2] with support for attributes will be investigated. The main goal of this extension is to use an attributing system to provide context for the disambiguation of languages that have ambiguous expression structures, or use the offside-rule. The primary type of ambiguities that we will aim to resolve are related to structures, rather than large-scale semantic disambiguation (such as type checking). Moreover, this disambiguation is to take place at parse-time as much as possible. This form of on-parse time disambiguation should serve as a proof of concept that it is possible for attributes to provide contextual information inside the GLL algorithm to serve as a disambiguation mechanism. We will start with a general introduction to parsing (Chapter 1), followed by a thorough explanation of the GLL algorithm (Chapter 2). Furthermore, the core concepts of the attribute system are explained (Chapter 3) and a step-by-step explanation is given on how to extend the data-structures in GLL to support these attributes (Chapter 4). We will wrap up with a case study that incorporates both expression-grammar and offside elements (Chapter 5).

1.1 Parsing

The first question that one might ask is: what is parsing? Parsing is the action of turning a sequence of characters into a tree structure that represents the actual meaning of that sequence. To illustrate the need for parsing, take the example sequence "`2+5*3`". Under normal arithmetical priority rules, this means: add two to the result of multiplying five and three. However, someone who is not familiar with these rules might understand this sequence as: add two to five, and multiply the result by three. Depending on which operation we do first (addition or multiplication), we end up with one of two derivations, each with a different outcome (17 and 21 respectively). Thus, in order for someone (or something) to understand the actual meaning of "`2+5*3`", we need additional information (namely that we should multiply before doing addition). We encode this information in a tree structure, also called a *parse tree*. Both possible parse

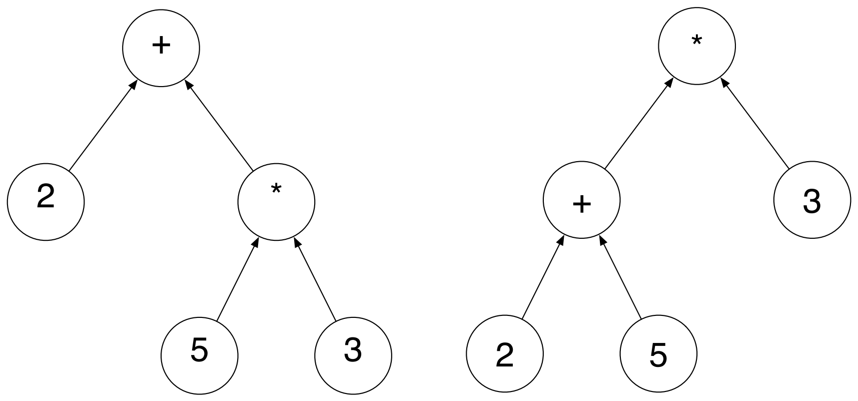


FIGURE 1.1: A figure showing the two possible parse trees for the sequence "2+5*3"

trees for the sequence "2+5⋆3" are presented in Figure 1.1. However, we would say the left one is correct.

Summarizing, the parser adds structure (e.g. perform multiplication before addition) to the input sequence in the form of a tree structure. This process of transforming an input sequence into a tree structure, is called *parsing* and is performed by a so called *parser*. A parser may also conclude that no tree structure exists for a given input sequence, in which case we say that an input is invalid and a descent error (regarding the reason why there is no tree) should be generated.

When performing these actions, the parser is often assisted by a scanner. A scanner is a component that transforms the input into a sequence of tokens, based on lexical patterns. Rather than performing its operations on segments of the input, the parser can work on tokens (abstracting from the actual characters in the input). This enables parsers to be more efficient and compact. Scanners may also perform tasks such as removal of (unnecessary) layout from the input sequence, to simplify the parser further.

Traditionally, parsers would be written by hand, which was a very cumbersome and mechanical effort. Due to the mechanical nature of this task, research was done into the automation of the process. Currently, most parsers are generated based on *context-free grammars*. These grammars specify the shape of any input sequence that is valid. In the structure of a grammar, one can immediately encode the additional information (such as priority) the parser should imply on an input sequence. Grammars will be discussed in depth in Section 1.2.

1.2 Grammars

A grammar is a set of rewrite rules, with which we may rewrite a sequence of symbols to another sequence of symbols. To illustrate, assume we want to parse all, non-empty, sequences of the character "a" followed by exactly that many occurrences of the character "b". Take sequence σ_{ex} = `aaabbb` as an example. A grammar for this can be seen in Grammar Γ_1. We see that this grammar has two rewrite rules:

1. We may substitute the symbol `S` for the sequence of symbols `aSb`.
2. We may substitute the symbol `S` for the empty sequence ε. This, in essence, means we may eliminate `S` from our sequence at any given point.

If we can rewrite the initial symbol `S` of a grammar to the input sequence, we say that the input sequence σ_{ex} is valid with respect to the grammar. In our example, indeed "`aaabbb`" is valid with respect to the input, as (using the first rule) we may rewrite `S` to "`aSb`", "`aaSbb`", "`aaaSbbb`" to finally eliminate `S` using the second rule to remain with the sequence "`aaabbb`". Actual parsers rewrite their symbols in a more structured way. We distinguish two main kinds: LL, which always chooses the left-most symbol, and LR, which always chooses the right-most symbol [3].

$$(\Gamma_1) \qquad \begin{aligned} \mathtt{S} &::= \mathtt{a} \quad \mathtt{S} \quad \mathtt{b};\\ \mathtt{S} &::= \varepsilon; \end{aligned}$$

As this is all quite informal, let us go into some more details. Formally, a grammar Γ is a tuple $\langle N, T, S, P\rangle$ where:

- N is a finite set of non-terminal symbols.
- T is a finite set of terminal symbols, including the empty symbol (denoted ε). Throughout this paper we will assume all terminal symbols to be regular expressions. However, we allow for the shorthand "t"($t \in T$), which is to be interpreted as the actual sequence of characters (e.g. "+" = [+], "test" = "[t][e][s][t]").
- $S \in N$ is the start non-terminal symbol.
- P is a finite set of productions.

Let $\mathcal{L}(X)$ denote a list (of an unspecified size) whose elements are from the set X. A production is of the shape $\alpha ::= \beta$, where $\alpha, \beta \in \mathcal{L}(N \cup T)$. However, we will limit ourselves to productions where $\alpha \in N$ (but still $\beta \in \mathcal{L}(N \cup T)$). Note that since P is a set, we ignore duplicate productions.

We say an input sequence σ (of terminal symbols) is valid with respect to a grammar Γ if and only if there exists a sequence of substitutions such that:

- We start with the sequence $\sigma' = \mathtt{S}$;
- We repeatedly choose the **left-most** non-terminal symbol $n \in N$ that occurs in σ'. Note that this is the case for parsers yielding a left-most derivations. Parsers yielding a right-most derivations choose their **right-most** non-terminal symbol for rewriting [3].
- We choose an $\alpha ::= \beta \in P$ such that $\alpha = n$;

- We substitute n in σ' by the sequence of literals β (where a literal is an element in $N \cup T$);
- When there are no remaining non-terminal symbols in σ', then it must hold that $\sigma = \sigma'$.

Please consider the example grammar Γ_2. This grammar has non-terminal symbols $N = \{S, B\}$, terminal symbols $T = \{a, b\}$ and start non-terminal `S`. Moreover, there are two productions: `S ::= a B` and `B ::= b`. To demonstrate an input sequence that is accepted by this grammar, consider "`ab`". This is a valid sequence, as a substitution exists that meets our specified restrictions as can be seen in Listing 1.1.

$\sigma' = S$
$\{S ::= aB\}$
$\sigma' = aB$
$\{B ::= b\}$
$\sigma' = ab = \sigma$

Listing 1.1: A simple grammar derivation

$$(\Gamma_2) \qquad \begin{aligned} \texttt{S} &::= \texttt{a} \quad \texttt{B}; \\ \texttt{B} &::= \texttt{b}; \end{aligned}$$

As was stated before, a grammar specifies additional information to be imposed on an input sequence (e.g. multiplication must be applied before addition). But this might not be completely obvious from the example presented. Take the (slightly) more sophisticated example of grammar Γ_3.

$$(\Gamma_3) \qquad \begin{aligned} \texttt{ASSIGNMENT} &::= \texttt{VARIABLE} \quad \text{“ := ”} \quad \texttt{NUMBER}; \\ \texttt{VARIABLE} &::= \text{“x”} \quad | \quad \text{“y”}; \\ \texttt{NUMBER} &::= \text{“1”} \quad | \quad \text{“2”}; \end{aligned}$$

If we parse the sequence "`x:=2`", observe the parse tree for this sequence presented in Figure 1.2. We see that we can substitute the start non-terminal `ASSIGNMENT` to "`VARIABLE ':=' NUMBER`" in which we can substitute `VARIABLE` for "x" and "`number`" for `2`. From the resulting tree (in Figure 1.2) we can derive that we should interpret the sequence "`x:=2`" as the assignment of the number "2" to variable "x".

1.3 Grammar classes

There are a number of different classes of grammars each having a different level of expressiveness [4]. That is, each incrementally greater class of grammars is able to specify an increasingly greater class of languages. For example, if we want to be able to

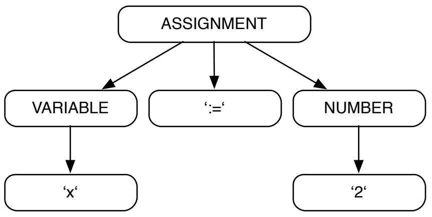


Figure 1.2: A parse tree for sequence “x:=2” with respect to Γ_3

specify all sequences $\{a^n b^n c^n \mid n > 1\}$ in a grammar, we cannot do so using a context-free grammar. However, a context-sensitive language is able to specify such a sequence. The two classes that we will consider are context-free and context-sensitive grammars.

The difference between a context-free and a context-sensitive grammar, is that in a context-free grammar we are always allowed to substitute a non-terminal symbol using any production with that non-terminal as its left-hand-side (LHS) [5]. These context-free grammars generate the class of context-free languages.

However, in context-sensitive grammars there may be additional restrictions on the applicability of a production than just the presence of a non-terminal. For example, a production in a context-sensitive grammar may be of the form `abX` ::= `c`. This means that we may only substitute X for c, if the two symbols preceding X are terminals a and b. Context-sensitive grammars generate the class of context-sensitive languages.

Another way of modeling context-sensitive productions is: $\alpha \times \omega ::= \beta$, where ω is a particular state of an environment. One may then only substitute non-terminal α for sequence β if and only if the environment matches ω. Attributes are one of mechanisms used to specify and manipulate these environments.

1.4 Attribute Grammars

A well known mechanism that relies on the general idea of attribute annotation of non-terminals is Attribute Grammar (AG) [6, 7]. Attribute grammars form an extension of the context-free grammars and have historically been proven useful for specifying context-sensitive syntax.

Besides attributes, computational rules on these attributes can be specified. These rules describes how to evaluate the values of certain attributes with respect to other attributes. Furthermore, an evaluation method is required to compute the actual values of the attributes in an efficient way. Whenever calculated values would (post-parse) conflict a set of specified rules, this would result in an error message. Is a sense, such an error message can be seen as rejecting the derived parse-tree based on semantic grounds. We aim to use this last concept in the context of a parse-forest, to reject all undesired trees based on their attributes. Based on the evaluation method of these attributes, we can partition the class of attribute grammars into four sub-classes:

- L-Attribute grammars [8], in which only depth-first left-to-right attribute evaluation is supported .
- S-Attribute grammars [8], in which only bottom-up attribute evaluation is supported.
- Absolutely non-circular attribute grammars [9], which (in essence) allows for any attribute-computation as long as their order of computation is non-cyclic. That is, an attribute-computation is cyclic if the value of an attribute (say `A`) depends (recursively) on its own value.
- Ordered Attribute Grammars [10], that require particular partial orders to hold on the attribute evaluation process.

Checking for cyclic attribute dependencies in an efficient way has been a challenge ever since AGs were envisioned by Knuth and has lead to many publications by Alblas and Kastens (among others) [11–14].

Classically , AGs were mainly used in deterministic parsers to annotate parse-trees [15–17]. Post-parse, such an annotated tree could then be checked for errors with respect to the specified attribute restriction. More modern work on attribute grammars focuses on using attribute grammars to generate parse-forest disambiguation filters [18] (i.e. post-parse disambiguation). Others work on extending specific languages with attribute grammars for increased extensibility [19], or extended LR parser generators [20].

To illustrate the basic concept of an attribute grammar, consider the context-free expression grammar Γ_4. Note that it is ambiguous, hence we assume that our parser returns all possible derivations. We extend the grammar with attributes, to represent parent-child relationships (the attribute "parent") and enforce that a multiplication (which has higher priority than addition) must always occur before addition (i.e. occur lower in the parse-tree). We do so by means of the semantic rule `parent != *` on the production for addition. Furthermore, we specify that recursive non-terminals should have their parent property set to the operator of the current production (e.g. `[parent=+]` for the addition-production). All this is presented in attributed grammar Γ_5.

Please note that this grammar is ambiguous, so there are multiple correct derivations. However, as non-general parsers only return a single derivation, the implementation of the parser regulates which derivation is yielded. For the sake of simplicity we will assume our parser to be deterministic. Say now we parse input sequence "`1+2*3`" with respect to Γ_5. Our parser yields the annotated parse-tree presented in Figure 1.3. We can then evaluate all nodes and their contextual restrictions. In particular, we see that there is an instance of the production `Exp::=Exp "+" Exp` being used. This instance is annotated with `parent="*"`, however, the LHS of the relevant production states that the parent property should be unequal to `"*"`. Hence, there is a violation in our parse-tree, and we can disqualify it.

(Γ_4) $$\texttt{Exp} ::= \texttt{Exp} \quad \text{“}+\text{”} \quad \texttt{Exp;}$$

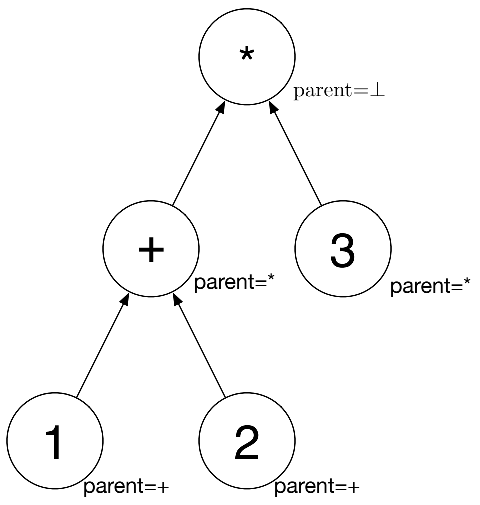


FIGURE 1.3: An annotated parse-tree for sequence "1+2*3" with respect to Γ_5

$$\begin{aligned} \texttt{Exp} &::= \texttt{Exp} \quad \text{“}*\text{”} \quad \texttt{Exp}; \\ \texttt{Exp} &::= [1-9][0-9]*; \end{aligned}$$

$$(\Gamma_5) \quad \begin{aligned} \texttt{Exp[parent!} = \text{“}*\text{”}] \quad &::= \quad \texttt{Exp[parent} = \text{“}+\text{”}] \quad \text{“}+\text{”} \quad \texttt{Exp[parent} = \text{“}+\text{”}]; \\ \texttt{Exp} \quad &::= \quad \texttt{Exp[parent} := \text{“}*\text{”}] \quad \text{“}*\text{”} \quad \texttt{Exp[parent} = \text{“}*\text{”}]; \\ \texttt{Exp} \quad &::= \quad [1-9][0-9]*; \end{aligned}$$

1.5 Attributing System

The system we propose resembles L-attribute grammars (L-AGs) [8] in the way our attributes are evaluated. Like in L-AGs, the attributes specified are evaluated in a depth-first left-to-right manner. We allow the user to add annotated Java code to the productions of the grammar. This Java code is then able to use an API to retrieve and pass on attributes, in addition to performing calculations on these attributes. Finally, we also provide API methods to allow the annotated Java code to prune parse derivations in case attributes conflict semantic rules. Note that we will only prune derivations, not remove then (as this might lead to incorrect results). This API will be discussed in Chapter 3.

The main goal of this research is to incorporate a top-down attribute system into the generalized LL (GLL) parsing to disambiguate normal expression languages and languages that use the offside-rule. Using depth-first left-to-right evaluation of attributes was already deemed feasible for integration in top-down parsers by Müller [21]. Additionally, there have already been some attempts to allow generalized bottom-up parsers to maintain a sense of context with respect to layout-sensitive grammars [22] on-parse time. However, to our knowledge, no such attempts have been made for generalized top-down parsing.

More specifically, the desired disambiguation should take place on-parse time. An on-parse time disambiguation mechanism is desirable as we want to be able to reject invalid trees as soon as possible, preventing unnecessary work. Furthermore, if we were to perform post-parse disambiguation, a parse-forest disambiguation filters would be more powerful and can be generated by means of attribute grammars [18].

In our attribute system, similar to more general attribute grammars, we distinguish between two distinct types of attributes. The first type of attributes that we will discuss are inherited attributes. Inherited attributes are received from parent non-terminals. For instance, whenever a nonterminal `N` is substituted for a sequence α, attributes that are passed from `N` to α are referred to as inherited attributes. Consider again grammar Γ_5. In the right-hand-side (RHS) of the production, the non-terminals have been annotated with attributes and values (e.g. `parent="+"`). When we substitute Exp using one of the productions in Γ_5, that instance of the production is annotated with the attribute `parent`, which has the value "+". Support for inherited attributes in GLL will be discussed in more detail in Section 4.1.

The second type of attributes are synthesized attributes. Synthesized attributes, in contrast to inherited attributes, travel upwards. To illustrate this, consider grammar Γ_6 and do not want both instances of `X` in `S` to derive the same terminal (i.e. disallow "bb" and "cc"). We could do so by passing information on which alternate was chosen to the production `S ::= X   X`, such that it can check that that they are different. See grammar Γ_7, in which we specify that `X ::= "b"` and `X ::= "c"` should annotate their parent non-terminal with attribute "type" with a value corresponding to their production (e.g. $\uparrow$ *type* = "*b*"). At the level of production `S ::= X X`, we can then check that the type of the first `X` and the type of the second `X` differ ($X_0.type \neq X_1.type$). Support for synthesized attributes in GLL will be discussed in more detail in Chapter 4.3.

$$(\Gamma_6) \qquad \begin{aligned} \mathtt{S} &::= \mathtt{X} \quad \mathtt{X};\\ \mathtt{X} &::= \text{“}\mathtt{b}\text{”};\\ \mathtt{X} &::= \text{“}\mathtt{c}\text{”};\end{aligned}$$

$$(\Gamma_7) \qquad \begin{aligned} \mathtt{S}[\mathtt{X}_0.\mathtt{type} \neq \mathtt{X}_1.\mathtt{type}] &::= \mathtt{X} \quad \mathtt{X};\\ \mathtt{X} &::= \text{“}\mathtt{b}\text{”}[\uparrow \mathtt{type} = \text{“}\mathtt{b}\text{”}];\\ \mathtt{X} &::= \text{“}\mathtt{c}\text{”}[\uparrow \mathtt{type} = \text{“}\mathtt{c}\text{”}];\end{aligned}$$

1.6 Summarizing

To summarize, there is a vast amount of literature available regarding attribute grammars. However, all applications of attribute grammars described in this literature serve different purposes than ours. We aim to use attribute grammars to disambiguate parse forests on-parse time. More specifically, we will attempt to extend GLL parsers such that they supports an attribute system similar to L-attribute grammars. In doing so, we aim to prune derivations on-parse time, using the attributes as context to do so.

1.7 Research questions

The objective of this thesis is to implement attribute grammars inside GLL and use them to disambiguate expression and offside grammars on-parse. We will first investigate the feasibility of implementing on-parse time attribute grammars, followed by a number of case-studies to evaluate the ability of on-parse AGs to disambiguate the ambiguous concepts in the grammars mentioned.

Question 1
Is it possible to extend the Generalized LL parsers with attribute grammars?
A search for literature with respect to generalized parsing and attribute grammars has yielded no results. We therefore pose the question is if it is at all feasible to implement attribute grammars into generalized parsing. Due to the depth-first top-down nature of GLL, we will restrict ourselves to the L-attribute grammars [21].

Question 2
Is it possible to reject derivations on-parse by using attributes? (in contrast to post-parse evaluation)
We are interested in evaluating the attributes on-parse in the GLL algorithm. Very powerful post-parse disambiguation is already available in the form of filters[18, 23]. To improve on this we attempt to evaluate our attributes with respect to their conditions on-parse time.In this way, we hope to prune a number of derivations to prevent post-parse disambiguation of the resulting parse forest. This in contrast to post-parse disambiguation, where the parse forest is yielded first, and invalid trees are removed afterwards. A crucial side-note to this is that during this process of pruning, no trees should be removed that do not violate attribute properties. This poses a challenge due to the high amount of sharing inside the data structures used by GLL algorithm.

Question 3
Are attribute grammars suited for practical on-parse time disambiguation?
We want to know if the lack of global information makes on-parse time disambiguation viable for practical uses. We will conclude this thesis with case studies into disambiguating two practical grammars: expression grammars and offside grammars. We choose the expression grammar as this is the cause of ambiguity and are present in most (if not all) modern programming languages.

Chapter 2

Generalized LL parsing

We will start off by explaining the general idea of GLL before proceeding to extending the algorithm to support attributes. As GLL consists of several easy-to-understand components, but with complex interactions, we will incrementally extend an LLRD parser until it becomes a GLL parser. In the next sections we will continuously refer to the code and concepts presented as “parsers”. However, in the explanatory examples we omit actual creation of the parse-tree to keep our examples understandable and compact. Since we do not generate a parse-tree, formally we are constructing a recognizer. In Section 2.10 we will then discuss the creation of the parse-tree, thus transforming the recognizer into a parser.

In an earlier project, an object-oriented Java implementation of the GLL (OOGLL) algorithm was created [1]. This implementation changes the original GLL algorithm in the sense that it uses method-calls rather than GOTO-statements, and attempts to use object-oriented paradigms to clarify the algorithm as much as possible. The code that is presented here, to explain the workings of the GLL algorithm, is based on this OOGLL variant of GLL.

2.1 Scanner

Before going into the details of actual LLRD and GLL parsers, we first need to elaborate on some auxiliary functions, which we will pack into an entity we will be referring to as a “scanner”. A scanner maintains two pieces of data:

- The input sequence we are working on, say σ;
- A position in the input, say i.

Besides operations to retrieve and modify the input position, the scanner has two interesting functions:

- $hasNext(\sigma : String, i : \mathbb{N}, \rho : RegularExpression) : Boolean$ that, given a regular expression ρ, will return true if ρ occurs starting at i in σ. For example: `hasNext("world", 0, w)` yields true, whereas `hasNext("world", 3, w)` and `hasNext("world", 0, x)` both yield false.
- $next(\sigma : String, i : \mathbb{N}, \rho : RegularExpression) : String$ also takes a regular expression ρ. if σ contains ρ starting from position i, return the occurrence of ρ from the input sequence and place the input pointer at the end of this occurrence. For instance, `next("world", 0, w) = "w"`. This is particularly useful when dealing with symbols ρ that are defined as regular expressions. For example: `next(``hello world", 6, [a-z]+) = "world"`.

An example of a scanner in Java can be found in Listing 2.1. Keep in mind that σ and i are instance variables of the scanner, and are therefore not present in the `next` and `hasNext` method calls. Note that the example implementation provided does not support regular expression (it uses substring equality). An implementation that does use regular expressions is more common, but does not add to the illustrative capabilities of our examples, and hence will be omitted. Note that if a regex implementation is used, our function `next` is required to return the sub-sequence from the input sequence, rather than just the pattern. For instance, `hasNext("1234", 0, [0-9]*) = "1234"`.

```java
class Scanner {
  private int inputPointer;
  private String sequence;

  public boolean hasNext(String symbol) {
    return sequence.subSequence(inputPointer).startWith(symbol);
  }
  d
  public String next(String symbol) {
    if (hasNext(symbol)) {
      inputPointer = inputPointer + symbol.length();
      return symbol;
    }
  }
}
```

Listing 2.1: A Java implementation of a scanner

One other thing to note is that, for now, the scanner is global, and retains its state across all parse functions.

2.2 LLRD

Now that we have seen an example of a scanner, we will start looking at a parser. We will first look at an LLRD parser for grammar Γ_8. The generation of such a parser is relatively straightforward and will be elaborated upon next.

$$(\Gamma_8) \qquad \begin{aligned} \texttt{S} ::= \texttt{A} \quad \text{“c”} \quad | \quad \texttt{B} \quad \text{“c”}; \\ \texttt{A} ::= \text{“a”}; \end{aligned}$$

`B` ::= “`b`”;

An LLRD parser usually has a single function per production rule of the grammar, we call these *parse functions*. Within an LLRD parse function, every right-hand-side (RHS) literal of the corresponding production rule is processed in order. If a literal is a terminal, we call the *next* function in our scanner, to “consume” the symbol. If this symbol cannot be consumed, an error is thrown, as (apparently) the input is not valid with respect to the grammar. If the literal is a non-terminal, we check what symbols the given non-terminal can start with. If the first symbol of the input sequence corresponds to one of these symbols, we invoke the parse function corresponding to the non-terminal (e.g. `parseA()`, if we encountered the non-terminal `A`). A possible implementation can be found in Listing 2.2. There also exists a backtracking variant of the LLRD parser, which does not require us to check the first symbol of a non-terminal, but returns to an earlier point in the parse if it fails within a recursive non-terminal. We will not discuss this variant any further, as GLL is based on the non-backtracking variant of LLRD parsers.

An LLRD parser strongly relies on the call-stack. That is, whenever a parse function for production `A ::= x Z y` invokes the parse function `parseZ()` for non-terminal `Z`, a lot of information is stored implicitly: after completing `parseZ()`, proceed by processing `y` in `A`.

```
Scanner scanner;
void parse(String i) {
  scanner = new Scanner(input);
  parseS();
}

void parseS() {
  if (scanner.hasNext("a")) {
    parseA();
    scanner.next("c");
  } else if (scanner.hasNext("b")) {
    parseB();
    scanner.next("c");
  } else {
    error();
  }
}

void parseA() {
  consumeInput("a", ptr);
  ptr++;
}

void parseB() {
  consumeInput("b", ptr);
  ptr++;
}
```

Listing 2.2: An implementation of a classic LLRD parser

2.3 LLRD with a higher granularity

Working towards a GLL parser, we will first increase the granularity of our functions and introduce a concise naming scheme for our parse functions. In a later extension, we will reduce dependency on the call-stack, as the call-stack limits us to a single derivation. Therefore, we fragment parse functions such that every point that the call-stack would return to (e.g. behind each non-terminal) becomes a separate function. When no longer using the call-stack, these return points are then uniquely addressable by their functions. For now, consider the higher granularity.

We will first define the concept of a *GLL Block*. Let $head(s)$ and $last(s)$ denote the first and last elements of a sequence s respectively. Furthermore, $tail(s)$ is the sequence s without $head(s)$. Any alternate may be divided into one or more GLL Blocks. A GLL Block for an alternate α is a sequence σ such that either $\sigma = \epsilon$ or $\sigma = t \mathbin{+\!\!+} last(\sigma)$ (where $a \mathbin{+\!\!+} b$ denotes the string concatenation of a and b).

- t is a sequence of terminals;
- σ is a sub-sequence of α;
- $last(\sigma)$ is either:
  - a non-terminal;

  - a terminal that is not followed by any literal;

- *head*(s) is not preceded by a terminal.

To illustrate: given an alternate $\alpha =$ `a b C D`, would be partitioned into three GLL Blocks: $\beta_1 =$ `a b C`, β_2= `D`, and an empty Block β_3 (the use of this last Block will be explained later). The concatenation of β_1 through β_3 again yields `a b C D`, which is equal to α. Observe that we can create a parser equivalent to the one in Listing 2.2 by calling GLL Blocks in order. That is, if a production consists of n GLL Blocks, at the end of the first Block, the function for the second Block is called, and so on until the n^{th} Block.

More formally we create the following functions:

- A function `parseP`() for every production `P` ::=Ψ (where Ψ is a set of alternates). This function will check the applicability of alternates and invoke the correct one.
- A function `parseP`$_{\texttt{i,j}}$() for every GLL Block j in alternate i of production P. The functions for all GLL Blocks of an alternate will, together, be responsible for parsing that specific alternate.

To improve the conciseness, we will introduce a fixed naming scheme for these $P_{i,j}$. We define the name of a function $P_{i,j}$ by means of the label function:

$$label(P_{i,j}) = \begin{cases} i = 0 & \rightarrow Pi \\ otherwise & \rightarrow Pi\_j \end{cases}$$

For example, take production rule `P` ::= `a B C | d | E F g H`, we would partition and name it as can be seen in Table 2.1. In our example grammar, this would mean generating functions `A(), A0(), B()`, and `B0()`. However, as `A()` only schedules `A0` and `B()` only schedules `B0`, for the sake of clarity, we combine these functions into `A()` and `B()` respectively. At this stage, this increase in functional granularity might seem like overhead, but will become of use in later extensions. Note, that the parser presented in Listing 2.3 performs equal to the parser in Listing 2.2.

```
Scanner scanner;
void parse(String input) {
  scanner = new Scanner(input);
  parseS();
}

void parseS() {
  if (input.has("a")) {
    parseA();
    parseS2();
  } else if (input.has("b")) {
    parseB();
    parseS2();
  }
}

void parseS2() {
  scanner.next("c");
}

void parseA() {
  scanner.next("a");
}

void parseB() {
  consumeInput("b");
}
```

Listing 2.3: LLRD with increased granularity

```
Scanner scanner;
void parse(String input) {
  scanner = new Scanner(input);
  parseS();
}

void parseS() {
  if (input.has("a")) {
    parseS0();
  } else if (input.has("b")) {
    parseS1();
  }
}

void parseS0() {
  parseA();
  parseS0_0();
}

void parseS0_0() {
  scanner.next("c");
}

void parseS1() {
  parseB();
  parseS1_0();
}

void parseS1_0() {
  scanner.next("c");
}

void parseA() {
  scanner.next("a");
}

void parseB() {
  consumeInput("b");
}
```

Listing 2.4: LLRD with increased granularity and concise naming scheme

2.4 LLRD with modified control flow

The next modification to our parser will be to its control-flow. Say we have two productions $\phi_1 ::= \beta_1\beta_2$ and $\phi_2 ::= \beta_3$. Say β_1 invokes the parse function for ϕ_2. Rather than having the function for β_3 return (via the call-stack) to ϕ_1 and having ϕ_1 invoke the function for β_2, we will have β_3 do this job. For a more specific example, consider input sequence "`ac`" (still under grammar Γ_8). The sequence of function invocations in the previous is as follows:

- We call `parseS();`
- `parseS` checks that we can parse and "a" and invokes `S0()`

- `S0()` invokes `A()`
- `A()` consumes "a"
- `A()` returns to `S0()`
- `S0()` invokes `S0_0();`
- `S0_0` consumes "c"
- `S0_0` returns to `S0`, returns to `S`, which terminates.

In our modified variant, we change this to the behavior mentioned below. This update will enable us (in a later extension) to reduce dependency on the call-stack. Rather than have the call-stack dynamically tell us where to continue. We define where to continue ourselves.

- We call `parseS();`
- `parseS()` checks that we can parse "a" and invokes `S0()`
- `S0()` invokes `A()`
- `A()` consumes "a"
- `A()` invokes `S0_0()`
- `S0_0()` consumes "c"
- `S0_0` returns via multiple steps and terminates

To realize this modification, we will move the invocation of `S0_0()` to the end of `A()`. Likewise, the last statement of `B()` will invoke `S1_0()`. There is however one problem: consider grammar Γ_9. Note that the parse function for non-terminal `X` can be invoked from the first GLL Block of both `A` and `B`. Thus, after `X` completes, it is unclear whether we should jump to the second GLL Block for `A` or the second GLL Block for `B`. Thus, this approach does not work for all grammars. It possible to return to both, or to store where to return to. This will be included in a later extension, fixing this problem.

(Γ_9)

```
S ::= A   |  B;
A ::= X   C;
B ::= X   D;
X ::= "a";
C ::= "b";
D ::= "b";
```

The main advantage to this functional decomposition is once a single function terminates and starts popping from the call-stack to resume a previous parse function, there are no more statements to execute in any of the parse functions that remain on the call-stack. We will need this observation in our extension with scheduling. An example of the resulting parser can be found in Listing 2.5.

```
Scanner scanner;
void parse(String input) {
  scanner = new Scanner(input);
  parseS();
}

void parseS() {
  if (input.has("a")) {
    parseS0();
  } else if (input.has("b")) {
    parseS1();
  }
}

void parseS0() {
  parseA();
}

void parseS0_0() {
  scanner.next("c");
}

void parseS1() {
  parseB();
}

void parseS1_0() {
  scanner.next("c");
}

void parseA() {
  scanner.next("a");
  parseS0_0();
}

void parseB() {
  consumeInput("b");
  parseS1_0();
}
```

Listing 2.5: LLRD with a modified control-flow

2.5 LLRD with an explicit stack

From this point on, we will omit the initial Parse(input : String) function for the sake of briefness.

Now that we have modified the control flow of the algorithm, we are ready to decrease our dependency on the call-stack. Before going into the details, we first declare an auxiliary function *invoke*(*label* : *String*), that (as the name might suggest) invokes the function corresponding to the provided label. For our example code in Listing 2.7, this function might look as presented in Listing 2.6. This might seem like a work-around for people used to languages that support function pointers, such as C. However, in languages that do not, such as Java, options for other constructs are limited.

```
void invoke(String label) {
  if (label.equals("S")) {
    S();
  } else if (label.equals("S1")) {
    S1();
  } else if (label.equals("S1_0")) {
    S1_0();
  } else if
    ...
  }
}
```

Listing 2.6: Auxiliary function for method invocation

The next parser, found in Listing 2.7 no longer uses the call-stack to store information on what literals remain to be processed. Additionally, we observed that after a function f_1 invokes another parse function f_2, f_1 never processes more literals, even after control is returned to it. As an effect, we can be more effective and no longer store where we need to continue (i.e. the successive GLL Block) as opposed to where we came from. Now we have the same behavior as in our previous parser, with the exception that this works for any context-free grammar.

```
Stack<String> stack;

void parseS() {
  if (input.has("a")) {
    parseS0();
  } else if (input.has("b")) {
    parseS1();
  }
}

void parseS0() {
  stack.push("S0_0");
  parseA();
}

void parseS0_0() {
  scanner.next("c");
}

void parseS1() {
  stack.push("S1_0");
  parseB();
}

void parseS1_0() {
  scanner.next("c");
}

void parseA() {
  scanner.next("a");
  invoke(stack.pop());
}

void parseB() {
  consumeInput("b");
  invoke(stack.pop());
}
```

LISTING 2.7: LLRD with an explicit stack

2.6 Scheduled LLRD with an explicit stack

The next extension to our parser will be that of scheduling. We will no longer invoke parse functions right away, we will schedule them. In this way, we can execute our parse functions in a more controlled way. This will, in the next extension, provide us with the power to support multiple derivations. For this scheduling extension, we use the observation that once a function returns (by means of the call-stack), no more statements are executed in any of the functions it (recursively) returns to.

```
void schedule(String nonTerminal) {
  queue.push(nonTerminal);
}
```

LISTING 2.8: The auxillary function "schedule"

In the code for scheduled LLRD, we replace calls to non-terminals with invocations of the "schedule" function (presented in Listing 2.8) with the desired non-terminal as argument.

To process them, we add a function "main" that continuously process queued items. The function "parse(input : String)" should invoke this function after it has completed its work. In more detail:

- The loop inside the function `main()` invokes a parse function (say `f()`) corresponding a non-terminal that is in the queue (the queue is initialized to contain the start non-terminal).
- The parse function (`f()`) begins execution.
- If the last literal of the production related to the executing parse-function is a non-terminal, that non-terminal is added to the queue by means of the "schedule" function.
- By definition our parse functions have no more statements to execute after this schedule takes place, hence control is passed back to the main function.
- Main invokes the non-terminal that was just scheduled, and the cycle repeats.

The parser to implement this scheme for grammar Γ_8 can be found in Listing 2.9. Again note that this code only works because we use a queue. As we still have a global input pointer, the order in which we process our queued work is relevant. This will be alleviated in the next extension. As the sequence above is quite abstract, consider the more illustrative sequence below. In this sequence we walk through a derivation of the input sequence `"ac"`.

- Inside the function `parse()`, the queue is initialized to contain the start non-terminal `S` ([S]). Our stack at this point in empty ($\perp$).
- The last statement in `parse()` invokes `main()`.
- The loop inside main() removes `S` from the queue and invokes its parse function `parseS()`. The function `hasNext("a")` yields true, as `"a"` is the first token in our input sequence. Thus, label `S0` is added to the queue via the schedule function. `parseS()` returns (by means of the call-stack) and resumes the main function. Our queue at this point is [S0] and our stack is still $\perp$.
- As the queue is not empty, we perform another iteration of our main loop, removing `S0` from the queue and invoking `parseS0()`. First the label `S0_0` is pushed onto the stack. Next, label `A` is added to the queue by means of the `schedule()` function. This leaves us with queue [A] and stack [$\perp$, S0_0]. Finally, control is passed back to main (again by means of the call-stack).
- Inside the main loop, `A` is removed from the queue, and `parseA()` is invoked. An `"a"` is scanned on the input, advancing the input pointer to position 1 (i.e. before the `"c"`). Next we should invoke the label that is on top of the stack. So, `stack.pop()` will return $S0\_0$ which will be invoked. This leaves us with an empty queue and stack. However, control flow is now passed to `S0_0()`.

- Function `S0_0()` will scan a `"c"` on the input, advancing the input pointer to the end of the input (position 2). Next, via the call-stack, we return to `S0()`, which has no statements left to execute. Thus, we return (again via the call-stack) to main(). Now, the queue is empty and the loop will terminate. At this point our scanner has reached the end of the input and thus our derivation was correct.

Please note that the parser we have presented in Listing 2.9 is still equal to the parser from Listing 2.2. In our next extension, we will step away from this equality and support multiple derivations.

```
Stack<String> stack;
Stack<String> queue;

void parse(String input) {
  ...
  schedule("S");
  main();
}

void main() {
  while (queue ≠ ∅) {
    invoke(queue.remove());
  }
}

void parseS() {
  if (input.has("a")) {
    schedule("S0");
  } else if (input.has("b")) {
    schedule("S1");
  } else {
    error();
  }
}

void parseS0() {
  stack.push("S0_0");
  schedule("A");
}

void parseS0_0() {
  scanner.next("c");
}

void parseS1() {
  stack.push("S1_0");
  schedule("B");
}

void parseS1_0() {
  scanner.next("c");
}

void parseA() {
  scanner.next("a");
  invoke(stack.pop());
}

void parseB() {
  consumeInput("b");
  invoke(stack.pop());
}
```

LISTING 2.9: LLRD with scheduled method invocation

2.7 Scheduled LLRD with multiple stacks : GLL

We will now step away from LLRD, and into GLL. That is, our algorithm will now yield all derivations, instead of just one. To do so, we need a couple of different extensions:

- Multiple stacks (one per derivations);
- An input pointer per derivation.

To keep track of each derivation, we will introduce the concept of a descriptor. A descriptor is a three-tuple containing:

- The label of the next function to parse;
- A stack;
- an input position.

Note that in our previous parser, what kept us from supporting multiple derivations was that we only had a single stack and a global input pointer. Now, each descriptor has its own stack and input pointer, we are no longer limited to a single derivation. Each descriptor now contains the work that has been, and still has to done, on a particular derivation. We denote a descriptor with label L, stack Σ, and input position i as $\langle L, \Sigma, i\rangle$ (in illustrations, descriptors will always be illustrated by angular brackets or hexagons).

To properly illustrate, we first need a grammar that actually produces multiple derivations. Consider the input sequence “abc” under Grammar Γ_{10}.

$$(\Gamma_{10}) \qquad \begin{aligned} \texttt{S} ::= \texttt{A} \quad \text{“}\texttt{bc}\text{”} \quad | \quad \texttt{B} \quad \text{“}\texttt{c}\text{”};\\ \texttt{A} ::= \text{“}\texttt{a}\text{”};\\ \texttt{B} ::= \text{“}\texttt{ab}\text{”}; \end{aligned}$$

With this extension, whenever we encounter an ambiguity (i.e. two alternates that are applicable for a single non-terminal) we create a descriptor with a duplicate of the stack for each of these alternates. In this way, we execute both complete derivations. We can do this because each descriptor has a private stack and input pointer that it (and its child descriptors) work on. When finished, each descriptor $\langle L, \perp, i\rangle$ that has no more characters in its input sequence which have not been consumed at the end of a parse-function represents a successful derivation. This is the case because there is no more work to be performed (as its stack is empty) and the entire input has been consumed. More implementation oriented: whenever a `pop()` is executed, the stack of the current descriptor is $\perp$, and the position of the scanner is equal to the length of the input sequence, our derivation is successful.

```
Stack<⟨Identifier, Stack, Integer⟩> queue;

Stack currentStack;
int currentPointer;

void parse(String input) {
  queue.add(⟨S, ⊥, 0⟩);
  while (!queue.isEmpty()) {
    ⟨label, stack, ptr⟩ ←queue.remove()
    currentStack = stack;
    currentPointer = ptr;
    invoke(label);
  }
}

void parseS() {
  if (scanner.has("a", currentPointer)) {
    currentStack.push("S0\_0");
    queue.add(⟨A, currentStack.copy(), scanner.getPosition();⟩);
  }
  if (scanner.has("a", currentPointer)) {
    currentStack.push("S1_0");
    queue.add(⟨B, currentStack.copy(), scanner.getPosition();⟩);
  }
}

void parseA() {
  consume("a");
  Label l = currentStack.pop();
  queue.add(⟨l, currentStack, scanner.getPosition();⟩);
}

void parseB() {
  consume("ab");
  Label l = currentStack.pop();
  queue.add(⟨l, currentStack, scanner.getPosition();⟩);
}

void parseS0_0() {
  consume("bc");
  queue.add(⟨l, currentStack, scanner.getPosition();⟩);
}

void parseS1_0() {
  consume("c");
  queue.add(⟨l, currentStack, scanner.getPosition();⟩);
}
```

Listing 2.10: Initial version of GLL that uses stack replication

2.8 The Graph Structured Stack

Note that here we duplicate the stack for every ambiguous alternative that we encounter. This can become very expensive for highly ambiguous grammars. Something that one might observe is that whenever we copy a stack, both stacks will have an identical prefix. The obvious solution to this is to share this common prefix and save on the copy operations. To accomplish this, GLL uses a slightly modified version of the Graph Structured Stack by Tomita [24]. The modification is that the GSS in GLL can have

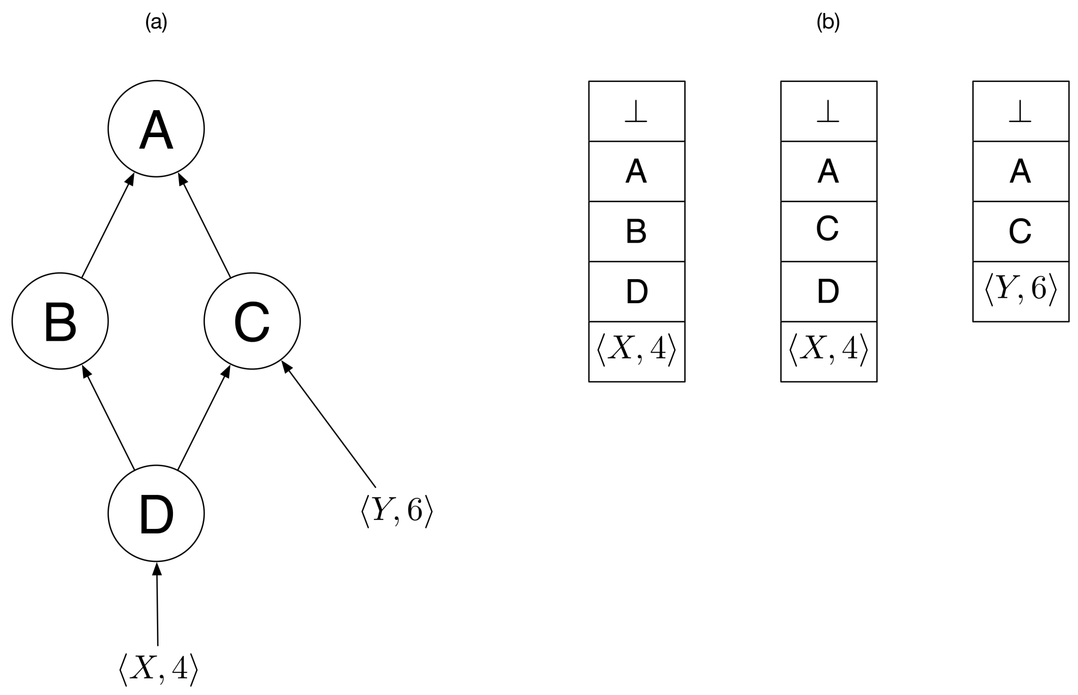


Figure 2.1: (a) A GSS with 3 descriptors scheduled. (b) 3 stacks with descriptors scheduled on top of them

self-loops. The use of these self-loops will be used to detect left-recursion (which will be explained in more detail in our next extension). The ability to deal with left-recursion allows support of the entire class of context-free grammars.
With the GSS, we are no longer replicating complete stacks only to schedule functions on top of them (or to push different non-terminals). We use a single centralized data structure to store all stacks that have been constructed so far. Additionally, these stack states are persistent. That is, even if there are no derivations currently using a particular stack state, it is not removed. Because we do not remove this information, we can use it later on to detect sharing of stacks between derivations. In Figure 2.1 one can see a GSS with three scheduled descriptors in (a), and the three stacks (with descriptors scheduled on them) it would require to encode the same information without a GSS in (b).

2.8.1 The pop() function

Now that we have a graph structured stack, we no longer have a unique parent for a stack head. So, when we perform a `pop()` on the current stack (which is actually a stack head) it is possible to end up with multiple new stacks. In order to maintain the paradigm from stack copying, work has to continue on each of these new stacks. To handle this in an elegant way, we introduce the `pop()` method in Listing 2.11.

```
void pop() {
  for (GSSNode n : currentStack.parents()) {
    queue.add(⟨currentStack.getLabel(), n, scanner.getPosition()⟩);
  }
}
```

Listing 2.11: The pop method for GLL

To illustrate the workings of this method, consider the following example: Say we have a descriptor $d = \langle X, Y, Z \rangle$, where stack-pointer $Y = (l_y, i_y)$ has three parenting GSS nodes W_1, W_2, W_3. Following our analogy, after completion of d (resulting in an input position i_d), we need to continue parsing of l_y at position l_i. After l_y has finished, we see that there are three stacks (derivations) that we need to continue working on W_1, W_2, and W_3. Thus we need to create three descriptors to accomplish this:

- $\langle l_y, W_1, i_d \rangle$
- $\langle l_y, W_2, i_d \rangle$
- $\langle l_y, W_3, i_d \rangle$

Such a `pop()` is performed whenever we finish parsing a production, and perform a `pop()` to continue parsing the parent production (i.e. the production that invoked the production we just finished). In general we can say that,all these descriptors now derive a single non-terminal `X`$_4$ at the same input position `Z`$_4$.

2.8.2 The push function

Now that we have a special method for performing pops, lets also make an additional method for pushing (Listing 2.12).

```
void push(Label label, int pointer) {
  GSSNode newStack;
  if (gss.has((label,pointer))) {
    newStack = gss.get((label,pointer));
  } else {
    newStack = gss.create((label,pointer));
  }
  gss.addEdge(newStack, currentStack)
}
```

Listing 2.12: The push functon

2.9 Efficient GLL

Now that we have our efficient stack and the `pop()` method to accompany it, we add one more function: namely one for scheduling. This new scheduling function will check if a particular descriptor has already been created before (Listing 2.13). This is allowed, because if a particular descriptor already exists, the descriptor(s) responsible for the subsequent parts of the derivation have already been scheduled. Correctness of this follows directly from the fact that we are dealing with a context-free grammar. Consider the GLL parser presented in Listing 2.14. Additionally, not checking for duplicate descriptors will cause the algorithm to break on left-recursive grammars, as will be explained later in this section.

```
void schedule( Label label) {
  d = ⟨label, currentStack, currentPointer⟩
  if (!processedDescriptors.contains(d) && !queue.contains(d)) {
    queue.add(d);
  }
}
```

Listing 2.13: The schedule functon

```
Stack<⟨Identifier, Stack, Integer⟩> queue;
Set<⟨Identifier, Stack, Integer⟩> processedDescriptors;

GSSNode currentStack;
int currentPointer;

void parse(String input) {
  queue.add(⟨S, ⊥, 0⟩);
  while (!queue.isEmpty()) {
    ⟨label, stack, ptr⟩ ←queue.remove()
    processedDescriptors.add(⟨label, stack, ptr⟩);
    currentStack = stack;
    currentPointer = ptr;
    invoke(label);
  }
}

void parseS() {
  if (scanner.hasNext("a", currentPointer)) {
    currentStack = push("S0_0");
    schedule("A");
  }
  if (scanner.hasNext("a", currentPointer)) {
    push("S1_0");
    schedule("B");
  }
}

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

void parseB() {
  scanner.next("ab");
  pop();
}

void parseS0_0() {
    scanner.next("bc");
    pop();
  }

  void parseS1_0() {
    scanner.next("c");
    pop();
  }
```

Listing 2.14: A GLL recognizer

The version of GLL we have presented includes an implicit mechanism for handling left-recursion. The handling of left-recursion is hidden in the tight collaboration of the self-loops of the GSS, together with the fact that we do not create duplicate descriptors or GSS nodes. Whenever we would push a left-recursive non-terminal, we instead create a self-loop at that stack-head. This encodes all possible levels of left-recursion. This can be seen more clearly in Figures 2.3 (a) and (b). (a) Shows an example of a GSS with a self-loop where three descriptors have been scheduled on various stack heads. (b) Shows the same information, encoded in regular stacks. Lets now take a look at an actual example. Assume we have a production `P` with left-recursive alternate `P0 ::= P "a"` and an non-left-recursive alternate `P1 ::= "a"`. Consider the workings of the GLL algorithm illustrated in Figure 2.2.

1. Non-terminal `P` is scheduled on the empty stack. The input position at thus point is 0, thus the descriptor scheduled is $\langle \mathtt{P}, \bot, \mathtt{0} \rangle$
2. Both alternates of `P` (i.e. `P0` and `P1`) are applicable and are thus scheduled on the same stack as `P` (the input position is still 0).
3. `P0` is processed, its first GLL Block consists of non-terminal `P`. The succeeding GLL Block (`P0_0`) and input position (0) is pushed, and the corresponding non-terminal (`P`) is scheduled on top.
4. The `P0` descriptor scheduled on stack-head $(\mathtt{P0}, \mathtt{0})$ is processed. Again we need to push the corresponding non-terminal and input position $(\mathtt{P0\_0}, \mathtt{0})$. However, this node already exists, hence we reference it. Next an edge should be created from the "new" node to its parent. However, these nodes are the same, thus a self-loop is added. `P` is not scheduled on this stack top, as the descriptor $\langle \mathtt{P}, \mathtt{P0\_0}, \mathtt{0} \rangle$ was already scheduled at some point. The state after this step has been illustrated in (5).
5. Here we see the state after the self-loop has been created.
6. Now, the non-left-recursive alternate $\mathtt{P1}, \mathtt{0}$ ($\langle P1, P0\_0, 1 \rangle$) is processed on the stack-head $(\mathtt{P0\_0}, \mathtt{0})$ two new descriptors are created:
   - $\langle P0\_0, \bot, 1 \rangle$, representing that this is the correct amount of recursive calls, and we should continue.
   - $\langle P0\_0, P0\_0, 1 \rangle$, representing that we want to parse another recursive invocation.

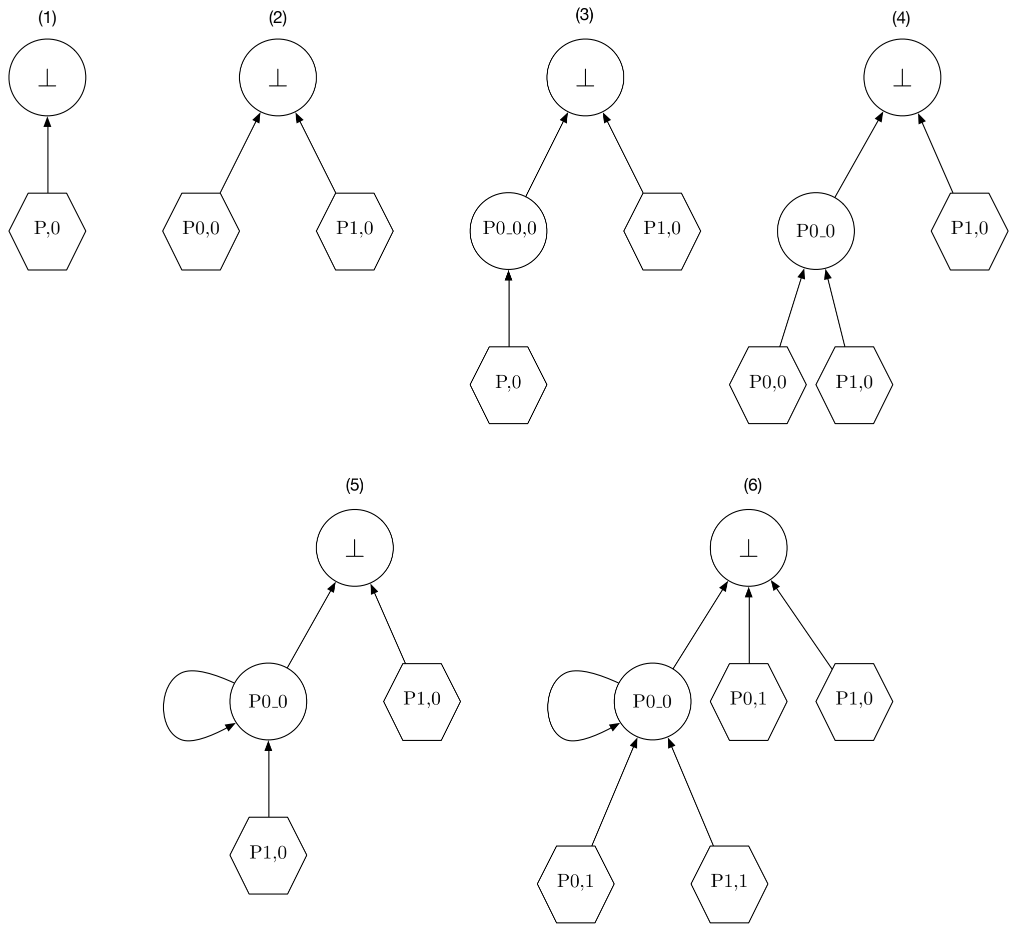


FIGURE 2.2: Illustration of the workings in GLL w.r.t. left recursion. GSS nodes are represented by circles, and descriptors by hexagons.

Note that these new descriptors may be created (and are not duplicates) because in step 5, $P1$ advanced the input pointer.

2.10 GLL with SPPF

Finally, we will present an actual GLL parser (i.e. including parse tree creation). We will also pinpoint locations where parse-forest data is being handled. However, we will abstract from actual calls, as the specific calls add little to the didactic value of the example and may be found in the original paper on GLL [2].

2.10.1 The concept behind the SPPF

But before we can begin looking into the actual parser, lets take a closer look at how the shared packed parse forest (SPPF) is actually constructed. Take the example grammar Γ_{11}. We illustrate the parse of input sequence "`abc`" using this grammar in Figure 2.4.

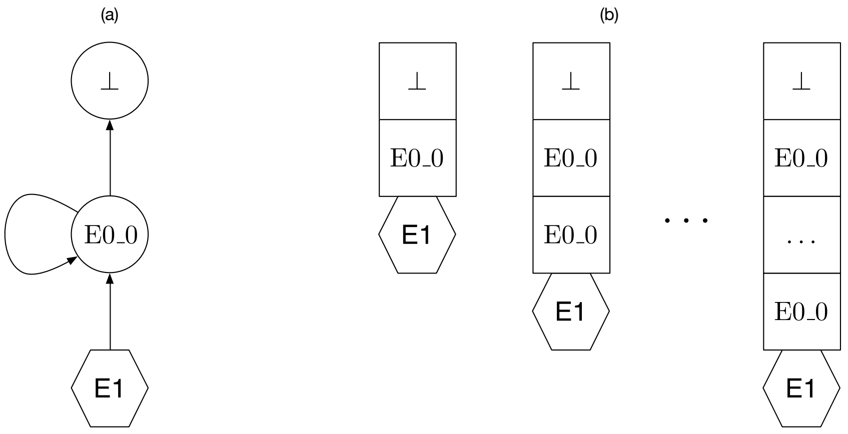


FIGURE 2.3: (a) A GSS with a self-loop on which a descriptor with function E1 has been scheduled. (b) The data from the GSS represented by normal stacks

The core idea for SPPF creation is that consecutive descriptors in a production pass incrementally bigger trees along via their descriptors. If intermediate non-terminals are invoked, references to these trees are stored on the stack. When the non-terminal finishes, its partial SPPF can then be combined with the partial SPPF that exists on the stack. Let `B` ::= $\alpha \cdot \beta$ denote a point in a derivation, where α has been successfully processed, but β has not. The construction of an SPPF has been illustrated in Figure 2.4 and described in the enumeration below.

1. Figure 2.4 (a) shows the normal start of the algorithm. Our start non-terminal is scheduled on the empty stack. If we process this descriptor, we will create a node in our parse forest for the terminal symbol `a`, push return location `E0_0` onto the stack, and schedule the descriptor for `B` on this stack. The parse-forest-node for `a` is placed on the GSS edge. The resulting state is visible in 2.4 (b).

2. When processing `b` from the production `B` ::= `b`, we create SPPF nodes for the terminal and non-terminal. Next, we see we have finished the production and we create additional nodes `B` ::= `b`· and `B` to specify that `b` was recognized as part of the production `B` ::= `b` (appending `b` as a child). As we are at the end of a production, the `pop()` statement is executed. In this `pop()` statement, the `a` node is retrieved (since we are traversing back along this edge to the empty stack, this is possible) and combine our partial trees. Now, when scheduling the descriptor for `E0_0`(), we pass a reference to this new tree along in our new descriptor. This can be seen in 2.4 (c).

3. `E0_0` parses the `c` and is joined, via a new node, to the SPPF that was passed along in the descriptor for `E0_0`. Some additional nodes are attached for administrative purposes, and our SPPF is complete. The resulting SPPF can be seen in Figure 2.4 (d).

(Γ_{11}) `E` ::= “`a`” `B` “`c`”;

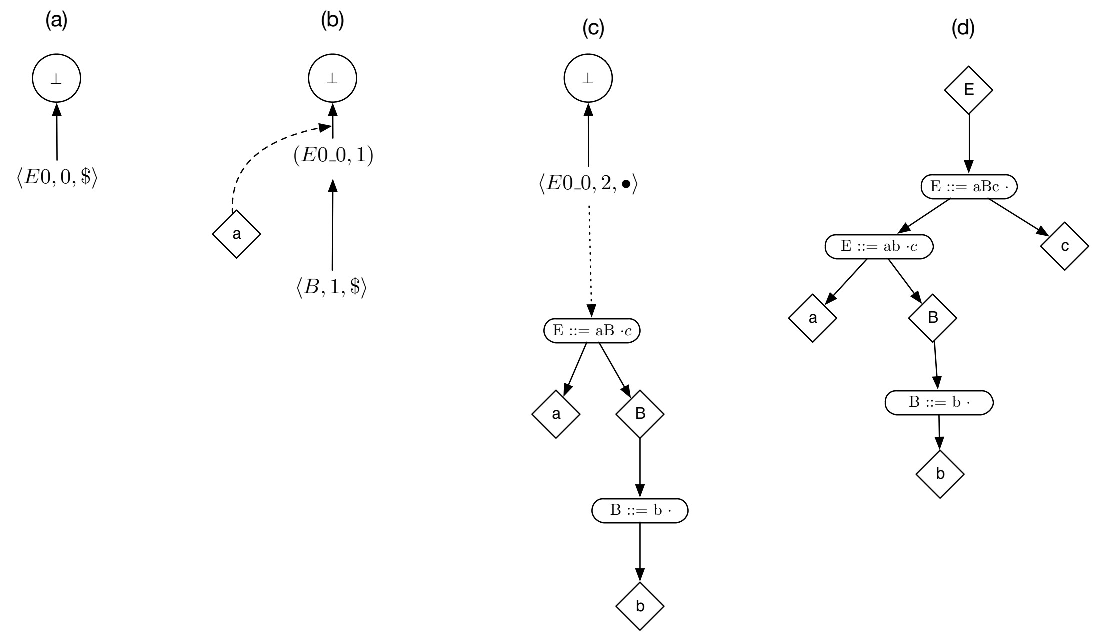


FIGURE 2.4: Illustration of parse forest construction in GLL, extents have been omitted for clarity.

`B ::= “b”;`

2.10.2 Implementation

So now that we have described the concepts of the SPPF, let us look at where the various steps are taken. We abstract from the actual SPPF methods, by replacing them by Ψ to indicate that at this position in the code, work on the SPPF is performed. The grammar for this parser is grammar Γ_{12} and the result may be found in Listing 2.15.

```
(Γ12)                S  ::= A  S  “d”  |  “a”  S  |  ;
                                  A  ::=  “a”;
```

Please note that the code presented is a simplified version of actual generated code, and might thus differ in shape slightly from the other examples. This choice was made to bridge the gap between the examples and actual generated parsers.

```
void S() {
  if (scanner.hasNext("a")) {
    schedule("S0");
  }
  if (scanner.hasNext("a")) {
    schedule("S1");
  }
  if (scanner.hasNext("")) {
    schedule("S1");
  }
}
void A() {
  if (scanner.hasNext("a")) {
    schedule("A0");
  }
}

void S0() {
  push("S0_0");
  A();
}

void S0_1() {
  push("S0_2");
  S();
}

void S0_2() {
  if (scanner.hasNext("d")) {
    Ψ
    scanner.next("d");
    Ψ
  } else {
    return;
  }
  pop();
}

void S1() {
  if (scanner.hasNext("a")) {
    Ψ
    scanner.next("a");
  } else {
    return;
  }
  push("S1_0");
  S();
}

void S1_1() {
  pop();
}

void S2() {
  if (scanner.hasNext("")) {
    Ψ
    scanner.next("");
    Ψ
  } else {
    return;
  }
  pop();
}

void A0() {
  if (scanner.hasNext("a")) {
    Ψ
    scanner.next("a");
    Ψ
  } else {
    return;
  }
  pop();
}

```

Listing 2.15: A GLL parser

Production	Alternate	GLL Block	Label
$P ::= a \quad B \quad C \mid d \mid E \quad F \quad g \quad H$	aBC	$a \quad B$	P0
		C	P0_0
			P0_1
	d	d	P1
	$E \quad F \quad g \quad H$	E	P2
		F	P2_0
		$g \quad H$	P2_1
			P2_2

Table 2.1: An example of the labeling scheme for GLL Blocks

Chapter 3

Using the attribute system

In Chapter 2 we have explained GLL and the construction of the SPPF in detail. We will continue with explaining how we can integrate attributes into our GLL implementation. Before we do this, we will explain our notion of attributes in more detail.

3.1 What can attributes do?

The main goal for the implementation of attributes into GLL is to use them as a framework for disambiguation. More specifically, we will focus on pruning sub-trees whenever the attributes conflict a set of parameters specified for a production. This leaves a few questions to be answered:

- How do we specify these attributes?
- How do we specify calculations on these attributes?
- How do we modify GLL to actually prune sub-trees?

All these questions will be answered in this section, starting with the latter.

3.2 Where to disambiguate?

First we need to make a decision on where the application can decide if the current derivation should be pruned. Consider inherited attributes. As these attributes are only passed down, the only relevant place to disambiguate is before we start processing any literals of a production. This is possible because processing additional literals yields no additional information. All information is passed down, and is thus already available before we have processed even the first literal.

Synthesized attributes are only available after processing a particular non-terminal. Therefore we only disambiguate when all non-terminals have been evaluated. To accomplish this, note that a production is always of the shape `X` $::=\alpha \uparrow$, where α is a sequence of literals and $\uparrow$ is the `pop()` statement. Just before $\uparrow$ would thus seems like an appropriate place to disambiguate, as all literals have been processed.

We might not need all non-terminals to make our decision. Therefore we also want to be able to disambiguate after every non-terminal. However, we only know that a non-terminal has completely been processed in the successive GLL Block of that non-terminal. For example: assume presence of a production `X` $::=\beta_1\beta_2$, where β_1 and β_2 are successive GLL Blocks and $\beta_1 \equiv$ `abC`. We can only be certain of the full completion of C at the start of β_2, as (recursive) GLL Blocks scheduled by β_1 might cause a failure before we reach β_2 (i.e. C was not parsed successfully).

To illustrate, consider the following productions, where every disambiguation moment has been marked by a $\downarrow$.

- $X ::=\downarrow a \downarrow B \downarrow C \downarrow\uparrow$ $(a \in T;\ B, C \in N)$
- $X ::=\downarrow abc \downarrow\uparrow$ $(a, b, c \in T)$

In order to be able to address these moments of disambiguation we introduce a naming scheme. The disambiguation point at the start of a productions LHS will be addressed as "`@pre(all)`", whereas the point at the end of a production (before popping) is addressed as `@post(all)`. To address the disambiguation point preceding the i^{th} non-terminal in a productions RHS, the corresponding address is "`@pre(i)`" (e.g. `@pre(0)` or `@pre(7)`).

3.3 Addition to BNF

We propose an extension to BNF. In this Attributed-BNF, one may specify attribute synthesis, inheritance, or deny statements annotated with disambiguation point specifications. A specification of this attributed BNF will be provided shortly, but to illustrate the idea, please consider:

```
@pre(all): attributes.put("priority", 5);
@pre(3): deny;
@post(all): synthesize("expressionType", "Integer");
```

As we are using a Java implementation of GLL [1] (as described in Chapter 2), we allow annotation of arbitrary Java code to be inserted and we define our attribute system on top of this. This approach saves a great deal of time, as we need not develop our own support for evaluation of attributes such as integers, doubles, or strings (we just use the native Java expressions). Additionally, we get higher order types for free, in the shape of

Java objects. This greatly increases expressiveness and ease of use as was also described by Hedin et.al. [25]. Additionally, we allow access to some parser-specific variables by means of an API, to further increase the usability of our attribute system. The most important methods in our API can be found below:

- `attributes.put(attributeName : String, value : Object) : void`. This function is used to specify inherited context for a specific non-terminal. When called in the `@pre` of a non-terminal, will pass down given attribute-value pair to the specified non-terminal. This implements inherited context, which will be discussed in Section 4.1.
- `attribute(non-terminal : Integer, String: attributeName) : Object`. This function, when called on the 0^{th} non-terminal, will yield the the value for the attribute with the corresponding attribute name that was inherited from the parent non-terminal. When called with a value greater than 0 for its argument "non-terminal", will yield the value of the synthesized attribute with the specified name (provided by argument "attributeName") on the corresponding non-terminal. Note that this function is used for reading both inherited and synthesized attributes.
- `synthesize(attributeName: String, value : Object) : void`. This function is used to specify synthesized context for a specific non-terminal. When called, pass up the providid attribute-value pair to the parent non-terminal. This implements synthesized attributes, which will be discussed in more detail in Section 4.3.

There are some additional functions in our API that allow us to easily access parts of our input, and read values of input pointers used in the algorithm. These are particularity useful when one is concerned with the actual input that was parsed by a terminal.

- `currentInputPosition`, which yields the position of the input before we started processing the current non-terminal.
- `scanner.position()`, which yields the current position of the scanner.

The fact that we allow execution of arbitrary Java immediately allows us to use all Java control-flow constructs and expressions (e.g. if-else constructs, boolean expressions etc.). Moreover, instead of regular primitive types as attributes we can pass along pointers to objects, enriching the sorts of data we can pass (e.g. lists and sets) [25].

The general idea is to allow users to write pieces of Java code that use the API specified earlier to access attributes of the current non-terminal and set attributes for child and parent non-terminals. As mentioned before, there are a few points in a production at which is it sensible to prune:

- Before having parsed any literal. The block of Java code that is to be executed at this point should be annotated with `@pre(all):`' (as that code is executed before any literal is parsed).
- After each non-terminal. The block of java code that is to be executed after completion of the i^{th} terminal on the RHS of a production should by annotated with "`@pre(i)`" (where is is to be substituted).
- After having parsed every literal of the production (before performing the `pop()`). The block of java code that is to be executed at this point should be annotated with "`@post(all):`" (as this code is performed after we have parsed all literals).

Note that the points of disambiguation are presented at the level of a production. It is thus no more than logical that such a set of annotated blocks of Java code can be present for every production. Note that formally, we have assumed that every production can only contain a single alternate. Thus in a more realistic setting, every alternate should have such a set of annotated Java blocks.

Finally, to clearly separate BNF from annotated Java we require that all annotated Java code be put between tildes (∼) at the end of every alternate as can be seen in Listing 3.1. Note that we need these annotations to discriminate between the semicolons of Java and the semicolons of BNF. A BNF specification of the Attributed BNF scheme that allows for these types of annotations can be found in Listing 3.2.

```
S ::= A ∼ @pre(0):attributes.put("p", "1") ∼ |  B∼ @pre(0):attributes.put("p","2")
    ∼;
A ::= "a" ∼ @pre(all): if (attribute(0, "p").equals(1)) \{ deny; \}∼;
B ::= "a";
```

Listing 3.1: A production with annotated Java

```
GRAMMAR ::= RULE RULES | RULE;
RULE ::= LHS '::=' ALTERNATES;
ALTERNATES ::= ALTERNATE | ALTERNATE ALTERNATES;
ALTERNATE ::= LITERALSEQUENCE | LITERALSEQUENCE '~' CONTEXT '~';

LITERALSEQUNECE ::= LITERAL | LITERAL LITERALSEQUENCE;
LITERAL ::= TERMINAL | NONTERMINAL;

NONTERMINAL ::= '[a-zA-Z]*([0-9]*[a-zA-Z]*)*'
TERMINAL ::= '\'[^']*\'';

CONTEXT ::= CONTEXTDECL | CONTEXTDECL CONTEXT;
CONTEXTDECL ::= '@pre\(all\):[^@]*';
CONTEXTDECL ::= '@post\(all\):[^@]*';
CONTEXTDECL ::= '@pre\([1-9][0-9]*\):[^@]*';
CONTEXTDECL ::= '@pre\(0\):[^@]*';
```

Listing 3.2: attributed BNF specification in BNF (omitted layout)

3.4 Extending GLL

Now that we have seen where to disambiguate in theory, let us look at how to put this into practice. Recall the parser presented in Listing 2.15. We will show a number of code-fragments from this parser extended with the attribute system.

To show every position at which code can be injected, we abstract from a specific set of annotated Java code to. Rather we show the symbol Ω with a specific subscript at every location at which code may be injected . That is, $\Omega_{S0.pre(0)}$ means that at that position in the code, the context of the 1st alternate of production `S` annotated by `@pre(0)` should be inserted.

```
void S0() {
  $\Omega_{S0.pre(all)}$
  $\Omega_{S0.pre(0)}$
  push("S0_0");
  A();
}

void S0_1() {
  $\Omega_{S0.pre(1)}$
  push("S0_2");
  S();
}

void S0_2() {
  if (scanner.hasNext("d")) {
    $\Psi$
    scanner.next("d");
    $\Psi$
  } else { return; }
  $\Omega_{S0.post(all)}$
  pop();
}

void S1() {
  $\Omega_{S1.pre(all)}$
  if (scanner.hasNext("a")) {
    $\Psi$
    scanner.next("a");
  }
  else
  {
    return;
  }
  $\Omega_{S1.pre(0)}$
  push("S1_0");
  S();
}
void S1_1() {
  $\Omega_{S1.post(all)}$
  pop();
}

void S2() {
  $\Omega_{S2.pre(all)}$
  if (scanner.hasNext("")) {
    $\Psi$
    scanner.next("");
    $\Psi$
  }
  else
  {
    return;
  }
  $\Omega_{S2.post(all)}$
  pop();
}

void A0() {
  $\Omega_{A.pre(all)}$
  if (scanner.hasNext("a")) {
    $\Psi$
    scanner.next("a");
    $\Psi$
  }
  else
  {
    return;
  }
  $\Omega_{A.post(all)}$
  pop();
}
```

LISTING 3.3: A GLL parser that support attributes

3.5 Rejecting derivations

Now that we have seen where context is implemented, let us see how to actually reject derivations. To do so, let us see how derivations are rejected normally. A derivation in GLL without attributes is rejected whenever the next token that should be recognized, is not present in the input sequence. This check is performed by the *hasNext* function of the scanner. In Listing 3.3 we see that if this is the case, a return statement is executed. This halts further creation of derivation and thus stops the current derivation. For our disambiguation we will do the same thing. If at any point in our parse function, based on our attributes we conclude that we should reject the derivation, we perform a `return`. Note that the last moment in the algorithm at which we can reject a derivation is before the `pop()` or `push()` statements are executed. That is, if we were to allow a `push()` or `pop()` statement to be executed, new descriptors would already be created to continue the current derivation.

To illustrate, consider function `S1`() from Listing 3.3 in which $\Omega_{S1.pre(0)}$ is equal to the content of Listing 3.4 (i.e. the code is annotated with `@pre(0)`). The resulting code for `S1()` (after code injection) is presented in Listing 3.5. Note that the `deny` from the annotated Java code has been translated into a `return`. In this way, it mimics the behavior that would occur if the input did not meet the presented requirements. After the input has been deemed correct, the specified attributes are calculated to determine whether they lead to inconsistencies. If this is not the case, an identical course of action is taken, as if the input was incorrect. In this way, we extend GLL with a mechanisms for pruning derivations in a way that is as close to the mechanisms already in use.

```
if (attribute(0, "p").equal(true)) {
  return;
}
```

Listing 3.4: Java code that uses the attribute system API

```
void S1() {
  if (scanner.hasNext("a")) {
    scanner.next("a");
  } else {
    return;
  }

  if (attribute(0, "o").equal(true)) {
    return;
  }

  push("S1_0");
  S();
}
```

Listing 3.5: Part of a parser in which annotated Java code has been injected

Chapter 4

Implementing attributes in GLL

Now that we have a way of assigning and modifying attributes, let us find out how to actually implement them. In this chapter, we will look at the different types of attributes and how to implement them in GLL.

4.1 Implementing inherited attributes

4.1.1 Classic LLRD

In classic LLRD, often the stack would be used to propagate context (attributes) from one parse function to the other. Simply passing the context for a production as parameters to its parse function is very simple, but a clean solution to the challenge. However, in GLL this is a more challenging problem, because of a number of factors:

- Multiple functions represent a single production, in contrast to the single function per production in LLRD. Each of these functions should receive the identical context;
- GLL allows for multiple derivations, hence we have parallel stacks, each with a potentially different context;
- The GSS is used, which allows for the sharing of stacks. This further increases complexity.

In this chapter we will explore the ways to overcome these difficulties and implement attributes into GLL parsers.

4.1.2 Different approaches

Possible solutions for enabling parse functions to work with attributes may come along three reasonable lines of thought.

1. Emulate the behavior of a classical call-stack onto the GSS;
2. Pass attributes along via descriptors;
3. Store all contextual information in a globally accessible location.

Note that extending the SPPF is not viable, as we require a top-down mechanism for inherited context, where the SPPF is built bottom-

Passing attributes along via descriptors is not a feasible solution. To illustrate: given a production N $::=\beta_1\beta_2$ (where β_1 and $\downarrow_2$ are GLL Blocks). The attributes passed to N should also be available to β_1 and β_2. For β_1 this is not so much an issue as it is invoked directly by the descriptor for N. However, β_2 poses a challenge. Recall that β_2 is invoked by the last recursive GLL Block of the non-terminal in β_1 (say β_r). If we were to pass attributes along via descriptors, and β_2 is invoked by β_r, then those attributes need to be recursively passed along from β_1 all the way to β_r. This recursive passing of attributes involves an large amount of copying and thus overhead, which we wish to avoid.

A solution to this problem would be to make this context immutable and either store it globally or just pass around pointers. This would be a better solution, however we would have to create another secondary data structure that would further complicate the already complex interactions in GLL. We are looking for a more natural way to incorporate attributes into the algorithm.

An attempt to emulate workings of a normal stack onto the GSS seems to be the most logical variant to explore. Passing attributes along on stacks is something that is well known for classic LLRD variants. Additionally, GSS states are already immutable thus we have the same advantages as globally stored attributes (not having to copy attributes all the time). Thus, the choice is made to augment GLL by emulating LLRD attribute grammar behavior (with respect to the call-stack) onto the GSS. Note that we cannot use the SPPF for inherited context, as the SPPF is constructed bottom-up, and inherited context works top-down.

4.1.3 Extension of the GSS

The approach we will take is to attempt to emulate the behavior of an LLRD stack, with respect to context, onto the graph structured stack used in GLL. But before considering the GLL approach, lets us first look at an example from LLRD. Since LLRD uses the call-stack, attributes can be passed along to parse functions as parameters to these parse functions. When a parse function recursively invokes another parse function, the inherited attributes that are relevant in that parse function will be stored onto the call-stack to be restored when we return to this function. To fully understand, we will look into this process in more detail. Say we are in a production of the shape X $::=\alpha$ Y β. Say the stack after processing α is Σ. Consider the sequence presented in Figure 4.1

a The situation after completing X_α (i.e. α in X).

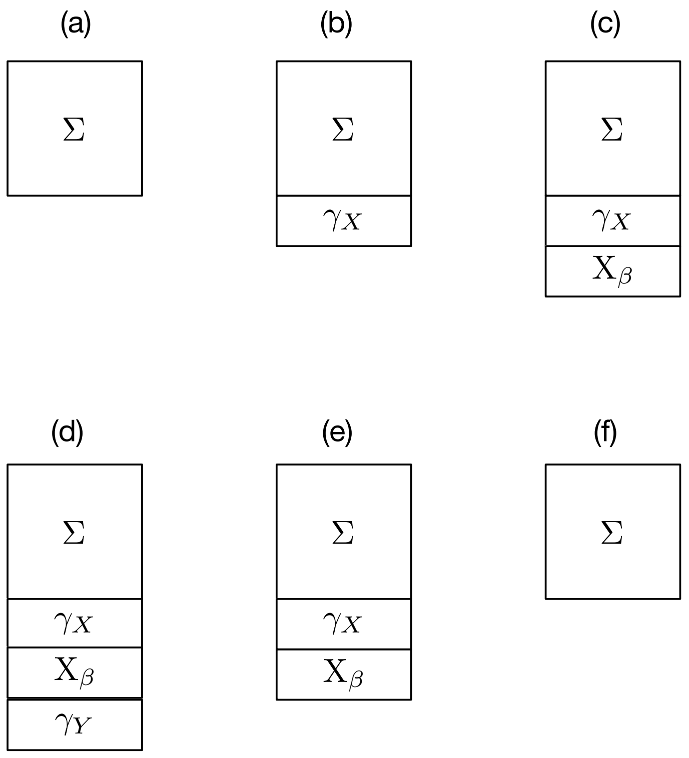


FIGURE 4.1: Visualization of the call-stack for method invocation in an LLRD attribute parser

b We store our current context, γ_X, onto the stack, to be restored after returning from Y. These are the attributes that `X` has been annotated with and that should be available throughout `X`.

c We store information on where to continue after finishing `Y` on the stack. In our case, $\texttt{X}_\beta$ (that is, β in `X`).

d We push the function parameters (context) of `Y` onto the stack (γ_Y).

e We invoke `Y`, which consumes its function parameters from the stack.

f After `Y` completes its execution (possibly containing recursive invocations of non-terminals), the return location and previous context is retrieved from the stack and restored. We can now continue with β.

Now that we have seen the workings of inherited attributes in a classic LLRD setting, let us look at the general idea for GLL. The sequence is similar to that in Figure 4.1, but now with a GSS instead of a stack. The sequence has been explained below and is illustrated in Figure 4.2. One important thing to notice is that our stacks are persistent. That is, the stack state is preserved when popping, our descriptors just reference a different stack, and the old stack head will remain in existence for future use. As a consequence of our stack states being persistent, we need not continuously write back context to the stack after retrieving it.

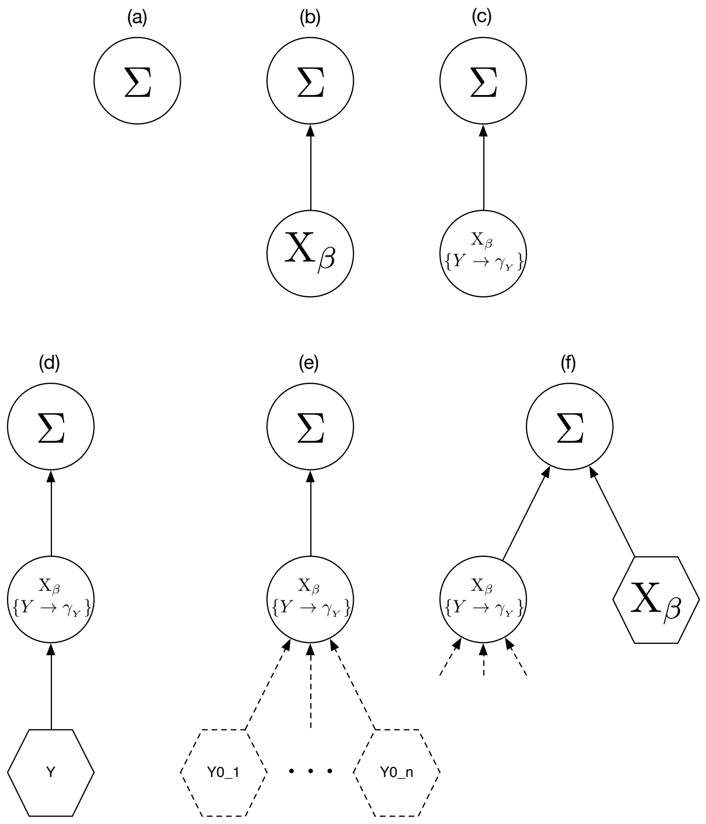


FIGURE 4.2: Visualization of the GSS for method invocation in an GLL attribute parser. Processed descriptors are visualized by dotted outlines.

a Again the situation after completing $\mathtt{X}_\alpha$.

b Before scheduling `Y`, we push the next GLL Block (say $\mathtt{X}_\beta$) onto the stack, lets call the stack descriptor needed for this δ_β.

c We now push the context for Y (γ_Y) onto the stack. Note that our stack is actually in the shape of a stack descriptor (a GSS node). Hence we store our context at that node.

d We schedule `Y` on top of δ_β.

e Every GLL Block of `Y` executes, possibly with recursive non-terminals (which have been committed for clarity). Note that all of the GLL Blocks of `Y` have access to the context of `Y` through the stack descriptor δ_β. This statement is discussed more thoroughly in Section 4.2.3.

f Lastly, we continue with β of `X` ($\mathtt{X}_\beta$).

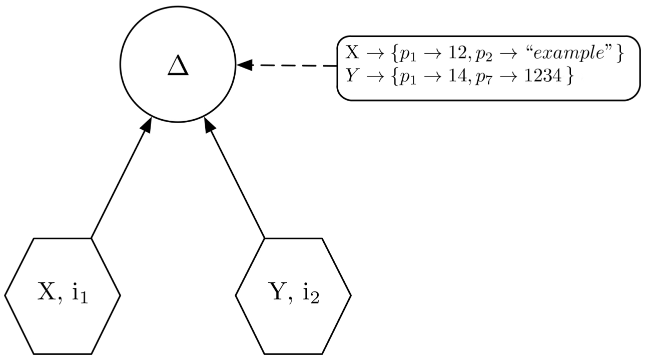

Δ
X → {p_1 → 12, p_2 → "*example*" }
Y → {p_1 → 14, p_7 → 1234 }
X, i_1
Y, i_2

FIGURE 4.3: An example of a context environment

4.2 Issues with this approach

Unfortunately, this approach will not work in all cases. We need a slight modification of GLL data structures in order to cover all cases. We will discuss the issues that arise, and their solutions in the coming sections.

4.2.1 Attribute collisions

The first problem we encounter has to do with the sharing of stack heads between multiple non-terminals with equivalent property names.

Context, as used in a particular production, is not restricted in the use of attribute names. It can be the case that two alternates are annotated with different values for the same properties. In such a case one might overwrite an existing value, thus causing an alternate to operate on the wrong attributes. Consider, for example, partial grammar Γ_{13}. When storing the specified attributes for `X` and `Y` in the node for `S`, property `p1` would be specified twice (causing an overwrite). To this extent, each instance of a non-terminal working on a particular stack-head will receive its own set of variables. We call such a collection of sets of variables a *Context Environment*. An illustration of such an environment added to a GSS node can be found in Figure 4.3.

$$(\Gamma_{13}) \qquad \begin{array}{lll} \texttt{S} ::= & \texttt{X} & \sim \texttt{@pre(0)} : \texttt{attributes.put}(\text{“}\texttt{p1}\text{”}, 12) \sim; \\ \texttt{S} ::= & \texttt{Y} & \sim \texttt{@pre(0)} : \texttt{attributes.put}(\text{“}\texttt{p1}\text{”}, 14) \sim; \end{array}$$

Note that by the labeling scheme in GLL, each alternate is assigned a unique name. This fact, in addition to the fact that two instances of the same alternate are never scheduled on the same stack-head, means attribute collisions will no longer occur.

4.2.2 Stack sharing

The next issue is also caused by stack sharing. One of the main assumptions in context-free grammars, as mentioned before, is that we may substitute a non-terminal by every production with that non-terminal as its LHS. Thus, the GSS will share identical stack heads, not stacks. That is, if we have two stacks $\Sigma_1 = \alpha\texttt{X}$ and $\Sigma_2 = \beta\texttt{X}$, in the GSS they will both be presented by the stack head `X`. Observe that the context for both these stack heads might be different, based on the attributes passed in α and β. To illustrate consider grammar Γ_{14}. This attribute grammar expressed that the non-terminal `A` may never derive a `D`, and that `B` may never derive a `C`. Consider now the workings of the GSS, illustrated in Figure 4.4. For completeness, note that all GSS nodes in this figure have an input position of 0.

$$(\Gamma_{14}) \qquad \begin{aligned} \texttt{S} &::= \texttt{A};\\ \texttt{S} &::= \texttt{B};\\ \texttt{A} &::= \texttt{E} \sim \texttt{@pre(0)} : \texttt{attributes.put}(\text{“p”}, \text{“a”}); \sim;\\ \texttt{B} &::= \texttt{E} \sim \texttt{@pre(0)} : \texttt{attributes.put}(\text{“p”}, \text{“b”}); \sim;\\ \texttt{E} &::= \texttt{C} \sim \texttt{@pre(all)} : \texttt{if}(\texttt{attribute}(0, \text{“p”}).\texttt{equals}(\text{“b”}))\{\texttt{deny};\} \sim;\\ \texttt{E} &::= \texttt{D} \sim \texttt{@pre(all)} : \texttt{if}(\texttt{attribute}(0, \text{“p”}).\texttt{equals}(\text{“a”}))\{\texttt{deny};\} \sim;\\ \texttt{C} &::= \text{“1”};\\ \texttt{D} &::= \text{“1”}; \end{aligned}$$

a Non-terminal `A` has been substituted for both its alternates `A` and `B`.

b E can be substituted for `A`, hence `E` is pushed onto the stack of `A` and the attributes specified are pushed (i.e. property `p` is set to “a”).

c `E` can be substituted for `B`. The GSS realizes it can share the stack-head `E` created by `A`. It then pushes property `p` is set to `b`, which overwrites `p` having been set to `a`. This is clearly an issue as the information pushed by `A` is lost.

The solution is based on the observation that the stacks $\Sigma_1 = [S; A; E]$ and $\Sigma_2 = [S; B; E]$ are different (even though their stack heads are the same), and can thus not share their stack heads. We thus propose the following modification: Let two GSS nodes be equal if and only if their labels, input positions, and context environments are equal (rather than just their labels and input positions). By updating GSS Node equality in this way, we obtain the modified working illustrated in Figure 4.5 and described below:

a Non-terminal `A` has been substituted for both its alternates `A` and `B`.

b E can be substituted for `A`, hence `E` is pushed onto the stack of `A` and the attributes specified are pushed (i.e. `p` is set to `a`)

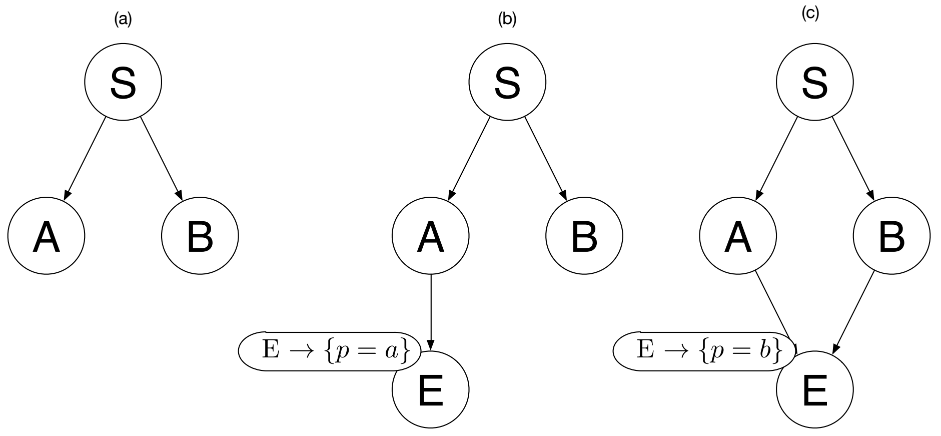


FIGURE 4.4: Partial working of the GSS for grammar Γ_{14} on input sequence "1"

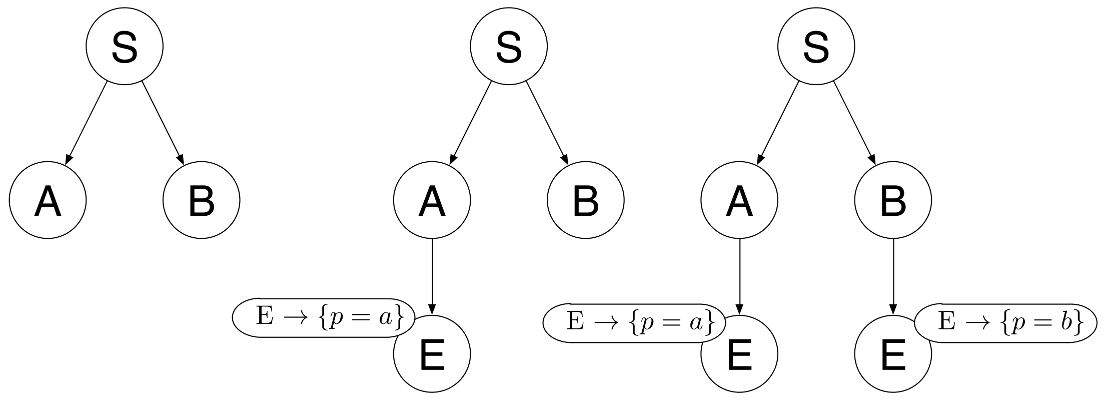


FIGURE 4.5: Updated working of the GSS for grammar Γ_{14} on input sequence "1"

c `E` can be substituted for `B`. The GSS might be able to share the stack-head `E` created by `A`. However, $E \rightarrow \{p = b\} \neq E \rightarrow \{p = a\}$ thus the stack head is not re-usable by the current stack. A new stack is created with symbol `E`' and the context environment is pushed to that stack.

The only thing that remains is to properly define the equality of two environments.
Definition:
Context environment e_1 is equal to environment e_2 if and only if:

$$\forall_{x \in e_1} [\forall_{v \in x} [\exists_{y \in e_2} [\exists_{w \in y} [x = y \wedge v = w]]]] \wedge \forall_{x \in e_2} [\forall_{v \in x} [\exists_{y \in e_1} [\exists_{w \in y} [x = y \wedge v = w]]]]$$

Where x, y are sets of attributes(with values) and v, w are attributes inside these sets. Put more informally, every set of properties for every non-terminal in e_1 should also be present in e_2 (with equal values) and vice-versa.

The elegance of this solution is that it ensure that stacks have full freedom to diverge (as is required based on context), but that they may only converge (share) if they have the same context. Thus sharing is not eliminated, it is only reduced. Note that the operation of GLL is not harmed, as stack splitting is nothing new to GLL, we just increase the frequency with which it occurs.

The one disadvantage that this approach entails is that is allows context to destroy the left-recursion handling mechanisms in GLL. Recall the left-recursion mechanisms presented in section 2.9. Now imagine a left-recursive production, that inherits an incremental attribute to the left-recursive non-terminal. For example:

$$\texttt{E} ::= \texttt{E} \quad \text{“x”} \sim \texttt{@pre(0)} : \texttt{attributes.put(“p”, attribute(0, “p”) + 1);} \sim$$

Now, whenever the descriptor for `E` with $\texttt{p} = \texttt{0}$ scheduled a recursive `E` it will annotate that `E` with $\texttt{p} = \texttt{1}$ (say $\texttt{E}_{\texttt{p=1}}$). Next, $\texttt{E}_{\texttt{p=1}}$ recursively schedules $\texttt{E}_{\texttt{p=2}}$ (which is different than $\texttt{E}_{\texttt{p=1}}$). This process continues, and as each next invocation of `E` is annotated with a different value. As all these GSS nodes are pairwise unequal, the left-recursion mechanisms in GLL will not work. However, this is an artificial example.

4.2.3 Availability of attributes

Now that we have updated the stack, it is crucial we ensure that descriptors on our stack heads are still able to access the correct attributes. Consider the reasoning below, showing that descriptors indeed have access for all attributes they should have access to.

Lemma:
All executions of functions $\texttt{P}_\texttt{i}$ (i.e. the GLL Blocks) for a particular production $\texttt{P} ::= \alpha$, have access to a single GSS node. (This is excluding recursive schedules for other productions). This single GSS node can then be used to store the context that all these $\texttt{P}_\texttt{i}$ should have access to (i.e. the inherited context for `P`).
Proof:
Given a function that is about to schedule our production `P`, with a current GSS node Δ and input position i. The descriptor $\langle P, \Delta, i\rangle$ is created to encode this fact. We will now prove that Δ is the GSS node accessible to all executions of $\texttt{P}_\texttt{i}$. There are two cases:

- $\texttt{P} \equiv \texttt{P0}$ (where `P0` is a GLL Block)
  In this case, `P` will schedule $\langle \texttt{P0}, \Delta, i\rangle$. Execution of this descriptor trivially implies that the execution of the `P0` with respect to this `P` has access to Δ. Note that there are no further executions of functions for `P`, as there are none and thus `P0` will perform a pop.

- $\texttt{P} \equiv \texttt{P0} \cdots \texttt{P0\_n}$ (where $\texttt{P0}, ..., \texttt{P0\_n}$ are GLL Blocks)
  In this case, `P` will first schedule `P0`. As in the previous case, we see that `P0` has access to Δ. Next, as there are multiple GLL Blocks, `P0` must end with a non-terminal, say `X`. `P0` will create a GSS node $\Delta_X = (P0\_0, \Delta, i)$ and schedule $\langle X, \Delta_X, j\rangle$. Subsequently, work for `X` will be done. After all (recursive) work has finished, a pop will be invoked targeted to Δ_x. This pop will create (at least) a descriptor $\langle P0\_0, \Delta, k\rangle$. And thus also the subsequent function of P, $P0\_0$, has access to Δ. This reasoning can be repeated until the function for the final GLL Block, which does not end in a non-terminal. However, as this is the last function,

and it has access to Δ, all functions regarding the processing of `P` have had access to Δ. □

Taking this lemma into account, it is only logical that the ideal place to store the context of `P`, is Δ. Combining this with the context environment to avoid attribute-name collisions between non-terminals and the new GSS node equivalency definition ensures that our stack indeed works.

4.2.4 Correctness argument

A major issue with the correctness of any disambiguation comes from the sharing. If a derivation is rejected, no derivations should be lost implicitly. For our attribute system, this is not the case. We will now present the reasoning why. In Section 4.2.3 we reasoned that descriptors always have access to the correct context. Any possible collisions of context is handled by restricting stack merges as described in Section 4.2.2. Note that these splits are necessary, as we are making local decisions. Not splitting the stacks may lead to conflicts further down the road.
Assume now, towards a contradiction that a derivation was rejected (based on attributes) that should not have been rejected. This means that a descriptor that was working on this derivation was pruned (i.e. did not produce new descriptors). However, if a derivation was pruned, then it must be the case that there is a violation in its attributes. If there is a violation in the attributes of a derivation, then the derivation is (by definition) incorrect. This is a contradiction, hence it cannot be the case that a derivation is removed that should not have been removed. □

A second important part is that we prune derivations, not remove them. That is, we may not allow derivation trees to come into existence and then remove them. We must prevent creation, in order to maintain consistency in the internal data structures. There are two possibilities to consider:

- The sub-tree already exists. A correct derivation has already constructed a sub-tree, however our current derivation is incorrect. The incorrect derivation that shares the sub-tree will simply not continue its derivation (by not creating new descriptors), but will not remove the existing derivation.
- The sub-tree does not exist. Our current derivation may not derive the sub-tree in question, but there might be a correct derivation that may derive the sub-tree. Note that the current (incorrect) derivation may create partial sub-trees. Note, as it is an incorrect derivation, this derivation will not yield an SPPF containing this partial sub-tree. However, any subsequent correct derivation may re-use the created sub-tree (as we do not remove it). Hence, if there is a correct derivation that uses the tree (even though an incorrect derivation was there before), the sub-tree will be constructed correctly.

We conclude that in both cases the sub-tree will be constructed correctly. Finally, note that once a derivation has been created, they cannot be rejected. At time of creation, if there were no attribute conflicts, no future derivations can invalidate this fact. Correctness of this statement follows directly from the depth-first left-to-right nature of our attribute evaluation. As a consequence, it is evident that this approach is not capable of eliminating all ambiguities, only ones that are locally detectable.

4.3 Implementing synthesized attributes

Just as we support inherited attributes, we also want to support synthesized attributes. Synthesized attributes allow us to pass context upwards (i.e to parent non-terminals). Recall the example in Section 1.5 related to grammar Γ_6 and Γ_7. In this example, we used synthesized attributes to restrict two instances of the non-terminal `X` from deriving the same terminal symbol. This has been expressed using the syntax introduced in Chapter 3 in grammar Γ_{15}.

(Γ_{15}) **S** ::= **X** **X** ∼
`@post(all)` : `if(attribute(1,` “`type`”`).equals(attribute(2,` “`type`”`)){return;}`
∼;
X ::= “**b**” ∼
`@post(all)` : `synthesize.put(`“`type`”, “`b`”`)` ∼;
X ::= “**c**” ∼
`@post(all)` : `synthesize.put(`“`type`”, “`c`”`)` ∼;

In an LLRD setting, a viable approach is to pass synthesized attributes along as return values of our parse functions. However, in GLL this is not feasible, as all functions operate under the main loop. Again we require an alternative solution. A solution could again come in three different forms:

1. Extend the modifications previously made to the GSS to support synthesized attributes;
2. Pass the synthesized context upwards using descriptors;
3. Use the SPPF to store the context, in a similar way as we store it in the GSS.

We will first explore the approach that further augments the GSS, but conclude that this approach is not feasible in all cases. We then present an SPPF-oriented solution, that does provides synthesized attributes in a correct fashion. One thing to recall before starting this chapter is the `synthesize` API presented in Section 3.3. This API will be used to pass attributes upwards (synthesize) inside our annotated Java.

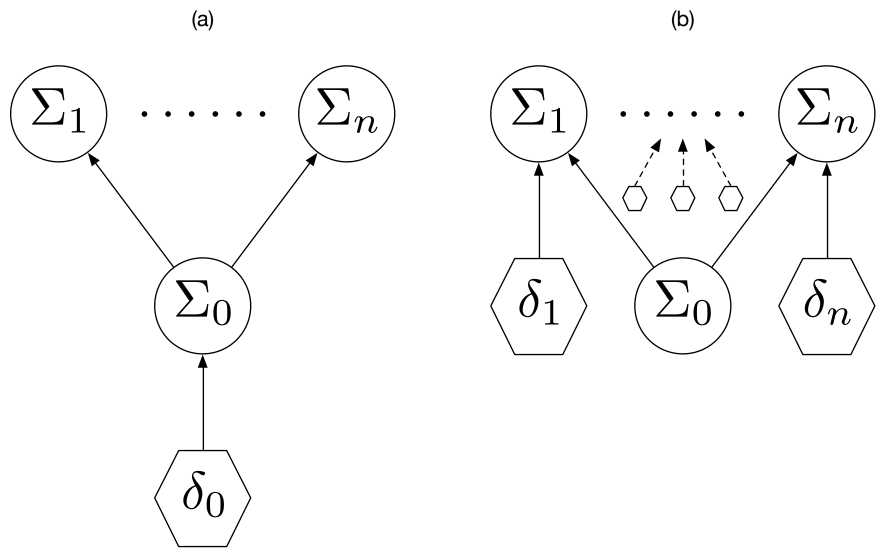


FIGURE 4.6: Illustration of synthesized context

4.3.1 GSS Approach

The GSS approach uses the context environments of augmented GSS to store synthesized attributes. The main idea is, just as in an LLRD call-stack, to traverse back the path on the stack, and annotate upwards. To understand the GSS approach, consider the following pattern that occurs during a `synthesize()` call. Say we are processing a descriptor (δ_0) that operates under stack-head $\Sigma_0 = (L_0, I_0)$ as illustrated in Figure 4.6 (a). Σ_0 has a number of parenting GSS Nodes, Σ_1 through Σ_n. Descriptor δ_0 is one of the descriptors responsible for parsing a specific alternate α_0. Now say either δ_0 or one of its successors for α_0 performs a `pop()` (marking that the entire alternate has been processed). Descriptors will be created for every parent of Σ_0 as illustrated in Figure 4.6 (b). That is, we have finished the non-terminal (say `X`) that is specified by the GLL Block preceding GLL Block L_0. So, δ_1 through δ_n continue to work on the alternate (α_1) that previously invoked `X`. Thus, they need access to the synthesized attributes from $\mathcal{X}$. As was the case for the inherited context, we make these available to them through their parenting GSS Node. The core observation to make is that when synthesizing a property, it suffices to provide all parents of the current GSS node with the property (i.e. we go up two GSS levels).

For a more complicated example, consider grammar Γ_{16} and input sequence $\sigma = ab$. The entire sequence has been illustrated in Figure 4.7. Note that input positions have been omitted in the illustration for the sake of readability. When processing, after executing the aprse function for our first non-terminal, the pattern illustrated in Figure 4.6 emerges. Now $\delta_0 = \langle A0, 0\rangle$, $\Sigma_0 = (X0\_0, 0)$ with only one parent $\Sigma_1 = \bot$. This situation is illustrated in (a) of Figure 4.7. $A0$ processes the terminal symbol “a” and synthesizes property ϕ up two levels as was explained. Next, the function for the next GLL Block ($\mathcal{A}'\_'$) is scheduled, as can be seen in (b) of Figure 4.7. Lastly, after $\mathcal{A}'\_'$

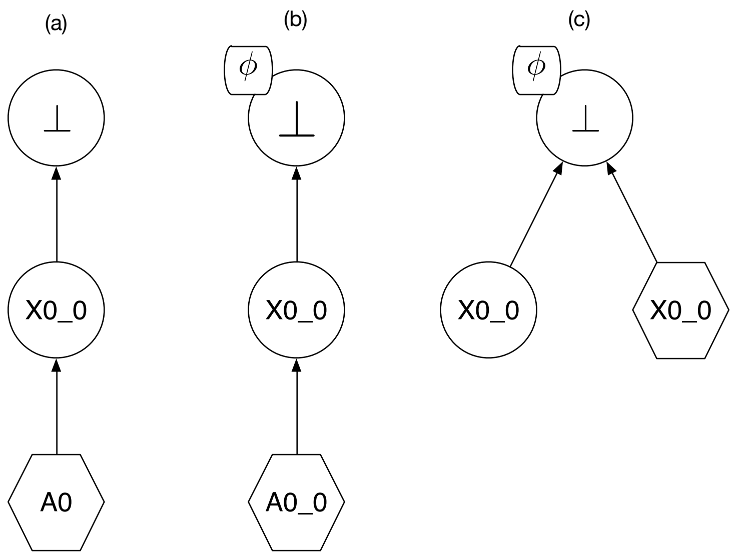


FIGURE 4.7: An instantiated example of attribute synthesis

has performed a `pop()`, and work continues on $\mathcal{X}$, we see that $\mathcal{X}$ ′_′ has access to ϕ in the same way as it has access to its inherited attributes.

$$(\Gamma_{16}) \qquad \texttt{X} ::= \texttt{A} \quad \text{“b”;}$$
$$\texttt{A} ::= \text{“a”} \sim \texttt{synthesize}(\text{“v”}, \text{“a”}); \sim;$$

Now that we have seen where to store synthesize attributes, let us look at how to store them. As explained in Section 4.2.1, we might have collisions in our attribute names. This problem replicates itself for synthesized attributes. For example, there is an inherited attribute named “p” and a synthesized attribute named “p”. To resolve this, we will put synthesized attributes in separate context environments. To be precise, we construct an environment per non-terminal in the RHS of a production. Rather than storing them by non-terminal (as we did for inherited attributes), they are stored by label (i.e. the label of their GLL Block). this is necessary, as a single RHS might have multiple non-terminals with the same name. Using GLL Block names for storage is ideal, as there is precisely one non-terminal per Block and we are able to address them in a concise way (as defined in Section 2.3).

We address synthesized attributes using the same function *attribute* we described in the chapter on inherited attributes. Using the descriptor label, we can reconstruct the label of the GSS. Let `L` be the label of the descriptor we are currently processing. Either $\texttt{L} \equiv \texttt{P} +\!\!+ \texttt{A} +\!\!+ \text{“\_”} +\!\!+ \texttt{B}$ or $\texttt{L} \equiv \texttt{P} +\!\!+ \texttt{A}$. Here `A` represents the number of the alternate of `P`, and `B` the GLL Block of `A`, as defined in Section 2.3. The function works by extracting the labels of the production (`P`) and alternate `A` and combining that with information on the current GLL Block (by means of the current descriptor) to re-create labels of the different context environments (e.g. `P+A` or `P+A+"_"+(nonTerminalNumber-1)`).

```
void attribute(int nonTerminalNumber, String attributeName) {
  Let P, A be the values described before.
  if (nonTerminalNumber == 0) {
    /*case we want to get inherited attributes, we only need the production (P), as
  the attributes are for that entire production*/
    return currentGSSNode.getContext().get(P).get(attributeName);
  } else if (nonTerminalNumber == 1) {
    /*case we want to get content form the first GLL Block, we just need the
  alternate (A) of the production (P)*/
    return currentGSSNode.getContext().get(P+A).get(attributeName);
  } else {
    /*case we want to get content from a specific GLL Block, we need the production
  (P), alternate (A), and Block number (non-terminal -1)
    -1 becase the 2nd non-terminal is in Block PA_1*/
    return currentGSSNode.getContext().get(P+A+"_"+nonTerminalNumber-1).get(
  attributeName);
  }
}
```

LISTING 4.1: Augmented attribute fetch function for synthesized context

Now, using this function we are able to address attributes that correspond to a given non-terminal, by the index of that non-terminal in the production. For instance, take production `F` ::= `Q R`. We are able to use:

- attribute(0, p) to fetch the value of inherited attribute `p`, as these are attached to the LHS non-terminal (`F`). Which is the 0^{th} non-terminal in our production.
- attibute(1,p) to fetch the value of attribute `p` that was synthesized by `Q` (as this is the 1^{st} non-terminal in the production)
- attibute(2,p) to fetch the value of attribute `p` that was synthesized by `R` (as this is the 2^{nd} non-terminal in the production)

This pattern evidently extends to productions with more than 2 RHS non-terminals.

4.3.2 Issues with the GSS approach

The first downside to the GSS approach is that we can only pass attributes back up along existing trees. We cannot create new trees, as this would require replication of trees. To illustrate, consider two descriptors δ_1 and δ_2, both operating on the same stack-head ($\Sigma = (L_0, I_0)$). These descriptors correspond to two alternates of a production that are both applicable (e.g. “abc” and “[a-c]+”). Say δ_1 synthesized property $p = 1$ and δ_2 synthesized `p` = 2. Depending on the order of execution of these descriptors, a different value will be available to descriptor δ_{L_0}. The behavior that we want is that we synthesize two different values, and that this leads to two different subtrees. This thought leads us to re-consider the SPPF approach for synthesized attributes.

The second issue concerns itself with the sharing properties in a normal SPPF. Start by noting that the SPPF is a data structure used for representing parse forests. As the majority of the parsing world focuses on context-free parsing, it should come to no surprise, that some assumptions with respect to context-free grammars are present in the SPPF. This could, and indeed is, an issue in the case of attribute grammars, as these languages form a super-class of the class of context-free grammars. The SPPF is a data structure that has sharing as one of its main priorities. This is usually a good thing, however for representing context-sensitive parse forests, this can be a problem. To illustrate, please consider Grammar Γ_{17}.

$$(\Gamma_{17}) \qquad \begin{aligned} \texttt{S} &::= \texttt{A} \quad | \quad \texttt{B}; \\ \texttt{A} &::= \texttt{E}; \\ \texttt{B} &::= \texttt{E}; \\ \texttt{E} &::= \texttt{C}|\texttt{D}; \\ \texttt{C} &::= "\texttt{1}"; \\ \texttt{D} &::= "\texttt{1}"; \end{aligned}$$

These the four possible parse trees for the input sequence `"1"` shown in Figure 4.8. Next, consider the SPPF for these parse trees in Figure 4.9. The SPPF shares symbols with equal non-terminal symbol, left extent, and right extent. This is logical, taking the assumption of a context-free grammar into account. That is, parsing of a non-terminal is not influenced by the history (context) of a derivation. For a context-sensitive grammar (and thus also for an attribute grammar), this is obviously not the case. As per definition of context-sensitive grammar, the context of a parse can influence the resulting derivation. This observation is a strong indicator that the SPPF approach (mentioned in the beginning of this chapter) is the only viable approach for solving this problem.

For example, say we want to restrict derivations containing an `A`, may not use the alternate `E` ::= `D` and productions containing a `B` may not use the alternate `E` ::= `C`. This results in the two parse trees in Figure 4.10. However, classical SPPF sharing would still yield the same SPPF in Figure 4.9. To solve this, we pose a very obvious solution: only share symbols that have identical context. As we assumed determinism of our declared context, two identical symbols (at the same position) with identical context must have identical derivations.

4.3.3 SPPF Approach

As we have concluded, the GSS approach is not suitable for solving the problem at hand, we consider augmenting the SPPF to incorporate synthesized attributes. The main idea behind the SPPF approach is to augment SPPF nodes in the same way that we augmented GSS nodes: we add attributes and require equality of these attributes for equality of SPPF nodes. With this extension, GLL automatically creates new SPPF trees for derivations that have different synthesized attributes (analogous to the GSS).

To understand this process better, first consider the normal workings of the SPPF presented in Figures 4.11 and 4.12. Figure 4.11 shows how, when a `pop()` is performed,

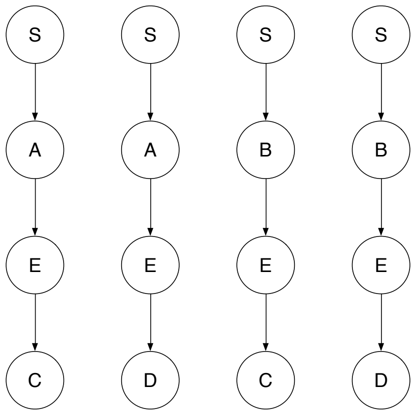


FIGURE 4.8: All four possible parse trees for input sequence "1" with respect to Grammar Γ_{17}
.

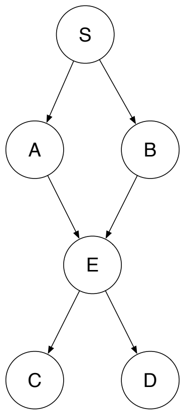


FIGURE 4.9: SPPF for input sequence "1" with respect to Grammar Γ_{17}
.

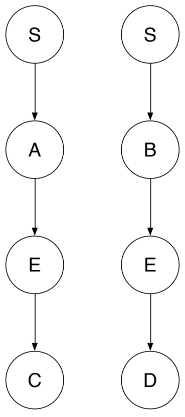


FIGURE 4.10: Disambiguated parse trees for input sequence "1" with respect to Grammar Γ_{17}
.

the partial SPPF is retrieved from the stack (a), and combined with the current branch using a packed and a symbol node (b). Figure 4.12(a) illustrates an ambiguous branch. Another derivation for `X` has been found, namely via `X` ::= `Z`, rather than the existing `X` ::= `Y`. This extra derivation is then attached to the existing packed node as shown in Figure 4.12(b).

The extension we propose, is to keep a variable *synthesize* (which is a context environment as described in Section 4.2.1), to which users may append custom attributes using our API. When actual synthesis takes place, the attributes will be attached to the corresponding SPPF node using a naming scheme similar to that in GSS environments (i.e. using GLL Blocks to address non-terminals). The advantage of this approach is that:

- For different alternates that derive the same non-terminal, normally their top-level symbol nodes would be equal;
- As they now have different context they will not be shared (as their SPPF nodes are now unequal);
- Corresponding descriptors will be created automatically, as SPPF data for the different descriptors are no longer equal. Thus, we obtain multiple derivations for free.

The issue that remains is in combining ambiguous derivations of the same alternate. This has been abstractly illustrated in Figure 4.13(a). Normally, we would attach a second derivation to the corresponding packed node. However, as the alternates for the derivation are the same, by default the GLL algorithm will ignore the second derivation. We thus extend the algorithm (more specifically the `getNodeP` method) to check the context of the children of the packed node. If there is no child present with the same context, we add it to the packed node. This has been illustrated in 4.13(b).

For a more instantiated example, consider grammar Γ_{18} and input sequence "aaa" (ignore context). The SPPF for this sequence can be found in Figure 4.14. Say now we want to synthesize which terminal we processed in `A`, and deny branches in which the first `A` is "aa", we can encode this using attributes as presented in Γ_{19}. Not considering the rejection of rules, an annotated SPPF is yielded as illustrated in Figure 4.15. The instance of `X` can then reject a branch that does not meet the requirements, yielding the disamibugated parse forest illustrated in Figure 4.16.

(Γ_{18})

$$\begin{aligned} \texttt{S} &::= \texttt{X};\\ \texttt{X} &::= \texttt{A}\quad\texttt{A};\\ \texttt{A} &::= \text{"a"};\\ \texttt{A} &::= \text{"aa"}; \end{aligned}$$

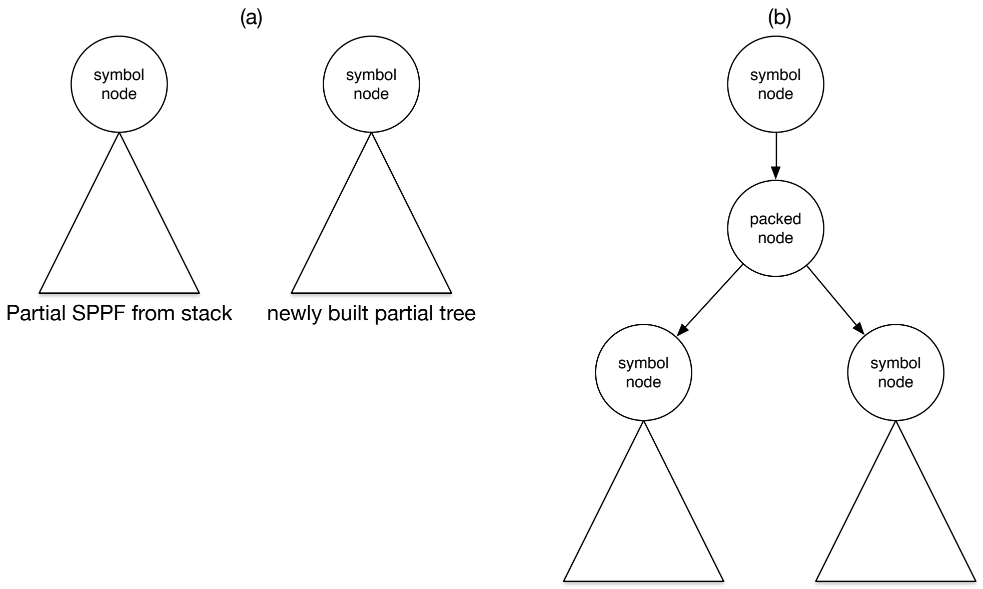


Figure 4.11: Normal workings of the SPPF

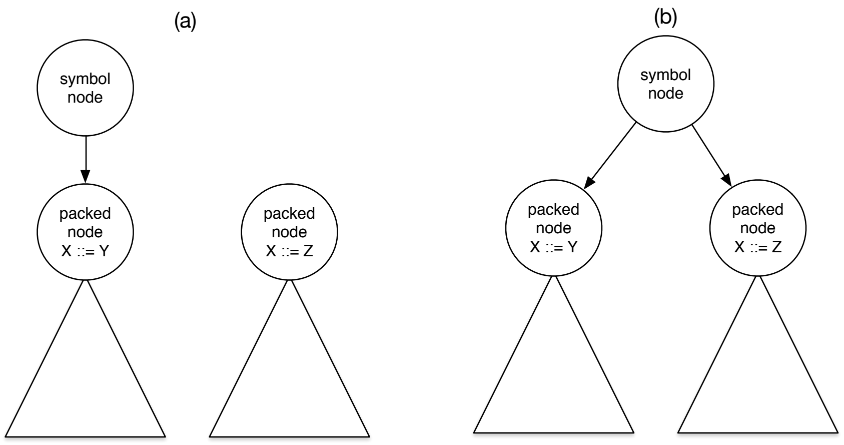


Figure 4.12: Normal handling of ambiguity in the SPPF

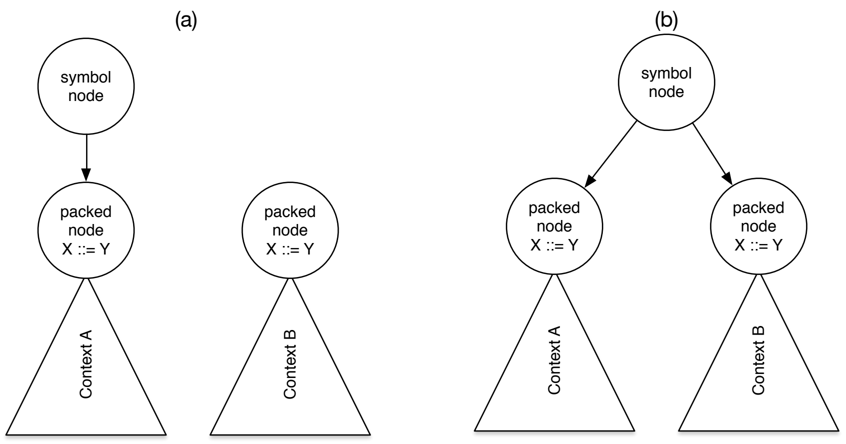


Figure 4.13: Augmented handling of ambiguity in the SPPF

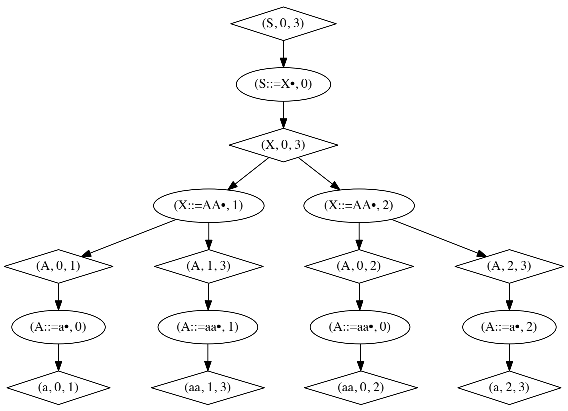


FIGURE 4.14: The original SPPF for sequence "aaa" with respect to Γ_{18}

$$(\Gamma_{19}) \qquad \begin{aligned}
&\texttt{S} ::= \texttt{X};\\
&\texttt{X} ::= \texttt{A} \quad \texttt{A} \sim\\
&\texttt{@post(all)} : \texttt{if(attribute(1, ``type'').equals(2))\{deny; \}}\\
&\sim;\\
&\texttt{A} ::= \text{``a''} \sim\\
&\texttt{@post(all)} : \texttt{synthesize.put(''type'', 1'');}\\
&\sim;\\
&\texttt{A} ::= \text{``aa''} \sim\\
&\texttt{@post(all)} : \texttt{synthesize.put(''type'', 2'');}\\
&\sim;
\end{aligned}$$

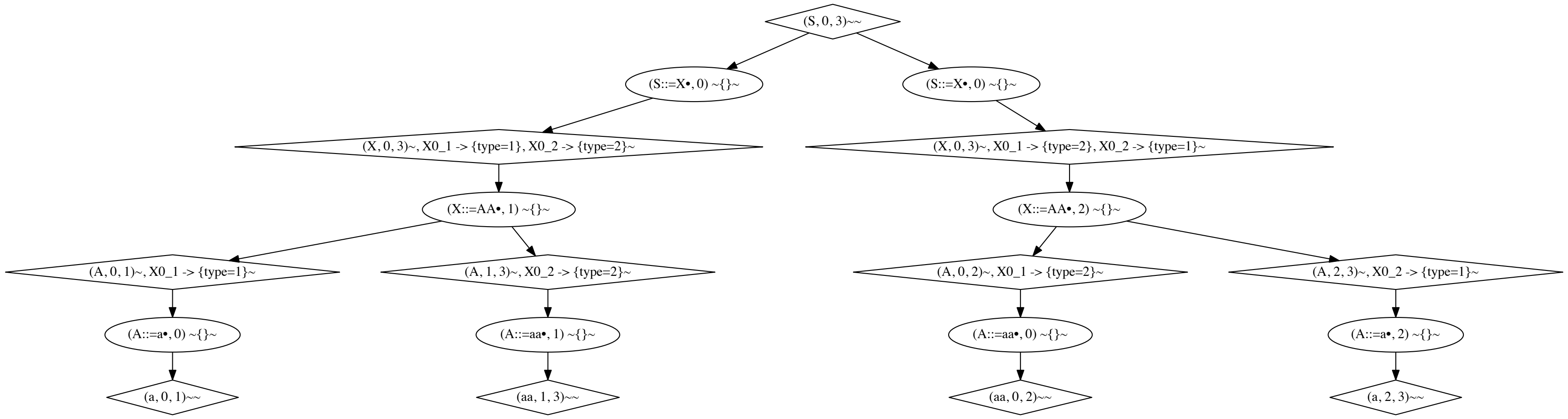

(S, 0, 3)~~
(S::=X•, 0) ~{}~
(S::=X•, 0) ~{}~
(X, 0, 3)~, X0_1 -> {type=1}, X0_2 -> {type=2}~
(X, 0, 3)~, X0_1 -> {type=2}, X0_2 -> {type=1}~
(X::=AA•, 1) ~{}~
(X::=AA•, 2) ~{}~
(A, 0, 1)~, X0_1 -> {type=1}~
(A, 1, 3)~, X0_2 -> {type=2}~
(A, 0, 2)~, X0_1 -> {type=2}~
(A, 2, 3)~, X0_2 -> {type=1}~
(A::=a•, 0) ~{}~
(A::=aa•, 1) ~{}~
(A::=aa•, 0) ~{}~
(A::=a•, 2) ~{}~
(a, 0, 1)~~
(aa, 1, 3)~~
(aa, 0, 2)~~
(a, 2, 3)~~

FIGURE 4.15: Attribute-annotated SPPF for sequence “aaa” with respect to Γ_{18}

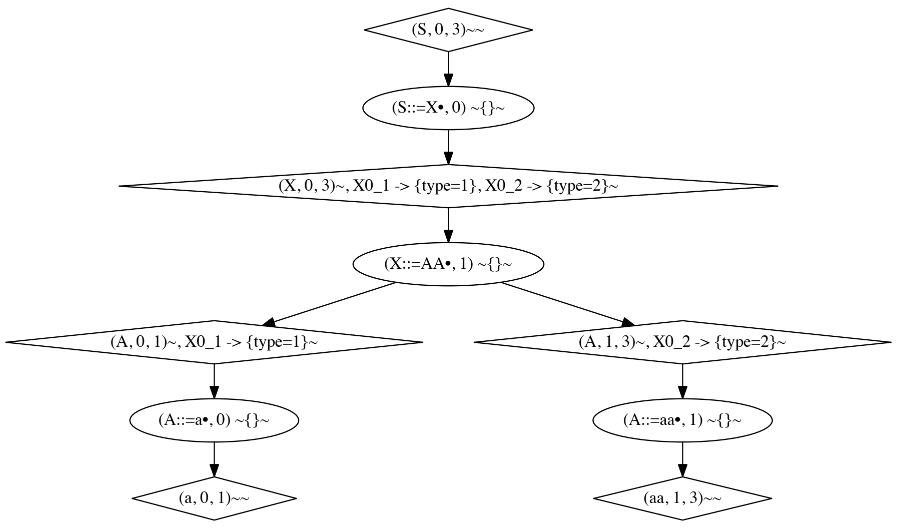

(S, 0, 3)~~
(S::=X•, 0) ~{}~
(X, 0, 3)~, X0_1 -> {type=1}, X0_2 -> {type=2}~
(X::=AA•, 1) ~{}~
(A, 0, 1)~, X0_1 -> {type=1}~
(A, 1, 3)~, X0_2 -> {type=2}~
(A::=a•, 0) ~{}~
(A::=aa•, 1) ~{}~
(a, 0, 1)~~
(aa, 1, 3)~~

FIGURE 4.16: Disambiguated SPPF for sequence “aaa” with respect to Γ_{18}

An additional upside to this method of handling synthesized context is that we yield an annotated SPPF, suited for post-parse attribute evaluation. The downside to this approach is that we modify the structural properties of the SPPF. In a normal SPPF, any given symbol node is unique. There are no two symbol nodes that have the same symbol and input-range. In the SPPF we suggest this can be the case, although the context of these nodes differs. Although this may appear to be a heavy modification to make, it is inherent in the character of context-sensitive parsing. Post-parse sharing may introduce invalid derivations in the parse forest, as was illustrated in Figures 4.9 and 4.10. Thus, to provide a generic solutions, this elimination of sharing in the SPPF is necessary

4.4 Inherited and synthesized attribute mixing

Now that we have seen both inherited and synthesized attributes, let us look closer at their co-existence. Recall the API for their use specified in Section 3.3.
The way we allow attributes to be specified and modified is limited. That is, our attribute evaluation takes place on a depth-first, left-to-right basis (a so called L-attribute grammar). Due to the on-parse time nature of our implementation, and GLL being a top-down algorithm, this is only natural. We can thus pose two questions:

- Can we use synthesized attributes in calculations on inherited attributes?
- Can we use inherited attributes in calculations on synthesized attributes?

First, we can indeed use synthesized attributes in dealing with inherited attributes. However, we can only use attributes that are available at that point in time. To illustate, consider grammar Γ_{20}. In this grammar, we see the start non-terminal passing down property `v` with value 1 to A. The production for A, once finished takes this inherited attribute, adds 1 to it, and synthesizes it back to S (`v` now has value 2). The same pattern is repeated for B and C, resulting in non-terminal S being annotated with $v = 6$. Note that indexing schemes may be confusing here, as `@pre(1)` refers to the moment in time before the `B`, but `attribute(1, "v")` refers to property `v` of `A`.

At any given time, only attributes from processed non-terminals are available. In `@pre(0)`, we could not use attributes from C, as these have not been evaluated yet. Furthermore, attributes inherited to A (i.e. in `@pre(0)`) cannot be accessed in `@pre(1)` but only to GLL Blocks inside the production for A. If one wishes to make these attributes accessible, they should be synthesized back up by hand. However, if one can compute an inherited value in **@pre(0)**, one can certainly compute it in **@pre(1)**, as all the same information is available.

Likewise, we can use inherited attributes in the calculation of synthesized attributes. Note that we are only able to use the inherited attributes from the LHS non-terminal of our production. That is, given a production `A` ::= `BC`, synthesized attributes may only depend on the inherited attributes of A, and the synthesized attributes of B and C. As was the case before, we can only use attributes that have already been derived. So, in the entire production, we may always use the inherited attributes of `A`. However, we cannot use the synthesized attributes of `C` in the `@pre` of `B` or `C` as `C` has not been completed at that time. In the `@pre` of `C` we are only able to use the inherited attributes of `A` and the synthesized attributes of `B`. The synthesized attributes of `C` are only available in the `@post(all)` of this production.

$$(\Gamma_{20}) \qquad \begin{aligned}
&\textbf{S} ::= \textbf{A} \quad \textbf{B} \quad \textbf{C} \sim \\
&@\texttt{pre}(0) : \texttt{attributes.put}(\text{"v"}, 1); \\
&@\texttt{pre}(1) : \texttt{attributes.put}(\texttt{attribute}(1, \text{"v"}), 1); \\
&@\texttt{pre}(2) : \texttt{attributes.put}(\text{"v"}, \texttt{attribute}(2, \text{"v"}) + 1); \\
&@\texttt{post}(\texttt{all}) : \texttt{synthesize}(\text{"v"}, \texttt{attribute}(3, \text{"v"}) + 1); \\
&\sim; \\
&\textbf{A} ::= \text{“}\textbf{a}\text{”} \sim; \\
&@\texttt{post}(\texttt{all}) : \texttt{synthesize}(\text{"v"}, \texttt{attribute}(0, \text{"v"}) + 1); \\
&\sim; \\
&@\texttt{post}(\texttt{all}) : \texttt{synthesize}(\text{"v"}, \texttt{attribute}(0, \text{"v"}) + 1); \\
&\textbf{B} ::= \text{“}\textbf{b}\text{”} \sim; \\
&@\texttt{post}(\texttt{all}) : \texttt{synthesize}(\text{"v"}, \texttt{attribute}(0, \text{"v"}) + 1); \\
&\sim; \\
&\textbf{C} ::= \text{“}\textbf{c}\text{”} \sim; \\
&@\texttt{post}(\texttt{all}) : \texttt{synthesize}(\text{"v"}, \texttt{attribute}(0, \text{"v"}) + 1); \\
&\sim;
\end{aligned}$$

We thus conclude that it is possible to mix inherited and synthesized attributes, but that we enforce a strict ordering on the attributes to avoid ending up with a complicated evaluation scheme. Due to the depth-first left-to-right evaluation of our attributes (and of the GLL algorithm), only attributes from the "past" are available. We cannot use attributes in a computation if those attributes still have to be evaluated.

4.5 Implementation effort

Using the mechanisms we have seen in this chapter, we have created an implementation that allows us to generate GLL parsers based on L-attribute grammar specifications. In Figure 4.17, the initial screen of our implementation (inside netbeans) can be seen. Figure 4.18 shows the specification of a grammar file, from which a parser can be generated (Figure 4.19). When providing an input sequence (visible in Figure 4.20), one can run the parser, which will (provided the input sequence is correct) yield an SPPF, as can be seen in Figure 4.21. Lastly, providing an invalid input sequence (Figure 4.22) will yield to an error message, as can be seen in Figure 4.23.

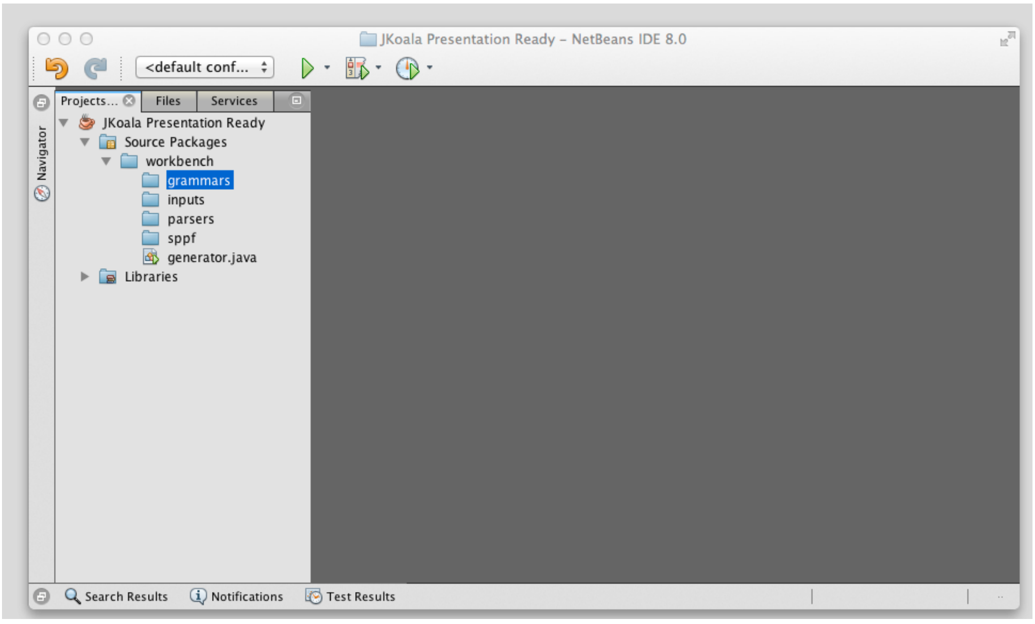
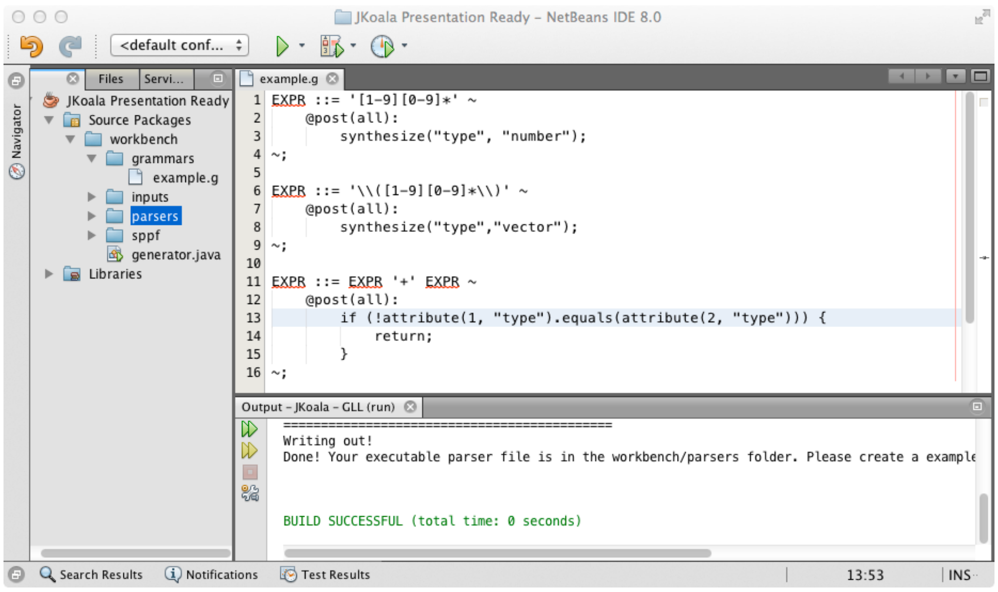


FIGURE 4.17: A screenshot of the home screen of our implementation

4.6 Conclusion

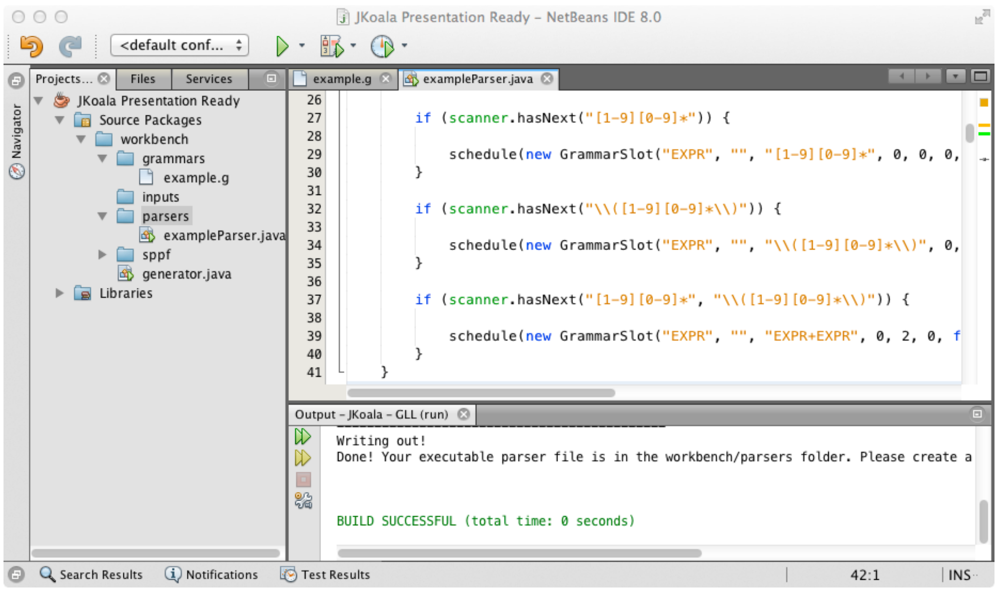
We have successfully extended GLL with support for an attribute system. More specifically, an attribute system that accepts L-attribute grammars. This attribute system is able to provide a context for disambiguation, as we will demonstrate in Chapter 5. However, there are some downsides, the system for inherited attributes can (potentially) destroy the left-recursion handling in GLL. Furthermore, the system that supports synthesized attributes modifies the SPPF so that its sharing is reduced. In this reduction,



```
EXPR ::= '[1-9][0-9]*' ~
    @post(all):
        synthesize("type", "number");
~;

EXPR ::= '\\([1-9][0-9]*\\)' ~
    @post(all):
        synthesize("type","vector");
~;

EXPR ::= EXPR '+' EXPR ~
    @post(all):
        if (!attribute(1, "type").equals(attribute(2, "type"))) {
            return;
        }
~;
```

```
============================================
Writing out!
Done! Your executable parser file is in the workbench/parsers folder. Please create a example


BUILD SUCCESSFUL (total time: 0 seconds)
```

FIGURE 4.18: A grammar file can be created, in which an attributed BNF grammar may be specified

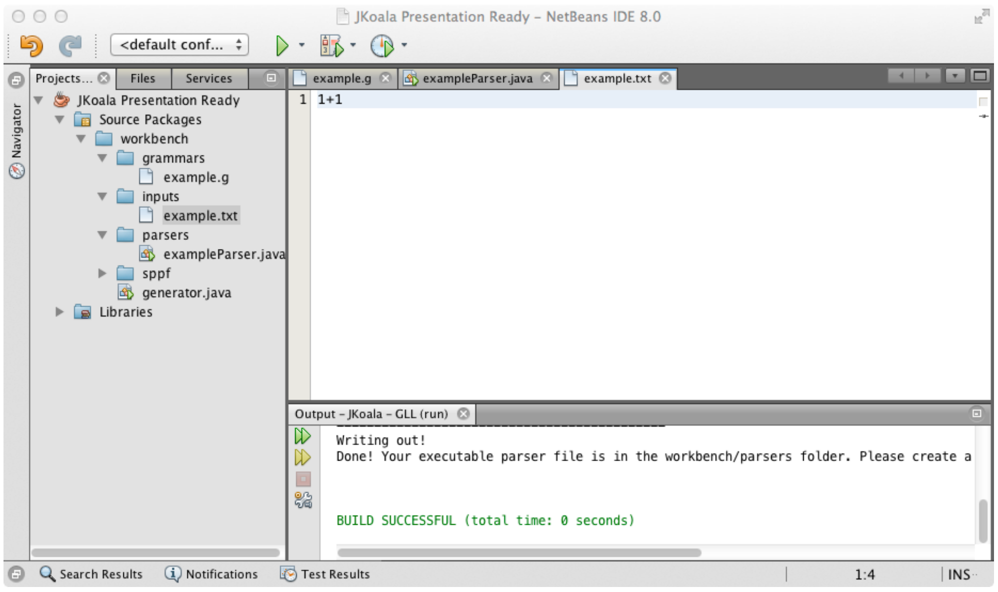


```

            if (scanner.hasNext("[1-9][0-9]*")) {

                schedule(new GrammarSlot("EXPR", "", "[1-9][0-9]*", 0, 0, 0,
            }

            if (scanner.hasNext("\\([1-9][0-9]*\\)")) {

                schedule(new GrammarSlot("EXPR", "", "\\([1-9][0-9]*\\)", 0,
            }

            if (scanner.hasNext("[1-9][0-9]*", "\\([1-9][0-9]*\\)")) {

                schedule(new GrammarSlot("EXPR", "", "EXPR+EXPR", 0, 2, 0, f
            }
        }
```

```
Writing out!
Done! Your executable parser file is in the workbench/parsers folder. Please create a


BUILD SUCCESSFUL (total time: 0 seconds)
```

FIGURE 4.19: Using the provided generator, a parser is generated for a grammar

some structural properties on the SPPF are lost. However, these structural modifications are in the nature of context-sensitive parsing, as we have explained.
Finally, we were able to create an implementation that supports these L-attribute grammars in practice. We have used this implementation to tackle the case-study we will present in Chapter 5.



```
1+1
```

```
Writing out!
Done! Your executable parser file is in the workbench/parsers folder. Please create a

BUILD SUCCESSFUL (total time: 0 seconds)
```

FIGURE 4.20: An input file is created for a grammar

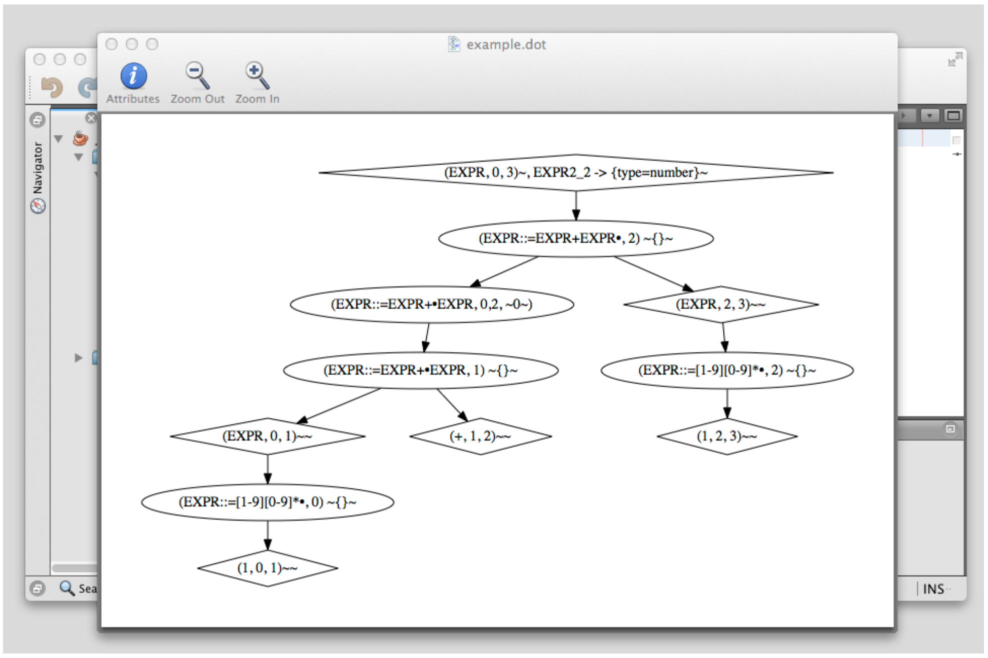
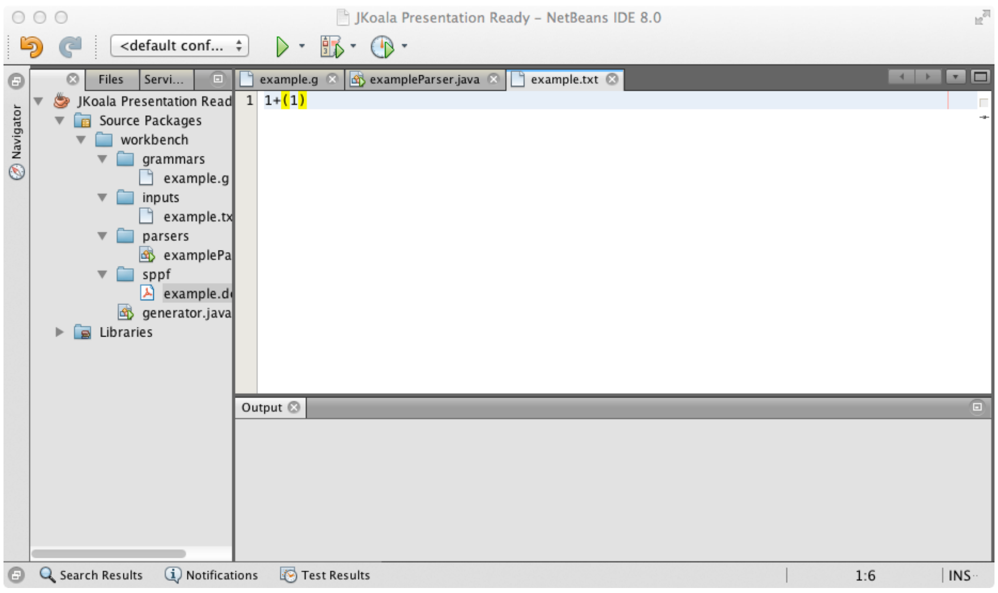
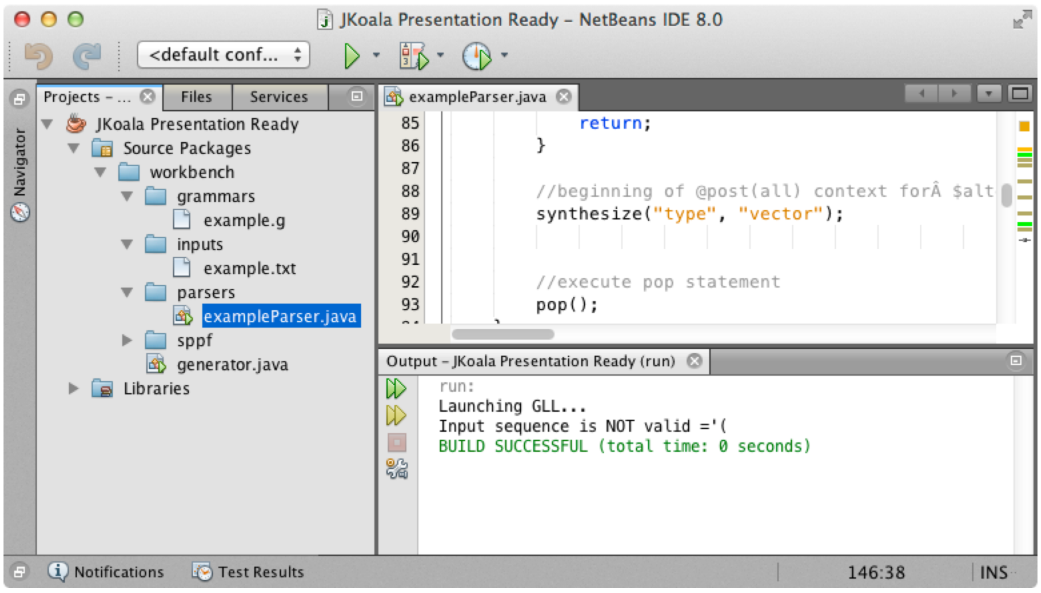

example.dot
Attributes
Zoom Out
Zoom In
(EXPR, 0, 3)~, EXPR2_2 -> {type=number}~
(EXPR::=EXPR+EXPR•, 2) ~{}~
(EXPR::=EXPR+•EXPR, 0,2, ~0~)
(EXPR, 2, 3)~~
(EXPR::=EXPR+•EXPR, 1) ~{}~
(EXPR::=[1-9][0-9]*•, 2) ~{}~
(EXPR, 0, 1)~~
(+, 1, 2)~~
(1, 2, 3)~~
(EXPR::=[1-9][0-9]*•, 0) ~{}~
(1, 0, 1)~~

FIGURE 4.21: The parser can be run on a specified input file, yielding an SPPF



```
1+(1)
```

FIGURE 4.22: Providing a work input sequence



```
            return;
        }

        //beginning of @post(all) context forÂ $alt
        synthesize("type", "vector");


        //execute pop statement
        pop();
```

```
run:
Launching GLL...
Input sequence is NOT valid ='(
BUILD SUCCESSFUL (total time: 0 seconds)
```

FIGURE 4.23: Leads to an incorrect parse

Chapter 5

Case Study : Python

To illustrate the expressive powers of GLL with attributes, we will present two grammars related to Python. First, we will discuss Python expressions in detail. Secondly, we will take Python control structures, and embed the expression grammar in them.

5.1 Expression Grammar

The first type of grammar that we will investigate are expression grammars. An expression grammar is a highly recursive grammar, in which each production represents the application of an operator. The operator is usually a terminal symbol, whilst recursive non-terminals for other (sub-)expressions. We distinct prefix (e.g. $\mathtt{aO}$ where $\mathtt{a} \in \mathtt{T}$ and $O \in N$), infix (e.g. OaP where $a \in T$ and $O, P \in N$), postfix operators (e.g. Oa where $a \in T$ and $O \in N$), and closed expression (e.g. aOb where $a, b \in T$ and $O \in N$). We chose the Python expressions as a real-life example to illustrate the power of our attribute system. However, before going into the actual grammar we will first discuss the patterns and structures that introduce ambiguities in the first place. Then, we will discuss some theoretical difficulties, after which we will present a scheme in which we may automatically generate disambiguation mechanisms for an expression grammar. We will use this automated way of generating attributes to extend the Python expression grammar for disambiguation.

5.1.1 Causes of ambiguity

In expression grammars, ambiguities are mainly caused by two properties: associativity and priority. Priority entails in what order operators are processed. To illustrate, take the sequence $1 + 2 * 3$. Do we process this as $1 + (2 * 3)$ or $(1 + 2) * 3$? This example was already discussed in section 1.1.

Associativity is slightly different. It does not have to do with the difference between operators, but with the equality of operators (or equality of their priorities). If a number of operators have the same priority, it is unclear which to evaluate first. Normal mathematical operators with equal priority are evaluated from left-to-right, but this might not be the case for every operator in every language (e.g. some languages group their `else` to the closest `if`). For instance, if we are presented with the sequence "1+2+3". Should we interpret this as "(1+2)+3" or "1+(2+3)". In this case, the result is equal, but one might consider operations such as raising to power, in which this is definitely not the case.

5.1.2 Difficulties

Disambiguation of the expression grammar brings with it a theoretical problem. What are the semantics of a unary operator with a lower priority than a binary operator. How does it bind? To illustrate, consider grammar Γ_{21} with respect to input sequence "not!a | b".

$$(\Gamma_{21}) \qquad \begin{array}{l} \texttt{E} ::= \texttt{E}\ \text{“}|\text{”}\ \texttt{E};\\ \texttt{E} ::= \text{“}\texttt{!}\text{”}\ \texttt{E};\\ \texttt{E} ::= \text{“}\texttt{not}\text{”}\ \texttt{E};\\ \texttt{E} ::= \text{“}\texttt{a}\text{”}\ |\ \text{“}\texttt{b}\text{”}; \end{array}$$

Convince yourself of the three possible derivations:

1. not(! (a | b))
2. not((! a) | b)
3. (not! a) | b

Say now we want to enforce the following priority restrictions: ! >|> not
We may disqualify tree (1) and (2), due to the fact that we are performing a not after the |, which violates the priority constraint $| \quad > \quad not$. We may disqualify tree (3), due to the fact that we are performing a | before the !, which violates the priority constraint ! >|. The opinions on which tree is correct are argued in existing literature [26–28]. One can argue that their reasoning is not sound as trees are returned that contain indirect violations. We will simply say that none are valid, and leave this issue for future work. Throughout the rest of this thesis, we will assume that all unary operators have higher priority than all binary operators. This assumption excludes this issue from occurring.

5.1.3 Automatic attributes for priority and associativity

As we have seen before, disambiguation of the expression grammar can be divided into two main concerns: priority and associations. We will investigate how to automatically encode both into attribute grammars.

5.1.3.1 Priority

The concept of priority is that, given two operators, the operator with the higher priority should be processed first. We thus need a way of telling our parser which choices to make (i.e. which operators to put lower in the tree). Taking a closer look, GLL has a top-down approach. Hence, if we encounter two possible operators, we should favor the one with the lowest priority. In this way, the operator with the higher priority is pushed deeper into the tree. If we exclude unary operators with lower priority than binary operators, then we can encode this in the following way:

- Assign every operator a priority ϕ.
- To the @pre(all) for every production of operator $\oplus$ add the following code:

```
    if ((int)attribute(0, "priority") > ϕ) {
      DENY;
    }

```

- To the @pre(i) for every production of operator $\oplus$ (where $1 \leq i \leq \#$ gllBlocks) add: “attributes.put(”priority”, ϕ)”.

In this way, we know that priorities may only increase or remain equal, but never decrease. Using a transitive argument, one can thus say that a production with higher priority will always be lower in the tree than a production of lower priority. Closed operators on the other hand require a different approach as they do not introduce ambiguity. A closed expression resets the priority of its contained sub-expression by inheriting the priority attribute with value 0. However, we are still left with associativity conflicts, which will be discussed next.

5.1.3.2 Associatiativity

Note that just adding priority will still not disambiguate some forests completely. The expression “1+1+1” will still have two derivations:

- (1+1)+1
- 1+ (1+1)

This example can be further generalized to any set of operators with equal priority. Hence when we are dealing with ambiguities of operators with equal priority, we have to take their associativity into account. In general associativity describes the side from which we group operators. Either left-to-right (as is usual in normal arithmetic) or right-to-left. For example: sequence $\sigma = 1 \oplus 1 \oplus 1$, where $\oplus$ is an arbitrary binary operator, may be interpreted in two ways.

- If $\oplus$ is left-associative, we should read σ as $(1 \oplus 1) \oplus 1$

- If $\oplus$ is right-associative, we should read σ as $1 \oplus (1 \oplus 1)$

We will extend the priority-based disambiguation code with a mechanism to filter associativities. Note that the associativity code alone is not sufficient, as we require priorities to be passed down to see if we should filter based on associativity at all.

Taking the code for priority as read. For every production of left-binding operator $\oplus$ with priority ϕ add to the @pre(all):

```
if (attribute(0, "priority").equals(ϕ) && attribute(0,"branch").equals("right")) {
  DENY;
}
```

For every production of right-binding operator $\oplus$ with priority ϕ add to the @pre(all):

```
if (attribute(0, "priority").equals(ϕ) && attribute(0,"branch").equals("left")) {
  DENY;
}
```

This ensures that for left-binding operators, we never have applications of the production in the rightmost non-terminal, and for right-binding operators the same for the leftmost non-terminal. All that remains is to pass down the correct branch to each non-terminal. To the @pre of the left recursive non-terminal, we add `attributes.put ("branch", "left")`. Analogously, to the left recursive non-terminal, we add `attributes.put ("branch", "right")` to the right non-terminal. Note that this approach is easily generalized to n-ary operators, by simple numbering each branch (i.e instead of "left" and "right", use "branch1" etc.) and restricting all branches that are not the desired branch (i.e. instead of "if(branch $\neq$ "left")" : "if (branch $\neq$ "branch4")"). Additionally the check for equal priority is essential, because for two operators of different priority, associativity is not defined. We thus need the priority attributes as defined in Section 5.1.3.1 for associativity to work.

5.1.3.3 Remaining conflicts

We have discussed how to solve associativity conflicts with respect to binary (and higher arity) operators. But we have failed to do so for unary operators. One might wonder why this is necessary. Consider the example of two productions for unary operators with equal priority in Grammar Γ_{22}.

$$(\Gamma_{22}) \qquad \begin{aligned} \mathtt{E} &::= \oplus\, \mathtt{E};\\ \mathtt{E} &::= \mathtt{E} \otimes;\\ \mathtt{E} &::= [1-9][0-9]*; \end{aligned}$$

Take, for example, input sequence $\oplus 12 \otimes$. As mentioned before, neither operator takes precedence over the other. This infers that the sequence may be interpreted as $(\oplus 12)\otimes$ or $\oplus(12\otimes)$. We cannot use our normal associativity attribute generation to assign a branch, as there is only one. The usual interpretation for operators of an equal precedence level is to evaluate them left-to-right. The key observation for encoding this into an attribute

TABLE 5.1: Description of the Python expressions taken from [29]

Operator	**Description**
If-else	Conditional Expression
or	Boolean OR
and	Boolean AND
not x	Boolean NOT
in, not in, is, is not, <, <=, >, >=. <>, !=, ==	Comparisons (including membership and identity tests)
—	Bitwise OR
ˆ	Bitwise XOR
&	Bitwise AND
<<, >>	Shifts
+, -	Addition and subtraction
*, /, //, %	Multiplication, Division, Floor division, remainer
+x, -x,	Positive, Negative
x[index], x[index:index], x(arguments...), x.attribute	Subscription, Slicing, call, attribute reference
(expressions...), [expressions...], {key:value...}	Binding or tuple display, list display, dictionary display

grammar is that, interpreting from left-to-right, we will always read prefix operator symbols before post-fix symbols. We can thus suffice by increasing the priority level of all unary prefix operators such that they are greater than all unary postfix operators in their class.

$$\cdots > \oplus, \otimes > \cdots \Rightarrow \cdots \oplus > \otimes > \cdots$$

If right-to-left evaluation is desirable, all postfix operators should receive an increased priority, rather than all prefix operators.

5.1.4 Python Expressions

Now that we have seen a means of generating rules that resolve priority and associativity conflicts, let us look at them in a real-life example. As a case study, the Python expressions have been chosen (more specifically Python2) [29]. The operators incorporated in the Python grammar are described in Table 5.1. For simplicity we have omitted the lambda operator. Furthermore, the parser for our attribute specification is very simplistic, hence we omit bitwise NOT (∼) to avoid conflicts. Operators are presented in increasing order. That is, `if-else` has lowest priority and is orsbinds weakest, `bindingt` (which is a construct in the python language) has the highest priority and binds strongest. Furthermore, the priorities of operators in the same row are equal and all operat bind left-to-righ. The context-free grammar for these expressions is presented in grammar Γ_{23}. Note that an empty alternate (i.e. |;) represents ε.

(Γ_{23}) `EXPRESSION` ::= `EXPR`;

EXPR ::= “if” EXPR “ : ” EXPR ELSEIFEXPR ELSEEXPR;

ELSEIFEXPR ::= “elseif : ” EXPR ELSEIFEXPR |;

ELSEEXPR ::= “else : ” EXPR ELSEEXPR | ;

EXPR ::= EXPR “or” EXPR ;

EXPR ::= EXPR “and” EXPR;

EXPR ::= “not” EXPR;

EXPR ::= EXPR “in” EXPR;

EXPR ::= EXPR “not in” EXPR;

EXPR ::= EXPR “is” EXPR;

EXPR ::= EXPR “is not” EXPR;

EXPR ::= EXPR “ < ” EXPR;

EXPR ::= EXPR “ <= ” EXPR;

EXPR ::= EXPR “ >= ” EXPR;

EXPR ::= EXPR “ <> ” EXPR;

EXPR ::= EXPR “! = ” EXPR;

EXPR ::= EXPR “ == ” EXPR;

EXPR ::= EXPR “ | ” EXPR;

EXPR ::= EXPR “ ∧ ” EXPR;

EXPR ::= EXPR “ & ” EXPR;

EXPR ::= EXPR “ << ” EXPR;

EXPR ::= EXPR “ >> ” EXPR;

EXPR ::= EXPR “ + ” EXPR;

EXPR ::= EXPR “ − ” EXPR;

EXPR ::= EXPR “/” EXPR;

EXPR ::= EXPR “//” EXPR;

EXPR ::= EXPR “ ∗ ” EXPR;

EXPR ::= EXPR “%” EXPR;

EXPR ::= “ + ” EXPR;

EXPR ::= “ − ” EXPR;

EXPR ::= EXPR “ ∗ ∗” EXPR;

EXPR ::= ATOM “.” IDENTIFIER;

EXPR ::= EXPR “[” EXPR “ : ” EXPR “]” ;

EXPR ::= EXPR “[” EXPR “]” ;

EXPR ::= EXPR “(” EXPR EXPRS “)” | EXPR “()”;

EXPR ::= “(” EXPR EXPRS “)” | “()” ;

EXPR ::= “[” EXPR EXPRS “]” | “[]”;

EXPR ::= “{” KVP KVPS “}” | “{}”;

EXPRS ::= “, ” EXPR EXPRS | ;

$$\texttt{KVP} \quad ::= \quad \texttt{ATOM} \quad \text{“ : ”} \quad \texttt{EXPRESSION};$$
$$\texttt{KVPS} \quad ::= \quad \text{“,”} \quad \texttt{KVP} \quad \texttt{KVPS} \quad |;$$
$$\texttt{IDENTIFIER} ::= [\texttt{a}-\texttt{zA}-\texttt{Z}][\texttt{a}-\texttt{zA}-\texttt{Z0}-\texttt{9}]*;$$
$$\texttt{EXPR} ::= \texttt{ATOM};$$
$$\texttt{ATOM} ::= \texttt{NAT} \mid \texttt{BOOLEAN} \mid \texttt{STRING} \mid \texttt{IDENTIFIER};$$
$$\texttt{NAT} ::= [\texttt{1}-\texttt{9}][\texttt{0}-\texttt{9}]* \mid \text{“}\texttt{0}\text{”};$$
$$\texttt{BOOLEAN} ::= \text{“}\texttt{True}\text{”} \mid \text{“}\texttt{False}\text{”};$$
$$\texttt{STRING} ::= [\backslash\text{”}][\texttt{a}-\texttt{zA}-\texttt{Z}]*[\backslash\text{”}]$$

Now that we have the normal BNF grammar, we can begin extending it with respect to the schemes presented in Sections 5.1.3.1 and 5.1.3.2. There are a few non-standard disambiguations need for grammar Γ_{27}. One is the reserved keyword mechanism implemented on the production for `IDENTIFIER`. We reject the current derivation if the identifier derives a keyword from the set of restricted keywords. Furthermore, the highest priority operators (Binding, tuple display etc.) do not cause ambiguity, hence we have omitted annotating them with attributes and disambiguation mechanisms. However, as mentioned in Section 5.1.3.1, they reset the priority attribute to 0, to correctly support fresh sub-expressions. These extension rules together ensure that one can write an ambiguity free syntactically correct Python expression. In addition the the non-standard mechanisms (keyword restrictions and priority resets) mentioned before, also the standard generated attributes have been added to prevent priority and associativity conflicts. The result of this can be found in appendix A.
With the addition of these attributes, we are able to resolve all local ambiguities related to associativity and priority. We consider an ambiguity local, if we can decide that a derivation is incorrect without knowledge of other derivations. Using these attributes, we are able to disambiguate all ambiguities related to priority and associativity. We will provide correctness arguments for these statements next.

Assume towards a contradiction, that there is an ambiguity related to priorities. This means that there must be two derivations in which two operators of different priority levels occur above and below each other. In other words, we have a derivation δ_1 in which $\oplus$ is higher in the tree than $\otimes$, and δ_2 in which $\otimes$ is higher in the tree than $\oplus$. As these operators have different priorities (otherwise it would not be a priority-based ambiguity) and we have encoded our priority attribute to be non-decreasing, one of these derivations must be incorrect. For if they are not, we can derive a contradiction (i.e. it cannot be the case that $\mathit{priority}(\oplus) > \mathit{priority}(\otimes)$ and $\mathit{priority}(\otimes) > \mathit{priority}(\oplus)$ both hold).

For associativity, assume towards a contradiction that there is an associativity-based ambiguity. This means we have two (possibly the same) operators of equal priority. Note that this pattern can also occur for two operators of equal priority. For these operators there must be two derivations, in which one of the operators is embedded in a subbranch of the other (e.g. $(\_ \oplus \_) \otimes \_$ or $\_ \oplus (\_ \otimes \_)$). If operators of equal

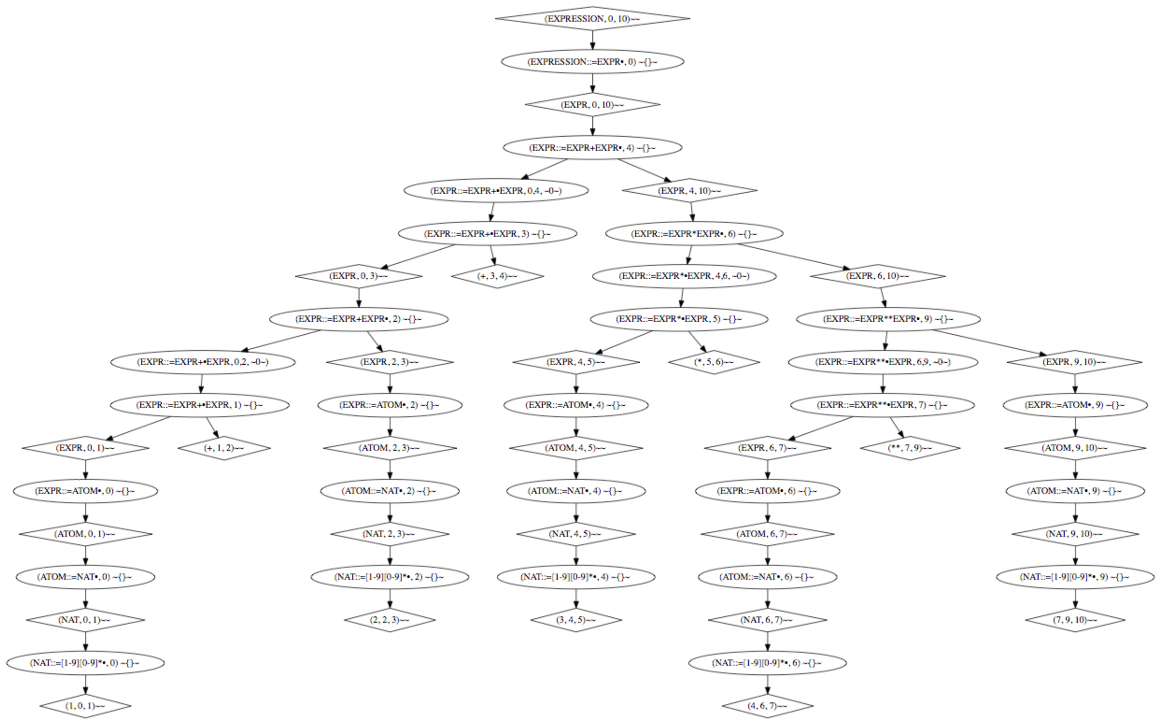


FIGURE 5.1: The SPPF for sequence 1+2+3*4**7 under the Attributes Python expression grammar

priority then group towards a single direction (either left, right, or an arbitrary branch that is fixed), something that holds for our Python grammar, this ambiguty leads to a contradiction. Note that the ambiguity described arises because both derivations choose a different branch to continue their derivation. This is a clear contradiction to the single grouping direction we have assumed. Note that if operators of a given priority group to different sides, this encoding does not work. To illustrate, assume $\oplus$ is right-binding (i.e. $\_ \oplus \_ \oplus \_ = \_ \oplus (\_ \oplus \_)$) and $\otimes$ is left-binding (i.e. $\_ \otimes \_ \otimes \_ = (\_ \otimes \_) \otimes \_$). If both are of equal priority then both $\_ \otimes (\_ \oplus \_)$ and $\_ \oplus (\_ \otimes \_)$ are valid derivations. (i.e. the ambiguity described is inherent in the language).

For a more practical proof, consider sequence `1+2+3*4**7`, for which the SPPF can be found in Figure 5.1. As this SPPF is quite large, it is probably not readable. However, note that we have obtained a nice tree structure, in which no ambiguities are present.

5.2 Offside rule

The offside rule is used in programming languages to denote the block structure of that language. Rather than the classic braces ({}) used in languages such as C or Java, languages using the offside rule use indentation to specify the nesting of block structures. That is, statements are part of the block structure of construct if their indenting is equal. Furthermore, the indenting of statements inside a control-structure must be greater than the indenting of the control itself. To illustrate, consider Listing 5.1. Note that `x=4` and `k=7` belong to the same block structure, namely to that of `if (x == True)`. Also note that indeed the indenting of the assignments is greater than that of the `if`. In normal python, the amount of extra indenting for statements in a control structure is arbitrary (as long as it is more than that of the control structure). For simplicity, we will consider the case where there is only one extra level of indenting. Also note, that the assignments `x = 5` and `x = 7` are no longer part of that control structure, as the level of indenting is interrupted by the lower-indented `if (x == False)`. In Listing 5.2 an exampple of incorrect indenting is given, as the `x=4` should have an indenting greater than that of the `if (x == True)`.

```
  if (x == True)
    x = 4
    y = 6
  if (x == False)
    x = 5
    y = 7
```

Listing 5.1: An example of block structure using indenting

```
  if (x == True)
  x = 4
```

Listing 5.2: An example of incorrect indenting

This indenting can be used to solve problems related to control-structure such as the dangling else problem. To illustrate, in the code in Listing 5.3 the `else` statement belongs to the second `if`, rather than the first `if`, because its indenting is equal to that of the second `if`. The performC() belongs to the block of the first `if`, rather than that of the `else`, because its indenting is equal to that of the first `if`.

```
begin
  if (condition1)
    if (condition2)
      performA();
    else
      perfromB();
    performC();
end
```

Listing 5.3: A piece of code using the offside rule

5.2.1 Causes of ambiguity

Ambiguity in in languages that support the offside rule is mainly caused by the lack of knowledge with respect to the actual indenting. That is, via a context-free production one cannot (by definition) express how much indenting any of the previous block structures have (because this is context-sensitive). We can thus conclude that the offside-rule cannot be expressed by a pure context-free grammar (i.e. the offside-rule is not context-free). If we would try and specify the syntax of the Python language in a context-free grammar, many ambiguities would arise due to the many possible block structures a statement can be in. Consider the following example.

In grammar Γ_{24} a small language that uses the offside rule has been specified. The language consists of an `if` statement and two `callFunction()` statements. Consider Listing 5.4 that shows a small fragment of code using the offside-rule. Note that `callFunctionA` is inside the scope of the `if`, while `callFunctionB` is not. However, if we were to look at all possible derivations for the sequence, we could derive `callFunctionB` to be inside the scope of the `if`. This is due to the production for `callFunctionB` not being aware that the indenting does not correspond to the desired indenting.

Here, the attribute system could be applied to annotate the indenting of all statements inside a block structure with the desired depth. Then, if the indenting is not of the correct depth, the derivation can be prunes, potentially decreasing the number of derivations. Consider an input in which we have a number of consecutive statements. If the indenting of the first statement is found to be incorrect, the rest of the statements do not need to be parsed in that derivation. We will see a larger example of this this in Section 5.2.2.

$$
(\Gamma_{24}) \qquad
\begin{aligned}
\texttt{P} &::= \texttt{STATEMENTS};\\
\texttt{STATEMENTS} &::= \texttt{STATEMENT} \quad \texttt{STATEMENTS} \quad | \quad \texttt{STATEMENT};\\
\texttt{STATEMENT} &::= \texttt{INDENTING} \quad \text{“}\texttt{if}\backslash\texttt{n}\text{”} \quad \texttt{STATEMENTS};\\
\texttt{STATEMENT} &::= \texttt{INDENTING} \quad \text{“}\texttt{callFunctionA();}\text{”}\\
\texttt{STATEMENT} &::= \texttt{INDENTING} \quad \text{“}\texttt{callFunctionB();}\text{”}\\
\texttt{INDENTING} &::= \text{“}[\backslash\texttt{t}]*\text{”};
\end{aligned}
$$

```
if
  callFunctionA();
callFunctionB();
```

LISTING 5.4: An example of offside code

5.2.2 Python constructs

In this section, we will discuss a grammar that uses a subset of the Python control-flow constructs in an offside-setting. We will proceed to extend this grammar with attributes to eliminate ambiguities. Inside the grammars presented in this section, we will use the non-terminal `EXPRESSION`, as we have defined it in grammar Γ_{27}. We consider the following additional structures:

- for-loops
- while-loops
- if-statements
- if-else statements

5.2.3 Generating attributes for the offside rule

Before actually extend the grammar, we will first see how to do it in general. The issue we aim to solve is that entities inside a block structure of the language are aware of their expected indenting. To this extent, we will recursively annotate each entity with their expected amount of indenting, and enforcing this at the lowest level. The control structures that require indenting will then modify the expected depth for the statements that it contains. More specifically, every control structure adds a piece of annotated Java to the @pre of their recursive statements as presented in Listing 5.5.

```
int expected = attribute(0, "expected");
attributes.put("expected", expected+1);
```

Listing 5.5: Java code for the indenting attribute

Furthermore, the non-terminal that handles the indenting should then enforce the expected indenting. In this example we enforce the level of indenting in the production `INDENTING` ::= [\`t`]∗ using API calls to find out how many indents were consumed. If the actual amount of indenting does not match the expected amount of indenting, the derivation will be rejected. Note again the we restrict ourselves to preciously one level of extra indentation. The result has been presented in Listing 5.6.

```
int start = currentInputPosition;
int end = scanner.getPosition();
int indenting = end - start;
if ((int)attribute(0, "exepcted") != indenting) {
  DENY;
}
```

Listing 5.6: Enforcing of expected indenting

5.2.4 The grammar

Now that we have seen how to generate attributes for our productions, let us look at some actual productions. As mentioned earlier, we use non-terminal `EXPRESSION` as defined in Γ_{27}. In our grammar, we specify a program padded by “begin” and “end” terminals. Furthermore, the program consists of statements. The structure of our attributes will be such that we require each of these statement to have an expected indenting, which we will then enforce. We require our program to have at least one statement. We support a number of different types of statements, namely the statements mentioned at the beginning of this chapter and assignments. Additionally, we regard an expression as a statement, as is the case in the Python interpreter. The result can be found in Grammar Γ_{25}.

$$(\Gamma_{25}) \qquad \begin{aligned} \texttt{PROGRAM} &::= \text{“}\texttt{begin\textbackslash n}\text{”} \quad \texttt{STATEMENTS} \quad \text{“}\texttt{end}\text{”}; \\ \texttt{STATEMENTS} &::= \texttt{STATEMENT} \quad | \quad \texttt{STATEMENT} \quad \texttt{STATEMENTS}; \\ \texttt{STATEMENT} &::= \texttt{INDENTING} \quad \texttt{EXPRESSION} \quad \text{“}\texttt{\textbackslash n}\text{”}; \end{aligned}$$

$$\begin{aligned}
\texttt{STATEMENT} ::= \texttt{INDENTING}\quad \texttt{IDENTIFIER}\quad ``=" \quad \texttt{EXPRESSION}\quad ``\backslash\texttt{n}";\\
\texttt{STATEMENT} ::= \texttt{INDENTING}\quad ``\texttt{for}"\quad \texttt{EXPRESSION}\quad ``\texttt{in}"\quad \texttt{EXPRESSION}\\
``:\backslash\texttt{n}"\quad \texttt{STATEMENTS};\\
\texttt{STATEMENT} ::= \texttt{INDENTING}\quad ``\texttt{while}"\quad \texttt{EXPRESSION}\quad ``:\backslash\texttt{n}"\quad \texttt{STATEMENTS};\\
\texttt{STATEMENT} ::= \texttt{IF}\quad \texttt{STATEMENTS}\quad \texttt{ELSE}\quad \texttt{STATEMENTS};\\
\texttt{STATEMENT} ::= \texttt{IF}\quad \texttt{STATEMENTS};\\
\texttt{STATEMENT} ::= \texttt{IDENTIFIER}\quad ``="\quad \texttt{EXPRESSION};\\
\texttt{IF} ::= \texttt{INDENTING}\quad ``\texttt{if(}"\quad \texttt{EXPRESSION}\quad ``):\backslash\texttt{n}";\\
\texttt{ELSE} ::= \texttt{INDENTING}\quad ``\texttt{else}:\backslash\texttt{n}";\\
\texttt{INDENTING} ::= ``[\backslash\texttt{t}]*";
\end{aligned}$$

Now that we have a basic grammar, we can continue with extending it. In grammar Γ_{26}, generated attributes have been added to the control structure, and the production for INDENTING has been extended to enforce indenting levels. Additionally, we have incorporated the expression grammar developed in Section 5.1.4 and incorporated them into the guards of our control structures. The resulting grammar allows us to parse Python programs without any ambiguities. We are now able to disambiguate offside constructs, and as we were already able to disambiguate expressions, we can now write small Python programs that are ambiguity free. For example, consider the small Python program in Listing 5.7 for which the SPPF has been illustrated in Figure 5.2. Again we note that there are no ambiguities and that we have obtained a single tree.

```
begin
  for x in [1,2,3,4,5]:
    if (x<=5):
      y=if x<2:0 else : 1
    else:
      y=2
end
```

LISTING 5.7: A small Python program

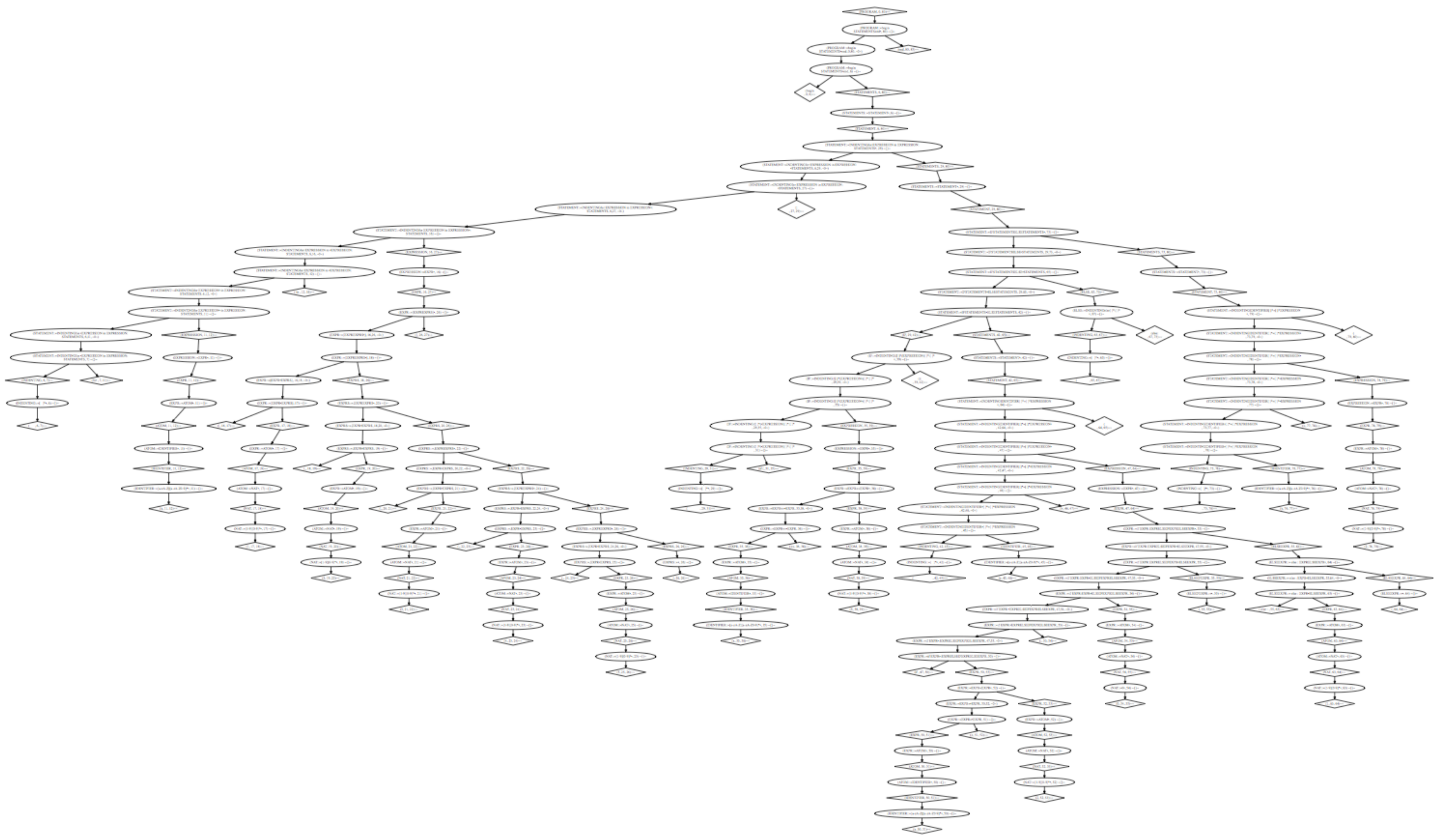


FIGURE 5.2: The SPPF for a small Python program

$$
\begin{aligned}
(\Gamma_{26}) \qquad & \textbf{PROGRAM} ::= \text{“}\textbf{begin\textbackslash n}\text{”}\textbf{STATEMENTS}\text{“}\textbf{end}\text{”} \sim \\
& \texttt{@pre(0)} : \texttt{attributes.put(“expected”}, 1); \\
& \sim; \\
& \textbf{STATEMENTS} ::= \textbf{STATEMENT} \quad | \quad \textbf{STATEMENT} \quad \textbf{STATEMENTS}; \\
& \textbf{STATEMENT} ::= \textbf{INDENTING} \quad \textbf{EXPRESSION}\text{“}\textbf{\textbackslash n}\text{”}; \\
& \textbf{STATEMENT} ::= \textbf{INDENTING} \quad \textbf{IDENTIFIER}\text{“} = \text{”}\textbf{EXPRESSION}\text{“}\textbf{\textbackslash n}\text{”}; \\
& \textbf{STATEMENT} ::= \textbf{INDENTING} \quad \text{“}\textbf{for}\text{”} \quad \textbf{EXPRESSION} \quad \text{“}\textbf{in}\text{”} \quad \textbf{EXPRESSION} \\
& \text{“} : \textbf{\textbackslash n}\text{”} \quad \textbf{STATEMENTS} \sim \\
& \texttt{@pre(3)} : \\
& \texttt{int} \quad \texttt{e} = \texttt{(int)attribute(0, “expected”)}; \\
& \texttt{attributes.put(“expected”}, \texttt{e} + 1); \\
& \sim; \\
& \textbf{STATEMENT} ::= \textbf{INDENTING} \quad \text{“}\textbf{while}\text{”} \quad \textbf{EXPRESSION} \\
& \text{“} : \textbf{\textbackslash n}\text{”} \quad \textbf{STATEMENTS} \sim \\
& \texttt{@pre(2)} : \\
& \texttt{int} \quad \texttt{e} = \texttt{(int)attribute(0, “expected”)}; \\
& \texttt{attributes.put(“expected”}, \texttt{e} + 1); \\
& \sim; \\
& \textbf{STATEMENT} ::= \quad \textbf{IF} \quad \textbf{STATEMENTS} \quad \textbf{ELSE} \quad \textbf{STATEMENTS} \sim \\
& \texttt{@pre(1)} : \\
& \texttt{int} \quad \texttt{e} = \texttt{(int)attribute(0, “expected”)}; \\
& \texttt{attributes.put(“expected”}, \texttt{e} + 1); \\
& \texttt{@pre(3)} : \\
& \texttt{int} \quad \texttt{e} = \texttt{(int)attribute(0, “expected”)}; \\
& \texttt{attributes.put(“expected”}, \texttt{e} + 1); \\
& \sim; \\
& \textbf{STATEMENT} ::= \textbf{IF} \quad \textbf{STATEMENTS} \sim \\
& \texttt{@pre(1)} : \\
& \texttt{int} \quad \texttt{e} = \texttt{(int)attribute(0, “expected”)}; \\
& \texttt{attributes.put(“expected”}, \texttt{e} + 1); \\
& \sim; \\
& \textbf{STATEMENT} ::= \textbf{IDENTIFIER} \quad \text{“} = \text{”} \quad \textbf{EXPRESSION}; \\
& \textbf{IF} ::= \textbf{INDENTING} \quad \text{“}\textbf{if(}\text{”} \quad \textbf{EXPRESSION} \quad \text{“}) : \textbf{\textbackslash n}\text{”}; \\
& \textbf{ELSE} ::= \textbf{INDENTING} \quad \text{“}\textbf{else : n}\text{”}; \\
& \textbf{INDENTING} ::= [\backslash\textbf{t}]* \sim \\
& \texttt{@post(all)} : \\
& \texttt{int} \quad \texttt{before} = \texttt{currentInputPosition};
\end{aligned}
$$

```
int after = scanner.getPosition();
int expected = (int)attribute(0, "expected");
if(after - before != expected){
return;
}
~;
```

5.3 Conclusions

Having added attributes to the python grammar, we can now resolve all ambiguities. Related to the offside rule, this is the case because every statement now has information on which depth it should reside, and this depth is also enforced. Assume (towards a contradiction) an offside-related ambiguity does exist. This means a statement can be parsed to conform to two block structures. That is, it can be parsed to have two different levels of indenting. This cannot be the case, as each statement is annotated with a specific value for its expected indenting. If a different indenting than the expected indenting is parsed, the derivation is rejected. A statement can therefore not be parsed with two different levels of indenting. Trough this disambiguation, we have obtained a great increase. Rather than yield every possible derivation, if a conflict with respect to indenting is rejected in a statement, we can prune work on all other statements in that derivation.

Chapter 6

Conclusions and Future work

In this thesis, we have looked at augmenting the GLL algorithm with an attribute system (Chapters 2 and 4). We have investigated the ability to, on-parse time, reject derivations of the generated parser based on these attributes. To be able to implement the attribute system, we have had to gain a deep understanding of the data structures and their interaction in the algorithm. The sharing that is implemented in the various data-structures of the GLL algorithm are very powerful. In some instances, a little too powerful, as they cause issues. After having relaxed the sharing to a level that is suitable, we find that the mechanisms are very generic and elegant and support our augments without difficulties. We have seen ways to disambiguate actual grammars using the attribute system without having to modify the shape of the grammer (with respect to production rules). We conclude that attribute grammars are feasible for GLL, and that they are a very nice way of efficient disambiguation, as disambiguation is performed as early as possible (on-parse time).

However, the attribute system does come with some downsides that should be considered. The attribute system requires relaxation of sharing from the original GLL algorithm. If used in a very naive way, the attribute system can destroy left-recursion (as was mentioned in Section 4.2.2). Note that this concerns highly artificial cases, as for most grammars, this is not the case. Additionally, the attribute system we have presented allow for higher order attributes (i.e. objects). This can make comparing attributes very expensive, voiding the time-gain of the on-parse time pruning of derivations.

6.1 Contribution to research questions

6.1.1 Question 1

Is it possible to extend the Generalized LL parsers with attribute grammars? Yes, in Sections 4.1 and 4.3 we have seen how we can pass along attributes to non-terminals. In particular, in Section 4.3 we have shown that we can annotate parse forests with attributes, suited for post-parser filtering.

6.1.2 Question 2

Is it possible to push evaluation of the attributes with respect to their restrictions into the parsers (in contrast to post-parse evaluation)?
To a large extent, evaluation of attributes on-parse time is possible. We have considered pruning derivations if a conflict in attributes was found. As a consequence, we cannot reject derivations in a more global scope. That is, if we have obtained a derivation d_1, the presence of a derivation d_2 cannot invalidate d_1. This is something that might be desirable, as we have seen in, for example, Section 5.2.2. Altogether, almost all useful disambiguation can be pushed into the parser, thus drastically decreasing disambiguation and parse time.

6.1.3 Question 3

Are attribute grammars suited for practical on-parse time disambiguation?
As we have seen in Chapter 5, we are able to disambiguate both realistic expression and offside grammars. Additionally, time is increased as we are pruning unwanted derivation as soon as possible. Moreover, we have shown that attribute grammars are excelent in combination with the grammar modularity, which is one of the highlights of GLL.

6.2 Future Work

In this work we have encountered a few areas in which future work may be conducted.

6.2.1 Precedence of unary operators

As was noted in Section 5.1.2, when one is dealing with unary operators that have lower priority than binary operators, there are some unclear edge cases. Further research is needed in order to ascertain which derivations are desired and why, before we can encode such precedence relations in an attribute grammar.

6.2.2 Production parser

The OOGL parser that was used as a basis for the attribute system is already quite fast, compared to the original academic OOGL parser [1]. However, performance increases are still available and should be explored further in order for GLL to become a viable option for language engineering (with or without attribute grammars).

6.2.3 EBNF

Both the OOGLL parser and this attribute extension were performed on BNF-based GLL. However, most production grammars these days are in EBNF, requiring grammar rewriting to allow them to be used with the system we have defined. It thus seems very interesting to repeat the current studies for EBNF grammar, rather than BNF.

6.2.4 Non-local disambiguation

Lastly, the research that has been done on non-local disambiguation using SPPF path removal [30] is very powerful. Experiments have already been conducted to incorporate these filters on-parse time. A fruitful area of research would be to see if these techniques can be incorporated in the attribute grammar approach, seeing if we can do more than just local pruning. With these non-local decisions, the ambiguities mentioned in Section 5.2.2 could be handled in a more elegant fashion.

6.3 Thanks to the reader

I would like to conclude by thanking the reader for taking the time to read through this thesis. I hope this work serves as a fruitful base for many more investigations into GLL to come, and wish to inspire those that will continue in this fields with the words:

> “The competent programmer is fully aware of the limited size of his own skull. He therefore approaches his task with full humility, and avoids clever tricks like the plague.”
> -Edsger W. Dijkstra

Bibliography

[1] Bram Cappers, Josh Mengerink, Bram van der Sanden, and Mark van den Brand. Object oriented gll with error handling. 2014.

[2] Elizabeth Scott and Adrian Johnstone. Gll parse-tree generation. *Science of Computer Programming*, 78(10):1828–1844, 2013.

[3] Seppo Sippu and Eljas Soisalon-Soininen. *Parsing Theory: LR (k) and LL (k) Parsing*, volume 20. Springer, 1990.

[4] Noam Chomsky and Marcel P Schützenberger. The algebraic theory of context-free languages. *Studies in Logic and the Foundations of Mathematics*, 26:118–161, 1959.

[5] John E Hopcroft, Rajeev Motwani, and Jeffrey D Ullman. Introduction to automata theory, languages, and computation. *ACM SIGACT News*, 32(1):60–65, 2001.

[6] Donald E Knuth. Semantics of context-free languages. *Mathematical systems theory*, 2(2):127–145, 1968.

[7] Henk Alblas. Introduction to attribute grammars. In *Attribute Grammars, Applications and Systems*, pages 1–15. Springer, 1991.

[8] Philip M Lewis, Daniel J Rosenkrantz, and Richard Edwin Stearns. Attributed translations. In *Proceedings of the fifth annual ACM symposium on Theory of computing*, pages 160–171. ACM, 1973.

[9] Martin Jourdan. Strongly non-circular attribute grammars and their recursive evaluation. *SIGPLAN Not.*, 19(6):81–93, June 1984. ISSN 0362-1340. doi: 10.1145/502949.502882. URL http://doi.acm.org/10.1145/502949.502882.

[10] Uwe Kastens. Ordered attributed grammars. *Acta Informatica*, 13(3):229–256, 1980.

[11] Henk Alblas. Attribute evaluation methods. In *Attribute Grammars, Applications and Systems*, pages 48–113. Springer, 1991.

[12] Uwe Kastens. Implementation of visit-oriented attribute evaluators. In *Attribute Grammars, Applications and Systems*, pages 114–139. Springer, 1991.

[13] Mehdi Jazayeri, William F. Ogden, and William C. Rounds. The intrinsically exponential complexity of the circularity problem for attribute grammars. *Commun. ACM*, 18(12):697–706, December 1975. ISSN 0001-0782. doi: 10.1145/361227.361231. URL http://doi.acm.org/10.1145/361227.361231.

[14] Henk Alblas. Incremental attribute evaluation. In *Attribute Grammars, Applications and Systems*, pages 215–233. Springer, 1991.

[15] Ken Kennedy and Scott K. Warren. Automatic generation of efficient evaluators for attribute grammars. In *Proceedings of the 3rd ACM SIGACT-SIGPLAN Symposium on Principles on Programming Languages*, POPL '76, pages 32–49, New York, NY, USA, 1976. ACM. doi: 10.1145/800168.811538. URL http://doi.acm.org/10.1145/800168.811538.

[16] Pierre Deransart, Martin Jourdan, and Bernard Lorho. *Attribute grammars: definitions, systems and bibliography*, volume 323. Springer, 1988.

[17] Jukka Paakki. Attribute grammar paradigms - a high-level methodology in language implementation. *ACM Comput. Surv.*, 27(2):196–255, June 1995. ISSN 0360-0300. doi: 10.1145/210376.197409. URL http://doi.acm.org/10.1145/210376.197409.

[18] Valentin David, Akim Demaille, and Olivier Gournet. Attribute grammars for modular disambiguation. In *Proceedings of the IEEE 2nd International Conference on Intelligent Computer Communication and Processing (ICCP'06), Technical University of Cluj-Napoca, Romania*, 2006.

[19] Eric Van Wyk, Lijesh Krishnan, Derek Bodin, and August Schwerdfeger. Attribute grammar-based language extensions for java. In *ECOOP 2007–Object-Oriented Programming*, pages 575–599. Springer, 2007.

[20] Eric Van Wyk, Derek Bodin, Jimin Gao, and Lijesh Krishnan. Silver: An extensible attribute grammar system. *Science of Computer Programming*, 75(1–2): 39 – 54, 2010. ISSN 0167-6423. doi: http://dx.doi.org/10.1016/j.scico.2009.07.004. URL http://www.sciencedirect.com/science/article/pii/S0167642309001099. Special Issue on {ETAPS} 2006 and 2007 Workshops on Language Descriptions, Tools, and Applications (LDTA '06 and '07).

[21] Karel Müller. Attribute-directed top-down parsing. In *Compiler Construction*, pages 37–43. Springer, 1992.

[22] Sebastian Erdweg, Tillmann Rendel, Christian Kästner, and Klaus Ostermann. Layout-sensitive generalized parsing. In *Software Language Engineering*, pages 244–263. Springer, 2013.

[23] Paul Klint and Eelco Visser. Using filters for the disambiguation of context-free grammars. In *Proc. ASMICS Workshop on Parsing Theory*, pages 1–20. Citeseer, 1994.

[24] Masaru Tomita. Graph-structured stack and natural language parsing. In *Proceedings of the 26th Annual Meeting on Association for Computational Linguistics*, ACL '88, pages 249–257, Stroudsburg, PA, USA, 1988. Association for Computational Linguistics. doi: 10.3115/982023.982054. URL http://dx.doi.org/10.3115/982023.982054.

[25] Görel Hedin. Reference attributed grammars. *Informatica (Slovenia)*, 24(3):301–317, 2000.

[26] Ali Afroozeh, Mark van den Brand, Adrian Johnstone, Elizabeth Scott, and Jurgen Vinju. Safe specification of operator precedence rules. In *Software Language Engineering*, pages 137–156. Springer, 2013.

[27] Alex P. ten Brink. Disambiguation mechanisms and disambiguation strategies. Master's thesis, Eindhoven University of Technology, the Netherlands, 2013.

[28] Annika Aasa. Precedences in specifications and implementations of programming languages. *Theoretical Computer Science*, 142(1):3 – 26, 1995. ISSN 0304-3975. doi: http://dx.doi.org/10.1016/0304-3975(95)90680-J. URL `http://www.sciencedirect.com/science/article/pii/030439759590680J`. Selected Papers of the Symposium on Programming Language Implementation and Logic Programming.

[29] `https://docs.python.org/2/reference/expressions.html`.

[30] Bram van der Sanden. Parse Forest Disambiguation. Master's thesis, Eindhoven University of Technology, the Netherlands, 2014.

Appendices

Appendix A

$$(\Gamma_{27}) \qquad \mathbf{EXPRESSION} ::= \mathbf{EXPR} \sim$$
$$\texttt{@pre(0)} : \texttt{attributes.put(“priority”}, 0);$$
$$\sim;$$
$$\mathbf{EXPR} ::= \text{“}\mathbf{if}\text{”} \quad \mathbf{EXPR} \quad \text{“ : ”} \quad \mathbf{EXPR} \quad \mathbf{ELSEIFEXPR} \quad \mathbf{ELSEEXPR} \sim$$
$$\texttt{@pre(all)} : \texttt{if((int)attribute(0, “priority”)} > 10)\{\texttt{return;}\}$$
$$\texttt{@pre(0)} : \texttt{attributes.put(“priority”}, 10);$$
$$\texttt{@pre(1)} : \texttt{attributes.put(“priority”}, 10);$$
$$\texttt{@pre(2)} : \texttt{attributes.put(“priority”}, 10);$$
$$\texttt{@pre(3)} : \texttt{attributes.put(“priority”}, 10);$$
$$\sim;$$
$$\mathbf{ELSEIFEXPR} ::= \text{“}\mathbf{elseif :}\text{ ”} \quad \mathbf{EXPR} \quad \mathbf{ELSEIFEXPR}\ |;$$
$$\mathbf{ELSEEXPR} ::= \text{“}\mathbf{else :}\text{ ”} \quad \mathbf{EXPR} \quad \mathbf{ELSEEXPR}\ |;$$
$$\mathbf{EXPR} ::= \mathbf{EXPR} \quad \text{“}\mathbf{or}\text{”} \quad \mathbf{EXPR} \sim$$
$$\texttt{@pre(all)} : \texttt{if((int)attribute(0, “priority”)} > 20)\{\texttt{return;}\}$$
$$\texttt{@pre(0)} : \texttt{attributes.put(“priority”}, 20);$$
$$\texttt{@pre(1)} : \texttt{attributes.put(“priority”}, 20);$$
$$\sim;$$
$$\mathbf{EXPR} ::= \mathbf{EXPR} \quad \text{“}\mathbf{and}\text{”} \quad \mathbf{EXPR} \sim$$
$$\texttt{@pre(all)} : \texttt{if((int)attribute(0, “priority”)} > 30)\{\texttt{return;}\}$$
$$\texttt{@pre(0)} : \texttt{attributes.put(“priority”}, 30);$$
$$\texttt{@pre(1)} : \texttt{attributes.put(“priority”}, 30);$$
$$math\sim;$$
$$\mathbf{EXPR} ::= \text{“}\mathbf{not}\text{”} \quad \mathbf{EXPR} \sim$$
$$\texttt{@pre(all)} : \texttt{if((int)attribute(0, “priority”)} > 30)\{\texttt{return;}\}$$
$$\texttt{@pre(0)} : \texttt{attributes.put(“priority”}, 30);$$
$$\sim;$$
$$\mathbf{EXPR} ::= \mathbf{EXPR} \quad \text{“}\mathbf{in}\text{”} \quad \mathbf{EXPR} \sim$$
$$\texttt{@pre(all)} :$$
$$\texttt{if((int)attribute(0, “priority”)} > 40)\{\texttt{return;}\}$$

if((int)attribute(0, “priority”) == 40&&attribute(0, “branch”).equals(“r”)){return; }
@pre(0) : attributes.put(“priority”, 40); attributes.put(“branch”, “l”);
@pre(1) : attributes.put(“priority”, 40); attributes.put(“branch”, “r”);
∼;

EXPR ::= **EXPR** **“notin”** **EXPR** ∼
@pre(all) :
if((int)attribute(0, “priority”) > 40){return; }
if((int)attribute(0, “priority”) == 40&&attribute(0, “branch”).equals(“r”)){return; }
@pre(0) : attributes.put(“priority”, 40); attributes.put(“branch”, “l”);
@pre(1) : attributes.put(“priority”, 40); attributes.put(“branch”, “r”);
∼;

EXPR ::= **EXPR** **“is”** **[EXPR** ∼
@pre(all) :
if((int)attribute(0, “priority”) > 40){return; }
if((int)attribute(0, “priority”) == 40&&attribute(0, “branch”).equals(“r”)){return; }
@pre(0) : attributes.put(“priority”, 40); attributes.put(“branch”, “l”);
@pre(1) : attributes.put(“priority”, 40); attributes.put(“branch”, “r”);
∼;

EXPR ::= **EXPR** **“isnot”** **EXPR** ∼
@pre(all) :
if((int)attribute(0, “priority”) > 40){return; }
if((int)attribute(0, “priority”) == 40&&attribute(0, “branch”).equals(“r”)){return; }
@pre(0) : attributes.put(“priority”, 40); attributes.put(“branch”, “l”);
@pre(1) : attributes.put(“priority”, 40); attributes.put(“branch”, “r”);
∼;

EXPR ::= **EXPR** “ < ” **EXPR** ∼
@pre(all) :
if((int)attribute(0, “priority”) > 40){return; }
if((int)attribute(0, “priority”) == 40&&attribute(0, “branch”).equals(“r”)){return; }
@pre(0) : attributes.put(“priority”, 40); attributes.put(“branch”, “l”);
@pre(1) : attributes.put(“priority”, 40); attributes.put(“branch”, “r”);
∼;

EXPR ::= **EXPR** “ <= ” **EXPR** ∼
@pre(all) :
if((int)attribute(0, “priority”) > 40){return; }
if((int)attribute(0, “priority”) == 40&&attribute(0, “branch”).equals(“r”)){return; }
@pre(0) : attributes.put(“priority”, 40); attributes.put(“branch”, “l”);
@pre(1) : attributes.put(“priority”, 40); attributes.put(“branch”, “r”);

$\sim$;

EXPR ::= **EXPR** “ >= ” **EXPR** $\sim$

@pre(all) :

if((int)attribute(0, “priority”) > 40){return; }

if((int)attribute(0, “priority”) == 40&&attribute(0, “branch”).equals(“r”)){return; }

@pre(0) : attributes.put(“priority”, 40); attributes.put(“branch”, “l”);

@pre(1) : attributes.put(“priority”, 40); attributes.put(“branch”, “r”);

$\sim$;

EXPR ::= **EXPR** “ <> ” **EXPR** $\sim$

@pre(all) :

if((int)attribute(0, “priority”) > 40){return; }

if(attribute(0, “branch”).equals(“r”)){return; }

@pre(0) : attributes.put(“priority”, 40); attributes.put(“branch”, “l”);

@pre(1) : attributes.put(“priority”, 40); attributes.put(“branch”, “r”);

$\sim$;

EXPR ::= **EXPR** “! = ” **EXPR** $\sim$

@pre(all) :

if((int)attribute(0, “priority”) > 40){return; }

if((int)attribute(0, “priority”) == 40&&attribute(0, “branch”).equals(“r”)){return; }

@pre(0) : attributes.put(“priority”, 40); attributes.put(“branch”, “l”);

@pre(1) : attributes.put(“priority”, 40); attributes.put(“branch”, “r”);

$\sim$;

EXPR ::= **EXPR** “ == ” **EXPR** $\sim$

@pre(all) :

if((int)attribute(0, “priority”) > 40){return; }

if((int)attribute(0, “priority”) == 40&&attribute(0, “branch”).equals(“r”)){return; }

@pre(0) : attributes.put(“priority”, 40); attributes.put(“branch”, “l”);

@pre(1) : attributes.put(“priority”, 40); attributes.put(“branch”, “r”);

$\sim$;

EXPR ::= **EXPR** “ | ” **EXPR** $\sim$

@pre(all) : if((int)attribute(0, “priority”) > 50){return; }

@pre(0) : attributes.put(“priority”, 50);

@pre(1) : attributes.put(“priority”, 50);

$\sim$;

EXPR ::= **EXPR** “ $\wedge$ ” **EXPR** $\sim$

@pre(all) : if((int)attribute(0, “priority”) > 60){return; }

@pre(0) : attributes.put(“priority”, 60);

@pre(1) : attributes.put(“priority”, 60);

∼;

EXPR ::= **EXPR** “&” **EXPR** ∼

@pre(all) : if((int)attribute(0, “priority”) > 70){return; }

@pre(0) : attributes.put(“priority”, 70);

@pre(1) : attributes.put(“priority”, 70);

∼;

EXPR ::= **EXPR** “ << ” **EXPR** ∼

@pre(all) :

if((int)attribute(0, “priority”) > 80){return; }

if((int)attribute(0, “priority”) == 80&&attribute(0, “branch”).equals(“r”)){return; }

@pre(0) : attributes.put(“priority”, 80); attributes.put(“branch”, “l”);

@pre(1) : attributes.put(“priority”, 80); attributes.put(“branch”, “r”);

∼;

EXPR ::= **EXPR** “ >> ” **EXPR** ∼

@pre(all) :

if((int)attribute(0, “priority”) > 80){return; }

if((int)attribute(0, “priority”) == 80&&attribute(0, “branch”).equals(“r”)){return; }

@pre(0) : attributes.put(“priority”, 80); attributes.put(“branch”, “l”);

@pre(1) : attributes.put(“priority”, 80); attributes.put(“branch”, “r”);

∼;

EXPR ::= **EXPR** “ + ” **EXPR** ∼

@pre(all) :

if((int)attribute(0, “priority”) > 90){return; }

if((int)attribute(0, “priority”) == 90&&attribute(0, “branch”).equals(“r”)){return; }

@pre(0) : attributes.put(“priority”, 90); attributes.put(“branch”, “l”);

@pre(1) : attributes.put(“priority”, 90); attributes.put(“branch”, “r”);

∼;

EXPR ::= **EXPR** “ − ” **EXPR** ∼

@pre(all) :

if((int)attribute(0, “priority”) > 90){return; }

if((int)attribute(0, “priority”) == 90&&attribute(0, “branch”).equals(“r”)){return; }

@pre(0) : attributes.put(“priority”, 90); attributes.put(“branch”, “l”);

@pre(1) : attributes.put(“priority”, 90); attributes.put(“branch”, “r”);

∼;

EXPR ::= **EXPR** “/” **EXPR** ∼

@pre(all) :

if((int)attribute(0, “priority”) > 100){return; }

if((int)attribute(0, “priority”) == 100&&attribute(0, “branch”).equals(“r”)){return; }

```
@pre(0) : attributes.put(“priority”, 100); attributes.put(“branch”, “l”);
@pre(1) : attributes.put(“priority”, 100); attributes.put(“branch”, “r”);
~;
EXPR ::= EXPR   “//”   EXPR ~
@pre(all) :
if((int)attribute(0, “priority”) > 100){return; }
if((int)attribute(0, “priority”) == 100&&attribute(0, “branch”).equals(“r”)){return; }
@pre(0) : attributes.put(“priority”, 100); attributes.put(“branch”, “l”);
@pre(1) : attributes.put(“priority”, 100); attributes.put(“branch”, “r”);
~;
EXPR ::= EXPR   “ * ”   EXPR ~
@pre(all) :
if((int)attribute(0, “priority”) > 100){return; }
if((int)attribute(0, “priority”) == 100&&attribute(0, “branch”).equals(“r”)){return; }
@pre(0) : attributes.put(“priority”, 100); attributes.put(“branch”, “l”);
@pre(1) : attributes.put(“priority”, 100); attributes.put(“branch”, “r”);
~;
EXPR ::= EXPR   “%”   EXPR ~
@pre(all) :
if((int)attribute(0, “priority”) > 100){return; }
if((int)attribute(0, “priority”) == 100&&attribute(0, “branch”).equals(“r”)){return; }
@pre(0) : attributes.put(“priority”, 100); attributes.put(“branch”, “l”);
@pre(1) : attributes.put(“priority”, 100); attributes.put(“branch”, “r”);
~;
EXPR ::= “ + ”   EXPR ~
@pre(all) :
if((int)attribute(0, “priority”) > 110){return; }
@pre(0) : attributes.put(“priority”, 110);
~;
EXPR ::= “ − ”   EXPR ~
@pre(all) :
if((int)attribute(0, “priority”) > 110){return; }
@pre(0) : attributes.put(“priority”, 110);
~;
EXPR ::= EXPR   “ * *”   EXPR ~
@pre(all) :
if((int)attribute(0, “priority”) > 120){return; }
if((int)attribute(0, “priority”) == 120&&attribute(0, “branch”).equals(“r”)){return; }
```

```
@pre(0) : attributes.put(“priority”, 120); attributes.put(“branch”, “l”);
@pre(1) : attributes.put(“priority”, 120); attributes.put(“branch”, “r”);
~;
EXPR ::= EXPR “[” EXPR “ : ” EXPR “]” ~
@pre(0) : attributes.put(“priority”, 0);
@pre(1) : attributes.put(“priority”, 0);
@pre(2) : attributes.put(“priority”, 0);
~;
EXPR ::= EXPR“[”EXPR“]” ~
~;
EXPR ::= EXPR “(” EXPR EXPRS “)” ~
@pre(0) : attributes.put(“priority”, 0);
@pre(1) : attributes.put(“priority”, 0);
~| EXPR“(” “)”;
EXPR ::= “(” EXPR EXPRS “)” ~
@pre(0) : attributes.put(“priority”, 0);
@pre(1) : attributes.put(“priority”, 0);
~| “()”;
EXPR ::= “[” EXPR EXPRS “]” ~
@pre(0) : attributes.put(“priority”, 0);
@pre(1) : attributes.put(“priority”, 0);
~| “[]”;
EXPR ::= “” KVP KVPS “” ~
@pre(0) : attributes.put(“priority”, 0);
@pre(1) : attributes.put(“priority”, 0);
~| “”;
EXPRS ::= “,”EXPR EXPRS |;
KVP ::= ATOM“ : ”ATOM;
KVPS ::= “,”KVP KVPS |;
EXPR ::= ATOM “[.]” IDENTIFIER;
IDENTIFIER ::= [a − zA − Z][a − zA − Z0 − 9]∗ ~
@post(all) :
int before = currentInputPosition;
int after = scanner.getPosition();
String parsed = scanner.subSequence(before, after);
if(parsed.equals(”True”)){return; }
if(parsed.equals(”False”)){return; }
if(parsed.equals(”for”)){return; }
```

$$\texttt{if}(\texttt{parsed.equals}(\text{"}\texttt{while}\text{"}))\{\texttt{return};\}$$

$$\sim;$$

$$\mathbf{EXPR} ::= \mathbf{ATOM};$$

$$\mathbf{ATOM} ::= \mathbf{NAT} \mid \mathbf{BOOLEAN} \mid \mathbf{STRING} \mid \mathbf{IDENTIFIER};$$

$$\mathbf{NAT} ::= [\mathbf{1-9}][\mathbf{0-9}]* \mid \text{``}\mathbf{0}\text{''};$$

$$\mathbf{BOOLEAN} ::= \text{``}\mathbf{True}\text{''} \mid \text{``}\mathbf{False}\text{''};$$

$$\mathbf{STRING} ::= [\backslash\text{''}][\mathbf{a-zA-Z}]*[\backslash\text{''}];$$