Eindhoven University of Technology

MASTER

Gtext : a language workbench based on GLL and term rewriting

Afroozeh, A.

Award date:
2012

Link to publication

Eindhoven University of Technology

Department of Mathematics and Computer Science

Software Engineering and Technology Group

Master's Thesis

$\mathcal{G}$text

A Language Workbench based on GLL and Term Rewriting

Ali Afroozeh

August 24, 2012

Supervisor:

Prof. dr. Mark van den Brand

I dedicate this work to my parents, for their encouragement and support at every step during my Master's studies.

Abstract

One of the main difficulties in developing domain-specific languages is the use of deterministic parsing technologies in language workbenches. These deterministic parsers do not naturally support modularity, and impose restrictions on defining the grammar of a language. For example, LL(k) parsers cannot deal with left recursion or ambiguities, which have to be removed by rewriting the grammar. These modifications usually lead to a grammar definition which is not readable and may cause maintenance problems. More importantly, the parse trees from a modified grammar may be significantly different from the ones of the original, ambiguous grammar. This may cause problems in processing parse trees, for example, for mapping to EMF models.

In this thesis, we present a new language workbench based on the GLL parsing algorithm. GLL is able to parse the full class of context-free grammars without any limitation and thus inherently supports modularity. Being a generalized parser, GLL produces a parse forest containing all the ambiguities. Our disambiguation method is based on pattern matching and rewriting within the resulting parse forest. One of the motivations for our disambiguation mechanism is solving hard parsing problems such as language embeddings and extensions. In addition, we present an error reporting mechanism for GLL-based parsers, an Eclipse plugin for developing new languages, and facilities for the mapping of ambiguous, complex concrete syntax to EMF models.

Contents

List of Figures

List of Tables

List of Listings

viii

Chapter 1

Introduction

Language workbenches [20] are tools that facilitate the development of domain-specific languages (DSLs). These tools provide an environment for editing the syntax of DSLs and can even generate a development environment for DSLs. In addition to syntax editing, many language workbenches also provide means for disambiguation, type checking, parse tree processing, and mapping to meta-modeling frameworks such as EMF [5].

The first step in language processing is parsing, and the choice of parsing technology significantly influences a language workbench's capabilities and usage. One of main difficulties in developing domain-specific languages is the use of deterministic parsing technologies in language workbenches. These deterministic parsers do not naturally support modularity, and impose restrictions on defining the grammar of a language. For example, LL(k) parsers cannot deal with left recursion and ambiguities, which have to be removed by rewriting the grammar. These modifications usually lead to a grammar definition which is not readable and may lead to maintenance problems. More importantly, the parse trees from a modified grammar may not be as natural as the ones of the original ambiguous grammar. This hinders the processing of a parse tree, for example, for mapping to arbitrary EMF models.

Furthermore, deterministic parsers cannot be used to solve hard parsing problems, such as language embeddings and extensions, at least not easily. First, the class of deterministic context-free languages is not closed under union [22], meaning that the union of two deterministic languages might not be deterministic. Therefore, even if one designs LL(k) or LR(k) syntax for two languages, there is no guarantee that the resulting language of combining them be deterministic. Second, different languages may share tokens, which may lead to ambiguities in language compositions. These ambiguities can not be easily resolved by deterministic parsers.

Current language workbenches can be divided in two categories. In the first group there are Xtext [13] and EMFText [6], which are based on ANTLR [1], a deterministic LL(k) parsing technology. EMFText and Xtext have the mentioned problems for deterministic parsers, thus they are not suitable for creating an environment for language extensions, for example. However, they provide good support, within the limitations of LL(k) parsing, for binding concrete syntax to EMF. The second category is composed of ASF+SDF [34] successors. These tools are Rascal [11] and Spoofax [12] which are based on SGLR [38], a generalized parsing technique. As generalized parsers support the full class of context-free grammars, these tools are powerful enough for language experimentation and building parsers for embedded languages, but they provide no support for EMF mapping.

During the last 30 years, more efficient implementations of generalized parsers have become available. Since the algorithm formulated by Tomita [32] there have been a number of generalized LR parsing (GLR) implementations, such as GLR [29], a scannerless variant (SGLR) [38] and Dparser [2]. Johnstone and Scott developed a generalized LL (GLL) parser [31] in which the function call stack in a traditional recursive descent parser is replaced by a structure similar to the stack data structure (Graph Structured Stack) used in GLR parsing. GLL parsers are particularly interesting because their implementations are straightforward and efficient.

In this thesis we present $\mathcal{G}$text, a prototype of a language workbench based on GLL and Term Rewriting. $\mathcal{G}$text uses a Java implementation of GLL as its underlying parsing technology, and provides mappings to EMF. The GLL parsing algorithm is relatively new, and there is currently no standard disambiguation mechanism for selecting the desired derivation from a set of derivations. A great part of this work is dedicated to developing a generic disambiguation mechanism for GLL parsers. The disambiguation mechanism is based on term rewriting within the parse forest produced from a GLL parser, without modifications in the parser. As we show in Chapter 6, the disambiguation mechanism can be used to deal with complex ambiguities, thus making $\mathcal{G}$text a suitable tool for language experimentation and developing parsers for embedded languages.

The research questions we answer in this thesis are as follows:

1. How can one select a derivation from a set of ambiguous derivations produced by a GLL parser? *i.e.*, how to disambiguate a parse forest produced by GLL.

2. How to report parse errors from GLL parsers?

3. Can our disambiguation mechanism, together with GLL, be used to solve complex ambiguities arising in programming languages and language embeddings?

4. How can GLL be integrated into the Eclipse platform [3] to build GLL-based Integrated Development Environments (IDEs)?

This rest of this thesis is organized as follows.

Chapter 2, Preliminaries, presents a set of preliminary topics, providing a technical basis for the rest of this thesis. The preliminary topics include an overview of context-free grammars, the GLL parsing algorithm, and the Tom language.

Chapter 3, MEBNF Syntax Formalism, presents a modular EBNF formalism, MEBNF, for describing the syntax of context-free grammars. MEBNF is the basis for our parser generator presented in Chapter 8.

Chapter 4, Error Reporting in Generalized LL Parsers, describes an error reporting mechanism for generated GLL parsers.

Chapter 5, Disambiguation by Rewriting within the Parse Forest, the core part of this thesis, presents our disambiguation mechanism, which is based on term rewriting within GLL's parse forest, in detail.

Chapter 6, Disambiguation by Example, presents a number of complex ambiguities in programming languages and island grammars [26]. Island grammars are particularly suitable for implementing parsers for language embeddings.

Chapter 7, Experiments, presents three case studies for the evaluation of our disambiguation mechanism.

Chapter 8, JGLL Parser Generator, describes the architecture and implementation of our Java-based GLL parser generator, JGLL.

Chapter 9, The Eclipse Plugin, is dedicated to the integration of GLL to the Eclipse platform.

Chapter 10, EMF Model Generation, incorporates the EMF mapping introduced in previous work [24] to $\mathcal{G}$text. In this chapter, we show how to map ambiguous grammars written in EBNF to meta-models.

Chapter 11, Future Work, discusses the current issues in this work, and propose research paths for future work.

Chapter 12, Conclusions, provides a conclusion to this work.

Chapter 2

Preliminaries

2.1 Context-free Grammars

A *context-free grammar* (CFG) consists of a set $\mathbf{N}$ of nonterminals, a set $\mathbf{T}$ of terminals, a set of grammar rules of the form $A ::= \alpha$ where $A \in \mathbf{N}$ and α is a string in $(\mathbf{T} \cup \mathbf{N})^*$, and a specified start symbol, $S \in \mathbf{N}$. The symbol ϵ denotes the empty string. Rules with the same left hand side can be written as a single rule using the alternation symbol, $A ::= \alpha_1 \mid \ldots \mid \alpha_t$. The strings α_j are the *alternates of A*.

A grammar Γ defines a set of strings of terminals, its *language*, as follows. A *derivation step*, has the form $\gamma A \beta \Rightarrow \gamma \alpha \beta$ where $\gamma, \beta \in (\mathbf{T} \cup \mathbf{N})^*$ and $A ::= \alpha$ is a grammar rule. A *derivation* of τ from σ is a sequence $\sigma \Rightarrow \beta_1 \Rightarrow \beta_2 \Rightarrow \ldots \Rightarrow \beta_{n-1} \Rightarrow \tau$, also written $\sigma \overset{*}{\Rightarrow} \tau$. A derivation is *left-most* if at each step the left-most nonterminal is replaced. The language defined by Γ is the set of $u \in \mathbf{T}^*$ such that $S \overset{*}{\Rightarrow} u$. A grammar is *ambiguous* if there is a string u for which there is more than one left-most derivation $S \overset{*}{\Rightarrow} u$.

The role of a *parser* for Γ is, given a string $u \in \mathbf{T}^*$, to find (all) the derivations $S \overset{*}{\Rightarrow} u$. In our case the derivations will be recorded as trees. A *derivation tree* in Γ is an ordered tree whose root node is labeled with the start symbol, whose interior nodes are labeled with nonterminals and whose leaf nodes are labeled with elements of $\mathbf{T} \cup \{\epsilon\}$. The children of a node labeled A are ordered and labeled with the symbols, in order, from an alternate of A. The *yield* of the tree is the string of leaf node labels in left to right order. A grammar Γ is ambiguous if and only if there are two or more derivation trees in Γ with the same yield.

The FIRST set of the symbol X is the set of all tokens which can appear as the first token derivable from X. The FIRST set can be computed from the grammar as follows:

- If X is a terminal, then $\text{FIRST}(X) = \{X\}$.

- Let $Y := X_1 X_2 ... X_n$ be a production rule. Then, $\text{FIRST}(X_1) - \{\epsilon\}$ is in $\text{FIRST}(Y)$, and if $X_1 \overset{*}{\Rightarrow} \epsilon$, $\text{FIRST}(X_2) - \{\epsilon\}$ is also in $\text{FIRST}(Y)$ and so on. In the general case, if $Y := X_1 X_2 ... X_{j-1} X_j ... X_n$, then $\text{FIRST}(X_i)$, $1 \leq i \leq j-1$, expect for ϵ are in $\text{FIRST}(Y)$. If $X_1 X_2 ... X_{j-1} \overset{*}{\Rightarrow} \epsilon$, $\text{FIRST}(X_j) - \{\epsilon\}$ is also in $\text{FIRST}(Y)$.

- If $X \overset{*}{\Rightarrow} \epsilon$, then ϵ is in the $\text{FIRST}(X)$.

The FOLLOW set of a nonterminal X determines the set of terminals which can immediately appear after X in sentential forms derivable from the start symbol. The FOLLOW set can be computed as:

- Let S be the start symbol of the grammar, then \$, denoting the end of input symbol, is in $\text{FOLLOW}(S)$.

- For the production rule $A ::= \alpha B \beta$, in which α is a list of zero or more symbols and β is a list of one or more symbols, $\text{FIRST}(\beta)$ except for the empty symbol, ϵ, is in $\text{FOLLOW}(B)$. Moreover, if $\text{FIRST}(\beta)$ contains ϵ, then $\text{FOLLOW}(A)$ is in $\text{FOLLOW}(B)$.

- For the production rule $A ::= \alpha B$, FOLLOW(A) is in the FOLLOW(B).

The TEST set of a nonterminal X is the same as the $\text{FIRST}(X)$, and if $\epsilon \in \text{FIRST}(X)$, then $\text{FOLLOW}(X)$ is also in $\text{TEST}(X)$. The TEST set of a nonterminal is used by parsers to select an alternate of the nonterminal based on the current token from the input.

2.2   Generalized LL Parsing

Top down parsers whose execution closely follows the structure of the underlying grammar are attractive, particularly because they make grammar debugging easier. GLL is a top down parsing technique which is fully general, allowing even left recursive grammars, which has worst-case cubic runtime order and which is close to linear on most 'real' grammars. In this section we give a basic description of the technique, a full formal description can be found in [31].

A GLL parser effectively traverses the grammar using the input string to guide the traversal. There may be several traversal threads, each of which has a pointer into the grammar and a pointer into the input string. For each nonterminal A there is a block of code corresponding to each alternate of A. At each step of the traversal, (i) if the grammar pointer is immediately before a terminal we attempt to match it to the current input symbol; (ii) if it is immediately before a nonterminal B then the pointer moves to the start of

the block of code associated with B; (iii) if it is at the end of an alternate of A then it moves to the position immediately after the instance of A from which it came. This control flow is essentially the same as for a classical recursive descent parser in which the blocks of code for a nonterminal X are collected into a *parse function* for X with traversal steps of type (ii) implemented as a function call to X and traversal steps of type (iii) implemented as function return. In classical recursive descent, we use the runtime stack to manage actions (ii) and (iii) but in a general parser there may be multiple parallel traversals arising from nondeterminism; thus in the GLL algorithm the call stack is handled directly using a Tomita-style graph structured stack (GSS) which allows the potentially infinitely many stacks arising from multiple traversals to be merged and handled efficiently.

Multiple traversal threads are handled using process descriptors, making the algorithm parallel rather than backtracking in nature. Each time a traversal bifurcates, the current grammar and input pointers, together with the top of the current stack and associated portion of the derivation tree, are recorded in a descriptor. The outer loop of a GLL algorithm removes a descriptor from the set of pending descriptors and continues the traversal thread from the point at which the descriptor was created. When the set of pending descriptors is empty all possible traversals will have been explored and all derivations (if there are any) will have been found.

We define a *grammar slot* to be a position immediately before or after any symbol in any alternate and the grammar pointer points to a grammar slot. We use the familiar LR(0)-style notation, $X ::= x_1 \ldots x_i \cdot x_{i+1} \ldots x_q$ denotes the slot before the symbol x_{i+1}. During a traversal, in the case where x_{i+1} is a nonterminal, the 'return' slot $X ::= x_1 \ldots x_{i+1} \cdot x_{i+2} \ldots x_q$ is added to the GSS and the pointer is moved to some slot of the form $x_{i+1} ::= \cdot \delta$.

Of course, there may be more than one such slot. The *selector set* associated with $x_{i+1} ::= \cdot \delta$ is the set of terminals a such that there is some derivation $S \overset{*}{\Rightarrow} v x_{i+1} \beta \Rightarrow v \delta \beta \overset{*}{\Rightarrow} vau$, where $v, u \in \mathbf{T}^*$. (If $\beta, \delta \overset{*}{\Rightarrow} \epsilon$ then the end-of-string symbol is also included in the selector set.) A slot is chosen only if the current input symbol belongs to its selector set, but even so there may be more than one choice of slot. Thus, when the traversal reaches the start or the end of an alternate the possible traversal continuations are recorded in *context descriptors*. These are 4-tuples (t, i, u, w) where t is the current grammar position (slot), i is the current input pointer position, u is the node on top of the current return-slot stack of the GSS and w is the root of the current derivation subtree. Details of the creation and processing of the descriptors by a GLL algorithm can be found in [31] and a more implementation oriented discussion can be found in [30].

The output of a GLL parser is a representation of all the possible derivations of the input in the form of a *shared packed parse forest* (SPPF), a union
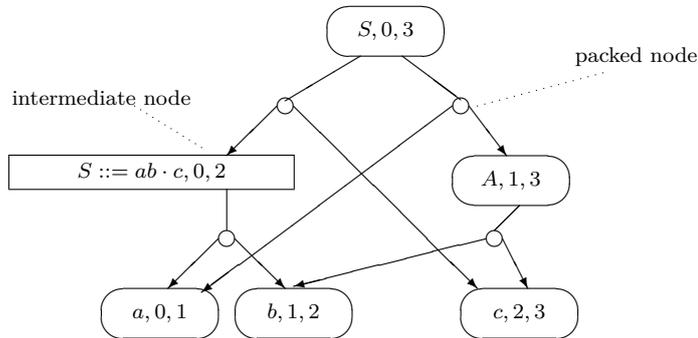
Figure 2.1: SPPF

of all the derivation trees of the input in which nodes with the same tree below them are shared and nodes which correspond to different derivations of the same substring from the same nonterminal are combined by creating a packed node for each family of children. To make sure that descriptors contain only one tree node, the root of the current subtree, the SPPFs are binarized in the natural left associative manner, with the two left-most children being grouped under an intermediate node, which is in turn then grouped with the next child to the left, etc. It is this binarization that keeps both the size of the SPPF and the number of descriptors worst-case cubic.

In detail, a binarized SPPF has three types of SPPF nodes: symbol nodes, with labels of the form (x, j, i) where x is a terminal, nonterminal or ϵ and $0 \leq j \leq i \leq n$; intermediate nodes, with labels of the form (t, j, i); and packed nodes, with labels for the form (t, k), where $0 \leq k \leq n$ and t is a grammar slot. (A grammar slot is essentially an LR(0)-item, a formal definition can be found in [31].) Terminal symbol nodes have no children. Nonterminal symbol nodes, (A, j, i), have packed node children of the form $(A ::= \gamma \cdot, k)$ and intermediate nodes, (t, j, i), have packed node children with labels of the form (t, k), where $j \leq k \leq i$. A packed node has one or two children, the right child is a symbol node (x, k, i), and the left child, if it exists, is a symbol or intermediate node, (s, j, k). For example, for the grammar $S ::= abc \mid aA, \quad A ::= bc$ we obtain the SPPF as shown in Figure 2.1. As is clear from this example, for ambiguous grammars there will be more than one derivation.

2.3 Tom in a Nutshell

Tom [16, 27] is a language based on rewriting calculus and is designed to integrate term rewriting and pattern matching facilities into general-purpose programming languages (GPLs) such as Java, C, C++, Python or Caml.

Tom relies on the Formal Island framework [17], meaning that the underlying host-language does not need to be parsed in order to compile Tom constructs.

The basic functionality of Tom is pattern matching through the `%match` construct. This construct can be seen as a generalization of the *switch-case* construct of many GPLs. The `%match` construct is composed of a set of rules where the left-hand side is a *pattern*, *i.e.,* , a tree that may contain variables, and the right-hand side an *action*, given by a Java block that may in turn contain Tom constructs.

A second construct provided by Tom is the backquote (`` ` ``) term. Given an algebraic term, *i.e.,* , a tree, a backquote term builds the corresponding data structure by allocating and initializing the required objects in memory.

A third component of the Tom language is a formalism named Gom to describe algebraic data structures by means of the `%gom` construct. This corresponds to the definition of an inductive types in classical functional programming. There are two main ways of using this formalism: the first one is defining an algebraic data type in Gom and generating Java classes which implement the data type. This is similar to the EMF [5], in which a Java implementation can be generated from a meta-model definition. The second approach assumes that a data structure implementation, for example in Java, already exists. Then one can define an algebraic data type in Gom and provide a *mapping* to connect the algebraic type to the existing implementation.

Listing 2.1 illustrates a simple example of a Tom program. The program starts with a Java class definition. The `%gom` construct defines an algebraic data type with one module, `Peano`. The module defines a sort `Nat` with two constructors: `zero` and `suc`. The constructor `suc` takes a variable `n` as a field. Given a signature, a well-formed and well-typed term can be built using the backquote (`` ` ``) construct. For example, for `Peano`, `` `zero() `` are `` `suc(zero()) `` are correct terms, while `` `suc(zero(),zero()) `` or `` `suc(3) `` are not well-formed and not well-typed, respectively.

```
public class PeanoExample {

%gom {
 module Peano
 Nat = zero() | suc(n: Nat)
}

 public Nat plus(Nat t1, Nat t2) {
  %match(t1,t2) {
   x,zero() -> { return `x; }
   x,suc(y) -> {
     return `suc(plus(x,y));
   }
  }
 }

 boolean greaterThan(Nat t1, Nat t2) {
 %match(t1,t2) {
   x,x          -> { return false; }
   suc(_),zero() -> { return true;  }
   zero(),suc(_) -> { return false; }
   suc(x),suc(y) -> {
          return `greaterThan(x,y); }
}}

public final static void main(String[] args) {
  Nat N = `zero();
  for(int i=0 ; i<10 ; i++) { N = `suc(N); }
}
```

Listing 2.1: An example of a Tom program

In the example of Listing 2.1, the `plus()` and `greaterThan()` methods are implemented by pattern matching. The semantics of pattern matching in Tom is close to the *match* that exists in functional programming languages, but in an imperative context. A `%match` is parametrised by a list of subjects, *i.e.*, , expressions evaluated to ground terms, and contains a list of rules. The left-hand side of the rules are patterns built upon constructors and new variables, without any restriction on linearity (a variable may appear twice, as in `x,x`). The right-hand side is a Java statement that is executed when the pattern matches the subject. Using the backquote construct (`` ` ``) a term can be created and returned. Similar to the standard `switch/case` construct, patterns are evaluated from top to bottom. In contrast to the functional *match*, several actions (*i.e.*, right-hand side) may be fired for a given subject as long as no `return` or `break` is executed.

In addition to the syntactic matching capabilities illustrated above, Tom also supports more complex matching theories such as matching modulo associativity, associativity with neutral element, and associativity-commutativity.

Chapter 3

Modular EBNF (MEBNF) Syntax Formalism

This chapter presents the MEBNF syntax formalism for defining context-free grammars. MEBNF provides means for defining modular grammars in EBNF. The version of the GLL algorithm we are currently using does not natively support EBNF. Therefore, we perform an EBNF to BNF conversion step prior to generating a GLL parser. In this chapter, we first give an abstract syntax for EBNF, in Section 3.1, and its conversion to BNF, in Section 3.2, with which our GLL parser generator works. We introduce the concrete syntax of the MEBNF formalism in sections 3.3 and 3.4. We conclude this chapter by briefly discussing modularity in context-free grammars and its implementation in MEBNF in Section 3.6.

3.1 An Abstract Syntax for EBNF

Listing 3.1 introduces an abstract syntax for EBNF. The abstract syntax is implemented in Gom. For implementation details, the reader is referred to Chapter 8. As can be seen, a context-free grammar, represented by `Grammar`, consists of a name, a start symbol, a set of context-free rules, and a set of lexical rules. Lists in Gom are defined with the `*` operator. For example, `concSymbol(Symbol*)` is a constructor for defining a list of `Symbols`. The name `conc...` is chosen to comply with Gom's naming convention, and it stands for a concatenation of types. This Gom module imports a Lexical module containing the abstract syntax of lexical rules. We discuss the lexical syntax in detail in Section 3.4.

```
module EBNF
imports String char Lexical
abstract syntax

Grammar = Grammar(name:String, startSymbol:Symbol,
                  contextFreeRules:ContextFreeRuleList,
                  lexicalRules:LexicalRuleList)

ContextFreeRuleList = concContextFreeRule(ContextFreeRule*)

LexicalRuleList = concLexicalRule(LexicalRule*)

ContextFreeRule = ContextFreeRule(head:Symbol, body:SymbolList)

SymbolList = concSymbol(Symbol*)

Symbol = Nonterminal(name:String)
       | Terminal(name:String)
       | Star(symbol:Symbol)
       | Plus(symbol:Symbol)
       | Opt(symbol:Symbol)
       | Alt(first:SymbolList, second:SymbolList)
       | Group(symbols:SymbolList)
```

Listing 3.1: The algebraic definition describing the abstract syntax of EBNF

In the current version of Gom, a constructor's parameters can only be types, defined on the left hand side and not other constructors. This is the reason that we defined a context-free rule as `ContextFreeRule(head:Symbol, ...)` and not `ContextFreeRule(head:Nonterminal, ...)`, although based on the definition in Section 2.1, the head of a context-free rule can only be a nonterminal. In the automatic translation of MEBNF's concrete syntax to the abstract syntax in 3.1, correct constructors are selected to instantiate a Symbol.

Different EBNF symbols are defined as follows:

- `Star(s)`, `Plus(s)`, `Opt(s)` denote, zero or more, one or more, zero or one, occurrences of `s`, respectively, where `s` is a `Symbol`.

- `Alt(l1, l2)` denotes a choice between `l1` and `l2`, where `l1` and `l2` are instances of `SymbolList`, a list of symbols.

- `Group(symbolList)` groups a list of symbols, an instance of `SymbolList`.

3.2   EBNF to BNF Conversion

As mentioned before, the version of the GLL parsing algorithm we are currently using does not natively support EBNF. As part of the parser gener-

ation process, we convert a grammar written in EBNF into an equivalent BNF one. The conversion rules are as follows.

1. Replace a group (α), where α denotes a sequence of symbols, by a nonterminal X, and add the rule $X ::= \alpha$ to the set of context-free rules.

2. Replace context-free rules of the form $X ::= \alpha \mid \beta$, where α and β denote a sequence of symbols, by two rules: $X ::= \alpha$ and $X ::= \beta$.

3. Replace a symbol $X*$ by a nonterminal Y, and add the production rules $Y ::= XY$ and $Y ::= \epsilon$ to the set of context-free rules.

4. Replace a symbol $X+$ by a nonterminal Y, and add the production rules $Y ::= XY$ and $Y ::= X$ to the set of context-free rules.

5. Replace a symbol $X?$ by a nonterminal Y, and add the production rules $Y ::= X$ and $Y ::= \epsilon$ to the set of context-free rules.

We demonstrate the above procedure by converting the production rule $S ::= (A* \mid B (C \mid D))+ E$ to its equivalent BNF. The steps are shown in Table 3.1. Each row in the table shows the production rules at a step of the conversion, the applicable rule, and the symbols being rewritten. The result of each conversion step is shown in its next row. Note that in steps two and three, two different actions are being performed at the same time.

The EBNF to BNF conversion produces intermediate nonterminals which will appear in the parse tree. The presence of these nodes in the parse tree leads to a derivation structure which does not directly correspond to the underlying grammar. Figure 3.1 shows the resulting SPPF from parsing the string `"aaacc"` using a parser for the following grammar:

$$S ::= A*\ B?\ C*$$
$$A ::= a$$
$$B ::= b$$
$$C ::= c$$

In Figure 3.1, nodes with the labels A_*, $B\_?$, and C_* correspond to the intermediary nodes of converting $A*$, $B?$, and $C*$, respectively. This naming convention has been chosen to have similar nonterminal names in EBNF and BNF.

After the parsing is finished, one can traverse the SPPF and remove the intermediary EBNF nodes. Based on the lexical definition of `Nonterminal` in Listing 3.3, the specific EBNF symbols, such as `"*"`, `"+"`, `"?"`, `"("`, `")"`, cannot appear in the name of a symbol node in the SPPF. Thus, any symbol

#	Production Rules	Rule	Action
0	$S ::= (A* \mid B \ (\ C \mid D \ ))+ E$	1	$(C \mid D) \to X$
1	$S ::= (A* \mid B \ X)+ E$ $X ::= C \mid D$	1	$(A* \mid B \ X) \to Y$
2	$S ::= Y+ E$ $X ::= C \mid D$ $Y ::= A* \mid B \ X$	2	expand $C \mid D$ expand $A* \mid B \ X$
3	$S ::= Y+ E$ $X ::= C$ $X ::= D$ $Y ::= A*$ $Y ::= B \ X$	3, 4	$Y+ \to W$ $A* \to Z$
4	$S ::= W \ E$ $W ::= Y \ W$ $W ::= Y$ $X ::= C$ $X ::= D$ $Y ::= Z$ $Z ::= A \ Z$ $Z ::= \epsilon$ $Y ::= B \ X$		

Table 3.1: A step-by-step example of EBNF to BNF conversion

node whose label contains one of these characters is an intermediary EBNF node and should be removed. The removal is performed bottom-up and the children of the removed node will be connected to its parent. In addition, nodes having the label ϵ will be removed after the removal of intermediary nodes. Note that EBNF intermediary nodes may be ambiguous, in which case the node should be disambiguated before removal, see Section 5.4. This removal process transforms the SPPF in Figure 3.1 into the SPPF in Figure 3.2.

The SPPF without intermediary EBNF nodes is particularly suitable for list matching. Let us consider that we need to collect all the nodes labeled `A` from the SPPF in in Figure 3.2. We achieve this by a single Tom pattern as follows:
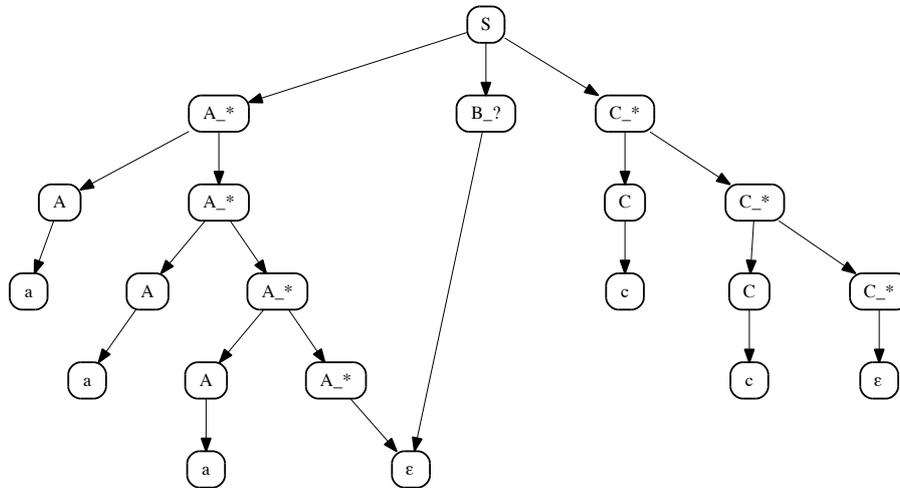
Figure 3.1: The SPPF resulting from the parsing of `"aaacc"` with intermediary EBNF nodes.
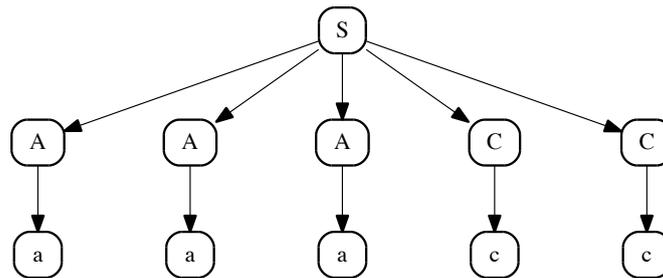
Figure 3.2: The SPPF resulting from the parsing of `"aaacc"` without intermediary EBNF nodes.

```
List<SPPFNode> getAs(SPPFNode node) {
        List<SPPFNode> as = new ArrayList<SPPFNode>();
        %match(node) {
            SymbolNode("S", concNode(_*, a@SymbolNode("A"), _*)) -> {
                as.add(`a);
            }
        }
        return as;
}
```

The `getAs()` method in Listing 3.2 gets an SPPF node and returns its children labeled "A". The body of the match construct is executed for each "A" node. Performing the same task for the SPPF in Figure 3.1, which contains the intermediary EBNF nodes, requires much more effort and is slower.

3.3 A Concrete Syntax for MEBNF

A syntax definition in MEBNF is comprised of four sections: the *context-free* section defining context-free rules, the *lexical* section defining lexical rules, the *disambiguation* section defining disambiguation rules, and the *associativity and priority* section defining the associativity and priorities of arithmetic operators.

```
module MEBNF

import ContextFree
import Lexical
import Disambiguation
import AssociationAndPriority

start symbol Module

context-free syntax

Module ::=  "module" ModuleName Import* StartSymbolSection?  Section*

Import ::= "import" ModuleName Retract*

Retract ::= "retract" (ContextFreeRule | LexicalRule)*

StartSymbolSection ::= "start" "symbol" Nonterminal

Section ::= LexicalSection
        | ContextFreeSection
        | DisambiguationSection
        | AssociationAndPrioritySection
        | LayoutDefinitionSection

lexical syntax

ModuleName ::= Id
Nonterminal ::= Id
Id ::=[a-zA-Z][a-zA-Z0-9\-_]*
```

Listing 3.2: The starting rules of MEBNF's concrete syntax

Listing 3.2 shows the starting rules of MEBNF's concrete syntax, which is written in MEBNF itself. A context-free grammar defined in MEBNF starts with a `module` definition followed by imports and sections. The `retract` keyword in a module definition is used to indicate the rules which should not be imported from the module. The retract mechanism is, however, not implemented in the current version of MEBNF. As can be seen, the MEBNF specification in Listing 3.2 imports four other modules: `ContextFree`, `Lexical`, `Disambiguation`, and `AssociationAndPriority`. In the rest of this

section, we describe the syntax for defining context-free rules, imported from the `ContextFree` module. The other imported modules are discussed in other chapters, where the functionality of the modules are described.

---

module ContextFree

context-free syntax

```
ContextFreeSection ::= "context-free" "syntax" ContextFreeRule*

ContextFreeRule ::= Nonterminal "::=" Symbol* ("|" Symbol*)*

Symbol ::= Symbol "*"
       |   Symbol "+"
       |   Symbol "?"
       |   Keyword
       |   Nonterminal
       |   "(" Alt ")"

Alt ::=  Alt "|" Alt
     |   Symbol+
```

lexical syntax

```
Nonterminal ::= Id
Keyword ::= String
Id ::=[a-zA-Z][a-zA-Z0-9\-_]*
String ::= ["]([^"\\]|[\\]["trnu\\])*["]
```

Listing 3.3: A concrete syntax for defining context-free rules in MEBNF

Listing 3.3 presents the concrete syntax for defining context-free rules. As can be seen, a context-free rule consists of a head, denoted by `Nonterminal`, and a body which is a sequence of symbols. Alternates of a nonterminal can be separated by a `"|"`, instead of repeating the nonterminal definition for each alternate. For example, `S ::= A B | C` defines the nonterminal `S` with two alternates. The first alternate consists of `A B` and the second one of `C`. Note that the `"|"` operator defines an alternation between the longest group of symbols surrounding it. In this example, if the user wants to express an alternation only between `B` and `C`, the alternate `Symbol ::= "(" Alt ")"` should be used, *i.e.*, `S ::= A (B | C)`. To sum up, `"|"` inside a group definition, *i.e.*, inside opening and closing parentheses, always denotes an alternation between the two surrounding sequence of symbols, whereas `"|"` outside a group always separates alternates for a nonterminal.

The grammar definition in Listing 3.3 is ambiguous. For example `(A | B | C)` can be interpreted as either `((A | B) | C)` or `(A | (B | C))`. This ambiguity can be resolved by defining the `"|"` operator left associative, *i.e.*, , selecting `((A | B) | C)` over `(A | (B | C))`. Disambiguation by assigning associativity to

16

operators is explained in detail in Section 5.3.

3.4 Lexical Syntax

In MEBNF every lexical definition is a regular expression which is defined as follows:

- A character "c" is represented by itself.

- "." is a special character matching all characters.

- Special characters, which have a specific meaning in regular expressions, should be escaped using a backslash. These characters are ".", "[", "]", "", "-", and "\".

- In addition to special characters, whitespace characters should also be escaped. The supported whitespace characters are "\ ", "\n", "\r", "\t".

- A character range is represented by c_1-c_2 where c_1 and c_2 are characters, and c_1 is before c_2 in the unicode character ordering. This is because the lexical definitions will be encoded into a Java Lexer implementation, and Java strings are represented in the unicode encoding. An example of a character range is 1-9 denoting the characters from 1 to 9.

- A character class is defined as $[r_1r_2...r_n]$ where each r_i is either a character or a character range. A character class represents a choice between its enclosing elements, *i.e.*, one of its elements should match. The character class is the smallest building block of in our lexical definition. All characters or character ranges should be expressed inside character classes.

- $[\^r_1r_2...r_n]$ defines the negation of a character class, meaning that none of r_is should match.

- If α and β are regular expressions, $\alpha\beta$ is a regular expression matching the concatenation of α and β.

- If α and β are regular expressions, $\alpha|\beta$ is a regular expression matching either α or β.

- If α is a regular expression, $\alpha*$ is a regular expression matching zero or more αs.

- If α is a regular expression, $\alpha+$ is a regular expression matching one or more αs.

17

- If α is a regular expression, $\alpha?$ is a regular expression matching zero or one α.

- If α is a regular expression, (α) is a regular expression denoting a group of regular expressions containing α.

Some examples of regular expressions defined in the lexical syntax of MEBNF are:

- `[a-zA-Z][a-zA-Z0-9\-_]*` matching a sequence of characters starting with a letter followed by any number of letters, numbers, hyphen or underscore. This lexical pattern corresponds to the definition of identifiers in programming languages such as Java.

- `[\ \r \n \t]+` matching a sequence of whitespace characters.

- `[1-9][0-9]+[\.][0-9]+` matching a floating point number such as `102.34`.

- `["]([^"\\]|[\\]["trn\\])*["]`
matching a string. A string should start and end with a double quote character. In between, all characters with the exception of `"` and `"\"` may appear. The only characters which are allowed to be escaped are explicitly defined: `"`, `t`, `r`, `n`, and `\`.

- `[/][/][^\r \n]*` matching a single line C-style comment. Based on this pattern, all the characters appearing after two `"/"` are matched until a newline character is encountered.

The Gom signature corresponding to the regular expression definition is given in Listing 3.4. The constructor `Any()` corresponds to `"."` matching any character. The constructor `LexNonterminal` is used to compose a lexical definition from other lexical definitions.

```
LexicalRule = LexicalRule(head:Symbol, pattern:RegularExpression,
                          exlcusions:RegularExpressionList)

RegularExpressionList = concRegularExpression(RegularExpression*)

RegularExpression = CharClass(list:CharClassTypeList)
                  | NotCharClass(notlist:CharClassTypeList)
                  | LexAlt(lhs:RegularExpression, rhs:RegularExpression)
                  | LexStar(re:RegularExpression)
                  | LexPlus(re:RegularExpression)
                  | LexOpt(re:RegularExpression)
                  | LexGroup(re:RegularExpression)
                  | LexNonterminal(name:String)
                  | Keyword(text:String)
                  | Concat(re1:RegularExpression, re2:RegularExpression)
                  | Any()

CharClassType = Char(c:char)
              | Escape(c:char)
              | CharRange(first:char, second:char)

CharClassTypeList = concCharClassType(CharClassType*)
```

Listing 3.4: The abstract syntax of MEBNF's lexical definition

context-free syntax

```
LexicalSection ::= "lexical" "syntax" LexicalRule*

LexicalRule ::= Nonterminal "::=" RegularExpression LexicalExclusion?

LexicalExclusion ::= "-/-" "{" RegularExpression
                            ("," RegularExpression)* "}"

RegularExpression ::= RegularExpression RegularExpression
                    | RegularExpression "|" RegularExpression
                    | RegularExpression "*"
                    | RegularExpression "+"
                    | RegularExpression "?"
                    | "(" RegularExpression ")"
                    | Nonterminal | Keyword | CharClass | "."

CharClass ::= "[" Not? Character+ "]"

Character ::= Char
            | EscapedChar
            | WordChar "-" WordChar
            | Digit "-" Digit
```

Listing 3.5: MBNF's concrete syntax for lexical definitions

Listing 3.5 shows the concrete syntax of lexical definitions in MEBNF. As can be seen, the definition is straightforward and closely follows the the definition of regular expressions provided before. The concrete syntax provides three more features than the standard regular expression definition given before. These features are *keywords*, *lexical exclusions*, and the *composition* of lexical definitions.

The basic regular expression definition needs separate character classes for each character. This makes the definition of lexical rules for keywords difficult. For example, one should write `class ::= [c][l][a][s][s]` to express a pattern matching the string `"class"`. We provide a shorthand syntactic sugar for keyword definition. With this new syntax, the lexical rule is written as `class = "class"`. In the abstract syntax of Listing 3.4, a keyword is identified by the constructor `Keyword(text:String)`.

The lexical exclusion mechanism excludes certain patterns to be matched. The set of lexical exclusions is specified by the `"-/-"` operator. For example, in the lexical definition `Id ::= [a-zA-Z][a-zA-Z0-9\-_]* -/- {"class", "import"}`, the recognition of `"class"` and `"import"` as an identifier is disallowed. Lexical exclusions are described as regular expressions. When the lexer matches a pattern, if a match has been found, none of the exclusions should match the matched lexeme, otherwise the lexeme is discarded. The lexical exclusion can be used as a powerful disambiguation mechanism which is described in more detail in Section 5.

A lexical definition in MEBNF can be composed of other lexical definitions. For example one may define the lexical definition for a floating number as follows:

```
Float ::= NonZeroDigit Digit* [\.] Digit+
NonZeroDigit ::= [1-9]
Digit ::= [0-9]
```

The parser generator rewrites the composing definitions before generating a lexer. For example, the final lexical definition for `Float` would be `[1-9][0-9]*[\.][0-9]+`. In composing lexical definitions the head of a lexical definition cannot appear as the first symbol in the body of a lexical definition. For example, the definition `Digit ::= Digit [\.] Digit` is illegal. Moreover, lexical definitions containing direct or indirect cycles are not allowed. Such definitions should be moved to a context-free section as regular expressions are not able to deal with such recursive constructs. It should also be mentioned that the provided concrete syntax for lexical rules is ambiguous. In Section 5.3, we show how to resolve the ambiguities by setting the operators' associativity and priority.

3.5 Layout

We distinguish a special kind of lexical rules and dedicate a separate section, the layout section, to it. The term "Layout", which is borrowed from SDF [21], describes whitespace and comments. In MEBNF, layout symbols are automatically added to the grammar. This frees the user of explicitly adding whitespace and comment nonterminals to context-free rules. An example of a layout definition is given in Listing 3.6. As can be seen, two layout elements are defined. The first definition, `Comment`, corresponds to a single-line C-comment comment, while the second definition corresponds to whitespace characters.

---

layout syntax

```
Comment ::= [/][/][^\n\r]
Whitespace ::= [\ \n \r \t]+
```

Listing 3.6: An example of a layout definition section

Our parser generator inserts the nonterminal `Layout`, after each terminal and before the start symbol of a grammar, before generating a parser for the grammar. To add the `Layout` nonterminal before the start symbol, a nonterminal named `Start` is added to the grammar. The definition for the added `Start` nonterminal is as follows:

```
Start ::= Layout S
```

where `S` is the actual start symbol of the grammar. The body of the `Layout` nonterminal is composed of the lexical heads defined in the layout section. For the example in Listing 3.6, the layout definition is as follows:

```
Layout ::= (Comment | WhiteSpace)*
```

3.6 Modularity

Modular grammars allow a separation of rules into several smaller modules. A module can then import the context-free and lexical-rules from other modules. Modularity in context-free grammars has many benefits. For example, It allows the separation of concerns in grammar definitions and facilitates reuse. However, the semantics of importing rules is not yet well understood. In addition, should imported grammars have different layout definitions, the resulting language might not be able to correctly recognize all sentences of the resulting language.

There are currently two approaches in importing context-free rules from another module. The first approach changes the name of the imported nonterminals to ensure unique names across modules, whereas the second approach imports the nonterminals without rewriting, effectively a textual import of rules. As an example, let us consider the following two grammars.

```
module M1
import M2

context-free syntax
E ::= E "+" E
```

```
module M2


context-free syntax
E ::= E "*" E
```

As can be seen, the module `M1` imports the module `M2`. The results of this importing, using both approaches, are shown below:

```
module M1

context-free rules
E ::= E "+" E
E ::= M2.E
M2.E ::= M2.E "*" M2.E
```

```
module M1


context-free rules
E ::= E "+" E
E ::= E "*" E
```

(a) with rewriting   (b) without rewriting

In the approach with rewriting, (a), an additional rule `E ::= M2.E` is added to provide a bridge for reaching the imported rules. In this scheme, there is no way to reach a nonterminal of the importing module from a nonterminal of an imported module. As a result, for example, one cannot derive the string `1 + 2 * 3 + 4` using the the grammar in (a) because the nonterminal `M2.E` cannot produce a "+". However, a derivation such as `1 * 2 + 3 * 4` is possible. In the approach without rewriting, (b), both derivations are possible. The approach without rewriting is more powerful as it can extend the grammar, but it may also lead to unexpected results, as alternates of all nonterminals with the same name will be merged. In some cases, two nonterminals with the same in two different modules do not necessarily should be merged.

For this thesis, we use the import mechanism which imports all the reachable nonterminals without changing the name of the imported nonterminals. In this scheme, if a nonterminal, appearing in the body of a production rule in the importing module, exists as the head of a production rules in an imported module, all rules reachable from the imported nonterminal are imported. Moreover, only the layout definition in the highest level importing module is considered as the layout for the whole composed grammar. In the future, we will provide a more sophisticated import mechanism, possibly using two different import commands, corresponding to the two import mechanism, to allow the user select how the rules for a nonterminal should be imported.

Chapter 4

Error Reporting in Generalized LL Parsers

Parser generators usually provide means for reporting syntax errors in generated parsers. However, error reporting in generated parsers is usually not satisfactory. Error messages in a generated parser are derived from the syntax of the language, and therefore cannot usually provide good hints on how the error should be fixed. Good error messages often rely on the semantics of a language, *e.g.*, block and statement structures, and are manually added to a parser. In this section, we primarily focus on reporting the location of parsing errors from generated GLL parsers, without trying to provide hints how the error should be fixed. Providing a mechanism to allow the user to customize error messages, based on the semantics of a language, is outside the scope of this thesis.

Detecting the exact location where a parse error is occurred is virtually impossible, even in deterministic parsers. Anyone who has worked with the LaTeX typesetting system has experienced that the reported location of an error is usually not where the error is actually occurred, rather the reported location is where the input could no longer be matched. Deterministic parsers have the *viable-prefix* property [14], meaning that as soon as a prefix of the input has been read which cannot be completed to form a sentence of the language, an error has occurred. In deterministic parsers, the location where the parser cannot match further is reported to the user. As a hint on how to fix an error, the expected token(s) at the error location, derived from the *first* and *follow* sets, is also reported to the user.

Error reporting in generated generalized parsers is even more challenging. In deterministic parsers, only one correct derivation can possibly exist. Therefore, if at one point, the parsing cannot proceed, an error is reported to the user. In contrast, in generalized parsers, many paths are explored in

parallel, and failure in matching the input in one path does not necessarily imply an error.

In GLL parsing, for selecting the location where the matching of input fails, we use a heuristic similar to the *viable-prefix*. We record the locations where the input could no longer be matched, and when the parsing fails, the location with the longest input prefix is reported as the error location.

We record a possible error location in the execution of a GLL parser if one of the following scenarios happen:

- The grammar pointer is immediately before a terminal we attempt to match it to the current input symbol, but no match can be found, *i.e.*, the set of tokens returned from the lexer is empty.

- The grammar pointer is immediately before a nonterminal, but the current input symbol does not belong to the *test* set of the nonterminal.

- The grammar pointer is at the start symbol, but the current input symbol does not belong to the *test* set of the start symbol's alternates, thus no alternate can be selected.

In each of these scenarios no descriptor is added to the descriptors set. If before executing a parsing step in which one of these scenarios happen, the descriptor set is already empty, then this location is a definitive error location. However, if there are descriptors in the descriptors set, not adding a descriptor at this step does not necessarily indicate an error, as the current descriptors may lead to a successful parsing of the input. Nonetheless, we add an error message in all cases, regardless of the state of the descriptors set. This is to ensure we capture the error location with the longest input prefix.

When one of these scenarios happen, a new error object containing the current input location and the grammar pointer is created. If no error object already exits, meaning that no error has occurred before, the newly created error object is recorded as the current error object. If an error object already exists, its location is checked against the location where the new error has occurred. If the new location is farther than or equal to the location of the current error object, the current error object is replaced by the new one. Otherwise, if the new error occurs in a location before the location of the current error object, the new error object is discarded. Based on this procedure, at each point during the parsing, at most one error object exists. If the parsing of an input fails, the location and grammar pointer of the current error object will be reported to the user.

With this error reporting method, we always get the location of a parse error when the parsing fails. However, the location may not be the best location

for diagnosing the error. The benefit of this error reporting technique is that it is fairly straightforward to implement. Based on our experience, this approach produces fairly accurate results for the MEBNF grammar introduced in Section 3.3. In the future, we should investigate the use of more advanced heuristics for improving the error reporting.

Chapter 5

Disambiguation by Rewriting within the Parse Forest

A context-free grammar is ambiguous if it produces a sentence in more than one way, leading to more than one left-most (or right-most) derivation. As different derivations may have different meanings, at some point in language processing, ambiguities should be resolved. The process of selecting a derivation from a set of ambiguous derivations is called disambiguation.

Ambiguity detection in context-free grammars is undecidable [22]. However, for specific kinds of ambiguities, there are known disambiguation solutions. For example, for some grammars, one can eliminate the ambiguity by rewriting a grammar to an equivalent grammar, producing the same language. A classic example of disambiguation by rewriting the grammar is the grammar of arithmetic expressions. A grammar for arithmetic addition is shown in Figure 5.1.

$$E ::= E + E$$
$$E ::= Digit$$

$$E ::= E + T \mid T$$
$$T ::= T \; F$$
$$F ::= Digit$$

Figure 5.1: A grammar for arithmetic addition

As can be seen in Figure 5.1, the original grammar on the left side can be transformed to the grammar on the right side by applying the left-recursion elimination and left factorization. More information on these transformations can be found in [14]. The transformed grammar is not ambiguous, but it is difficult to read. More importantly, its parse trees are not as natural as those of the original grammar.

Nevertheless, disambiguation by rewriting a grammar is not applicable to all cases, as for some grammars no unambiguous, equivalent grammar exists [22]. A grammar for which no equivalent unambiguous grammar exists is called an inherently ambiguous grammar. Even for grammars which are not inherently ambiguous, disambiguation by rewriting grammar rules is not straightforward as there is no standard algorithm for that.

Disambiguation can also be achieved by modifying a parser to disallow the creation of undesired derivations in the first place. This method of disambiguation is popular in deterministic parser generators. Deterministic parsers require that there be at most one chance of action at each step. Therefore, if at a step, multiple actions are possible, the parser should be able to select one of the actions. The selection is usually based on some heuristics embedded in the parser. For example, in Yacc [23], the rules for associativity and priority of operators can be embedded within the generated parsers, so that the parser can resolve shift/reduce conflicts resulting from operator ambiguities. In addition to the heuristics, more lookahead symbols may also be employed to help the parser eliminate choices.

Modifying a parser to cope with ambiguities is usually complex, and the modifications are not reusable for other ambiguity types. More importantly, only resolving some well-known ambiguities, such as operator associativities and priorities, is supported by conventional parser generators. For resolving other complex ambiguities, often a parser should be hand-crafted, which involves significant hacking. This approach is not accepted in a languages workbench whose main purpose is to provide an environment for language prototyping and experimentations. For this, the user should be provided with a disambiguation mechanism which is generic and easy to apply.

Generalized parsers produce a parse forest containing all the ambiguities occurred during the parsing. Packing all the ambiguities in a data structure, a parse forest provides the user with the opportunity to traverse the produced parse forest and select the desired derivation. An example of disambiguation by filtering unwanted derivations in the parse forest is implemented in SDF [21], which is based on Scannerless GLR [38]. SDF provides the prefer and avoid [35] mechanisms to mark nodes in the parse forest which should be kept or removed. These mechanisms, however, do not work in all cases and may result in unpredictable results [28, 34, 36].

In this chapter we describe our disambiguation method which is based on rewriting within the parse forest. In contrast to SDF, which provides mechanisms for filtering the parse forest based on top-level production rules, we provide concrete syntax for describing complete tree structures. Many ambiguities are not resolvable by merely examining the top-level production rule under ambiguity nodes. The main motivation for our disambiguation mechanism is building a generic, powerful disambiguation mechanism which

is able to cope with complex ambiguities.

The rest of this chapter is organized as follows. First, we introduce pattern matching and rewriting within GLL's parse forest in Section 5.1. Then, in Section 5.2, we provide concrete syntax, a wrapper for Tom, in order to make the writing of patterns easier and more intuitive. Section 5.3 provides a Yacc-like notation for assigning associativities and priorities to operators. Section 5.4 is dedicated to the Tom implementation of the disambiguation mechanism. Finally, in Section 5.5, we evaluate our disambiguation mechanism as a term rewriting system.

5.1   Rewriting within the SPPF

As discussed in Section 2.2, an SPPF provides efficient means for representing ambiguities by sharing common subtrees. An ambiguity node in an SPPF is a symbol or an intermediate node having more than one packed node as children. For example, consider a simple grammar for arithmetic addition defined in Listing 5.1.

---

context-free syntax
```
E ::= E "+" E | Digit
```

lexical syntax
```
Digit ::= [1-9]+
```

Listing 5.1: A simple grammar for arithmetic addition

For this grammar, the input string `"1+2+3"` is ambiguous. The corresponding SPPF resulting from parsing this input is presented in Figure 5.2. For the SPPF visualization in this chapter, packed nodes are represented by small circles, intermediate nodes by rectangles, and symbol nodes by rounded rectangles in which the name of the symbol node is written. Moreover, for a more compact visualization, the nodes associated with the recognition of layout are removed from the graphs in this section. The SPPF of Figure 5.2 contains an ambiguity which occurs below the first symbol node labeled E.

While an SPPF is built, intermediate nodes are added for binarization, which is essential for controlling the complexity of GLL parsing algorithm. Furthermore, the version of the GLL algorithm we are using creates packed nodes even when there are no ambiguities. These additional nodes lead to a derivation structure which does not directly correspond to the expected abstract syntax of the grammar from which the SPPF has been created.

After the successful construction of an SPPF, one can traverse the SPPF and remove all packed nodes which are not part of an ambiguity, *i.e.*, the

Figure 5.2: The complete SPPF resulting from parsing `"1+2+3"`.

packed nodes which are the only child. The intermediate nodes which only
have one child, the ones which do not present an ambiguity, can also be
removed. When a node is removed, its children are attached to the parent
of the node. Care should be taken that the order of children in the parent
remains intact. By removing unnecessary packed and intermediate nodes,
the SPPF presented in Figure 5.2 can be reduced to the SPPF in Figure 5.3.

The removal of unnecessary intermediate and packed nodes is not applicable
to SPPFs containing cycles. An SPPF contains a cycle only if its underlying
grammar contains a cycle. Cycles do not affect the language produced by a
grammar and can be removed from the grammar. However, the automatic
removal of cycles may significantly alter the grammar, and is not supported

in MEBNF. Moreover, the EBNF to BNF conversion may also introduce cycles, for example, if there is a nonterminal X producing epsilon, *i.e.*, $X ::= \epsilon$, and $X*$ exists in the grammar. In the future, we shall notify the user of the presence of cycles in a grammar and assist her in removing cycles from the grammar.
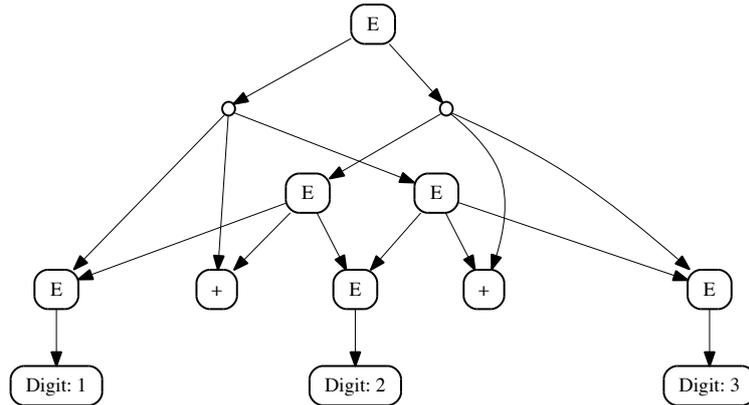
Figure 5.3: The SPPF resulting from parsing `"1+2+3"` without unambiguous intermediate and packed nodes.

For disambiguating an SPPF, we use the pattern matching and rewriting facilities of Tom. Listing 5.2 defines an algebraic data type for SPPF using Gom. The algebraic type defines a sort `SPPFNode` with three constructors, corresponding to three node types present in an SPPF, and the sort `NodeList` defining a list of nodes. The `SymbolNode` constructor creates a symbol node with its name and list of the children. The attribute `name` is the name of a terminal or nonterminal in the grammar. `PackedNode` and `IntermediateNode` constructors create a packed or intermediate node by their list of children. `concNode` constructs a list of `SPPNode` terms. The name `concNode` is selected in compliance with Gom's naming conventions, and it represents the concatenation of nodes as a list.

```
SPPFNode = SymbolNode(name:String, children:NodeList)
         | PackedNode(children:NodeList)
         | IntermediateNode(children:NodeList)

NodeList = concNode(SPPFNode*)
```

Listing 5.2: An algebraic type definition for the SPPF

The ambiguity shown in Figure 5.3 can be resolved by preferring the left-associative derivation, *i.e.*, the input `"1+2+3"` should be interpreted as `(1+2)+3` and not as `1+(2+3)`.

```
SymbolNode("E", concNode(z1*,
  PackedNode(concNode(SymbolNode[name="E"],
    SymbolNode[name="+"],
    SymbolNode("E", concNode(SymbolNode[name="E"],
                             SymbolNode[name="+"],
                             SymbolNode[name="E"])))))
  )), z2*)) -> { SymbolNode("E", concNode(z1*,  z2*)); }
```

Listing 5.3: A rewrite rule in Tom for preferring the left-associative derivation.

Listing 5.3 shows a Tom rewrite rule resolving the ambiguity in Figure 5.3. As can been seen, the packed nodes under the symbol node labeled E are examined, and if there is a right-associative derivation, it will be removed. The reason that we do not directly write a pattern to select the desired derivation is that we do not know about other packed nodes under an ambiguity node. Rewriting a symbol node by selecting a packed node matching the desired pattern may remove other packed nodes which should not be removed. To have a generic disambiguation mechanism, we search for a packed node matching the opposite pattern, *e.g.*, right-associativity, which we call the illegal pattern. The illegal patterns are not desired, regardless of where they appear, and should be removed. In Listing 5.3, z1* and z2* match other packed nodes which should be kept, while the packed node matching the illegal pattern is rewritten to nothing, and hence removed.

For the arithmetic addition example, in Listing 5.1, inputs having more than three digits have another derivation. For example, 1+2+3+4 can also be interpreted as ((1+2)+(3+4)). This derivation, however, is automatically eliminated, as it matches both the left and right associative patterns, considering that (1+2) and (3+4) can be seen as single E.
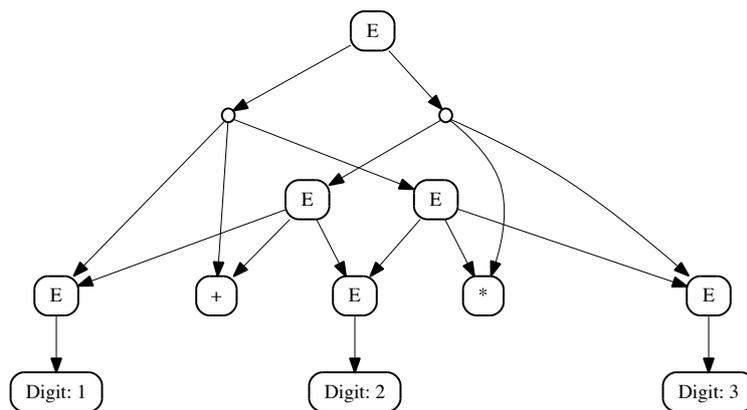
Figure 5.4: The SPPF resulting from parsing 1+2*3

31

We extend the grammar of arithmetic expressions in Listing 5.1 by adding the rule `E ::= E "*" E`. The resulting grammar is ambiguous, for example, for the input `1+2*3`, whose SPPF is shown in Figure 5.4. To resolve this ambiguity, one can assign a higher priority to the `"*"` operator, preferring the derivation `1+(2*3)` to `(1+2)*3`. This means that a `"*"` should appear in a deeper subtree in relation to a `"+"`. The rewrite rule disambiguating this example is shown in Listing 5.4. Note that for defining the precedence relation between the `"+"` and `"*"` operators, one should consider all orderings of the operators. Listing 5.4 shows the two different possible orderings. The first pattern describes a priority relation in which the `"+"` operator appears first, whereas in the second pattern, the `"*"` operator appears first.

```
SymbolNode("E", concNode(z1*,
  PackedNode(concNode(SymbolNode[name="E"],
                      SymbolNode[name="*"],
                      SymbolNode("E", concNode(SymbolNode[name="E"],
                                               SymbolNode[name="+"],
                                               SymbolNode[name="E"])),
                  )), z2*)) -> {
                  SymbolNode("E", concNode(z1*,  z2*)); }

SymbolNode("E", concNode(z1*,
  PackedNode(concNode(SymbolNode("E", concNode(SymbolNode[name="E"],
                                               SymbolNode[name="+"],
                                               SymbolNode[name="E"])),
                      SymbolNode[name="*"],
                      SymbolNode[name="E"]
              )), z2*)) -> {
              SymbolNode("E", concNode(z1*,  z2*)); }
```

Listing 5.4: A rewrite rule in Tom giving `"*"` a higher priority over `"+"`

So far, in the presented examples, we assumed that no layout definition is present in the grammar. Should a grammar have layout definitions, one should incorporate them in rewrite rules. This can be simply achieved by putting a `"_"` after each terminal symbols in the patterns. `"_"` will match the layout symbol after terminal symbols, see Section 3.5 for more information. Considering the layout definition, the rewrite rule of Listing 5.3 is written as in Listing 5.5.

```
SymbolNode("E", concNode(z1*,
  PackedNode(concNode(SymbolNode[name="E"],
    SymbolNode[name="+"], _,
    SymbolNode("E", concNode(SymbolNode[name="E"],
                             SymbolNode[name="+"], _,
                             SymbolNode[name="E"])))))
  )), z2*)) -> { SymbolNode("E", concNode(z1*,  z2*)); }
```

Listing 5.5: The rewrite rule for preferring the left-associativity by taking the layout definition into account

The rewrite rules presented so far can only remove one packed node matching an illegal pattern. As described before, an ambiguous node has more than one packed node. Each rewrite rule matching a packed node reduces the number of packed nodes by one. To fully disambiguate an ambiguity node, a rewrite rule should be applied as long as it can be matched. The disambiguation is successful if only one packed node remains. Then, the children of the remaining packed node replace the packed node. This replacement resolves the ambiguity.

If a rewrite rule cannot disambiguate a node, other available rules should be applied. The disambiguation process fails if after applying all the available rewrite rules, still more than one packed nodes is present under an ambiguity node. The disambiguation process should be applied bottom up to disambiguate the whole SPPF. In many cases, resolving an ambiguity in a higher level in an SPPF depends on first resolving the ambiguities in deeper levels of the SPPF, *i.e.*, subtrees of an ambiguous node in a higher level are ambiguous themselves. By performing the disambiguation bottom up, it is ensured that the subtrees of an ambiguous nodes are not ambiguous, thus can uniquely be specified by a tree pattern.

The successful disambiguation of an SPPF transforms it to a tree only consisting of symbol nodes. Packed and intermediate nodes which are not part of an ambiguity node are removed before applying a rewrite rule, as discussed at the beginning of this chapter. Ambiguous intermediate nodes, having more than one packed node, are removed when they are disambiguated, as part of disambiguating higher level ambiguity nodes. Packed nodes which are part of an ambiguity node are removed one by one by disambiguation rules. The remaining packed node under an ambiguity node is replaced by its children as the last step of disambiguating a node. After all the ambiguities are resolved, no intermediate or packed node can exist under disambiguated nodes. To give the SPPF a unified shape, we traverse the SPPF, now a tree, from the root and remove the rest of packed and intermediate nodes.

The procedure for removing unnecessary packed and intermediate nodes, regardless of where they appear, is as follows:

1. Remove the packed node under a non-ambiguous symbol node. The children of the packed node are attached to the symbol node.

2. Remove the packed node under a non-ambiguous intermediate node. The children of the packed node are attached to the intermediate node.

3. Remove intermediate nodes which are not ambiguous. The children of an intermediate node are attached to its parent (a packed node).

5.2 Concrete Syntax for Disambiguation Rules

Expressing disambiguation rules using the algebraic terms is tedious for the user because there are many technical aspects the user should be aware of. For example, the user should explicitly create a list of symbol nodes using the `concNode` constructor. To overcome these obstacles, we present a higher level syntax for writing disambiguation rules. In the syntax, the user only focuses on the name of symbol nodes in an SPPF, rather than their full algebraic representation.

In principle, the user can write a disambiguation rule removing more than one packed node at once. We, however, do not allow such a rewrite rule in the simplified syntax. In the syntax, the user can either specify that a single packed node matching an illegal pattern should be rewritten to nothing, *i.e.*, to be removed, or a packed node should be preferred over another sibling packed node. We call the former type of rule a *remove* rule and the latter type a *prefer* rule.

The user can always write a remove or prefer rule for an ambiguity node. In an SPPF, ambiguity nodes always contain more than two packed nodes, otherwise there would have not been an ambiguity in the first place. Furthermore, subtrees under the sibling packed nodes are different, otherwise there would would have not been different packed nodes. Given these two facts, there are always two packed nodes with different subtrees available under an ambiguity node, and the user can write a remove rule, describing which one is illegal, or a prefer rule, describing which one is preferred.

Considering this, the simplified syntax is as follows:

- An ambiguous symbol node is specified as `[P₁]` or `[P₁], [P₂]`, where `P₁` and `P₂` are packed nodes. The former case corresponds to a remove rule, whereas the latter case describes prefer rule. For prefer rules, the order in which `P₁` and `P₂` appear, or how many other packed nodes are before, after, or in between does not matter.

- `PackedNode(concNode(t₁, t₂, ..., tₙ))`, a packed node whose children are $t_1, t_2, ..., t_n$ is written as `[t₁, t₂, ..., tₙ]`, where each t_i describes a symbol node. In-

side a packed node, the order of children should be explicitly stated. Moreover, the user does not need to specify the layout nodes, as they are automatically matched.

- `SymbolNode("E",_)` or `SymbolNode[name="E"]`, a symbol node whose children are not important, is represented by `E`, its name.

- `SymbolNode("E", concNode(t`$_1$`, t`$_2$`, ..., t`$_n$`))` where $t_1, t_2, ..., t_n$ are the subterms of the `SymbolNode`, is written as `E(t`$_1$`, t`$_2$`, ..., t`$_n$`)`. In other words, the name of a symbol node becomes the root of the tree.

- `_` matches any symbol node while `_*` matches any sequence of symbol nodes.

In this syntax, no packed node can appear below another packed node. The reason is that the deepest ambiguity node has packed nodes as only its direct children, while the rest of the nodes in its subtrees are symbol nodes. Subtrees under the deepest ambiguous node are either not ambiguous, or haven been previously disambiguated, as the disambiguation process is performed bottom up.

---

context-free syntax

```
DisambiguationSection ::= "disambiguation" "rules" Rules*

Rule ::= "remove" PackedNode
     | "prefer" PackedNode "," PackedNode

PackedNode ::= "[" (SymbolNode | Any) ("," (SymbolNode | Any) )* "]"

SymbolNode ::= SymbolName
            | SymbolName "(" ( SymbolNode | Any) ("," (SymbolNode | Any) )* ")"
```

lexical syntax

```
Any ::= [_][*]?
Name ::= String
```

Listing 5.6: The MEBNF specification for disambiguation rewrite rules

Listing 5.6 illustrates the MEBNF grammar that defines the disambiguation rewrite rules. As can be seen, a disambiguation rule can be either a *remove* or *prefer* rule. A *remove* rule describes a packed node which should be removed, rewritten to nothing, while a *prefer* rule prefers the first packed node to the second one, *i.e.*, if both packed nodes are found, the one which is not preferred is eliminated. With the simplified notation, the rewrite rule in Listing 5.5 can be written as:

```
remove [E, "+", E(E, "+", E)]
```

5.3 Binary and Unary Operators

As we have seen in the above examples, some natural grammars for binary any unary operator expressions are ambiguous. For disambiguation, the user can directly write disambiguation rules, as shown in Section 5.1. However, due to the large number of associativity and priority combinations between operators, it is tedious for the user to manually write all the combinations. In addition, as the need for disambiguating binary and unary operators often occurs in practice, a Yacc-like notation for defining the associativity and priority between operators can be helpful. In this section, we provide a concrete syntax for describing the associativity and priority between the operators, and explain how each combination is translated into the disambiguation rules.

5.3.1 Concrete Syntax for Operator Associativities and Priorities

---

context-free syntax

```
PriorityGroup ::= "{" (AssociativityGroup (">" AssociativityGroup)*)? "}"
AssociativityGroup ::=  (Modifier ":")?
                         OperatorRule ("," OperatorRule)*
OperatorRule ::= BinaryOperatorRule
            | LeftUnaryOperatorRule
            | RightUnaryOperatorRule

BinaryOperatorRule ::= Head "::=" Operand Operator? Operand
LeftUnaryOperatorRule ::= Head "::=" Operator Operand
RightUnaryOperatorRule ::= Head "::=" Operand Operator
Operator ::= String | Id
```

lexical syntax
```
Head ::= Id
Operand ::= Id
Modifier ::= "left" | "right"
```

Listing 5.7: Concrete syntax for the associativity and priority of operators

Listing 5.7 provides a concrete syntax for describing the associativity and priority of operators. As can be seen, a `PriorityGroup` defines a set of priority rules for a nonterminal head. Each priority group is composed of a number of associativity groups. An associativity group defines the associativity for each rule and between the rules it contains. For example, to give the `"*"` operator a higher priority than the `"+"` operator in the grammar of arithmetic expressions, we write:

```
{
  left: E ::= E "*" E >

  left: E ::= E "+" E
}
```

The example above defines a priority group containing two associativity groups. The priority is determined by the greater-than sign, separating associativity groups. The type of associativity is identified by `Modifier`. In case the modifier is not specified, no associativity is applied to the associativity group. In the example above, both associativity groups are left-associative.

The classic notion of associativity is defined for only one binary operator. We, however, generalize this notion to more than one operator, to be able to disambiguate expressions containing operators of the same priority. For example, the user can write `left: E ::= E "+" E, E ::= E "*" E` to express that `+` and `*` are left-associative in respect to each other. Under the hood, for each pair of left-associative binary operators, four Tom rules are generated. One for each operator and one for each ordering of the operators. Note that if the user wants to give an operator a different associativity than the one given for a associativity group the operator belongs to, the operator should be moved to a separate associativity group.

We provide two more examples for illustrating the usage of the associativity and priority syntax. The first example, which is a grammar for arithmetic expressions, is shown in Listing 5.8.

context-free syntax

```
E ::= E "+" E
    | E "-" E
    | E "*" E
    | E "/" E
    | "-" E
    | E "^" E
    | Digit
```

Listing 5.8: A grammar for arithmetic expressions

The associativity and priority of operators in Listing 5.8 is shown in Listing 5.9.

associativity and priority

```
{
right: E ::= E "^" E  >

E ::= "-" E  >

left: E ::= E "*" E,
    E ::= E "/" E  >

left: E ::= E "+" E,
    E ::= E "-" E
}
```

Listing 5.9: The associativity and priority of arithmetic operations

Listing 5.9 defines four associativity groups. The first group gives the power operator right associativity. This production rule has the highest priority as well. The second group has a rule defining the unary negation operator. This rule cannot be left or right associative, thus no modifier is given. The purpose of defining the negation rule in a separate group, without providing an associativity modifier, is only to give it a different priority. The third group defines the priorities of the multiplication and division operators, both left associative themselves and in respect to each other. The same holds for the fourth group, which has the lowest priority. Using the associativities and priorities defined in Listing 5.9, the expression `1--2*3^-4^5*3` is interpreted as `1-((-2)*(3^(-(4^5)))*3)`, whose parse tree is shown in Figure 5.5.

The second example of using associativity and priority rules is resolving the ambiguities in MEBNF's lexical syntax defined in Listing 3.5. The ambiguity is caused by the alternates of `RegularExpression`. The associativity and priority of the operators involved in the ambiguity is specified Listing 5.10.

```
{
  RegularExpression ::= RegularExpression "*",
  RegularExpression ::= RegularExpression "+",
  RegularExpression ::= RegularExpression "?"
>
left: RegularExpression ::= RegularExpression RegularExpression
>
left: RegularExpression ::= RegularExpression "|" RegularExpression
}
```

Listing 5.10: Context-free priorities for `RegularExpression`

As shown in Listing 5.10, the operators `"*"`, `"+"`, and `"?"` should have a higher priority than `"|"` or the concatenation operator in `RegularExpression ::= RegularExpression RegularExpression`. The priority relationship implies

38
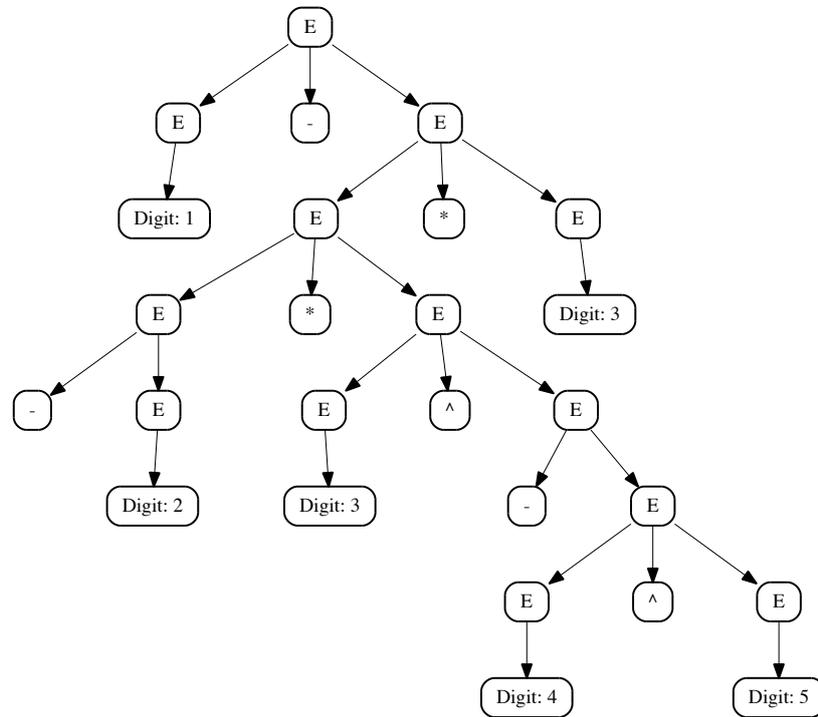
Figure 5.5: The parse tree resulting from parsing `1--2*3^-4^5*3"`

that the expression $XY*$, for example, should be interpreted as $X(Y*)$ and not $(XY)*$. The same holds for the "+" and "?" operators. In addition, the concatenation operator is defined as left-associative, so that the the regular expression XYZ is interpreted as $(XY)Z$ and not as $X(YZ)$. The same hold for the "|" operator. Finally, "|" has a lower priority than the concatenation operator. As a result, the regular expression $X\ Y\ |\ Z$ should be interpreted as $(XY)\ |\ Z$ and not as $X(Y\ |\ Z)$.

5.3.2 Translation to Disambiguation Rules

In this section we give a formal description of the translation of associativity and priority rules to disambiguation rules, which are eventually translated to Tom rules. A production rule describing a binary, left-unary, and right-unary rule is of the form $A ::= A\ x\ A$, $A ::= x\ A$, and $A ::= A\ x$, respectively, where A is a nonterminal, and x is either a terminal, nonterminal. For binary rules, x can be ϵ, defining a concatenation operator. The supported operators and their translation to disambiguation rules are as follows:

- `left: A ::= A x A`, the left associativity of a binary operator is translated to: `remove [A, x, A(A, x, A)]`.

- `left: A ::= A x A, A ::= A y A` , the left associativity of two binary operators is translated to: `remove [A, x, A(A, y, A)]` and `remove [A, y, A(A, x, A)]`. Note that both operators are left-associative themselves, *i.e.*, the previous rule applies to both operators.

- `right: A ::= A x A`, the right associativity of a binary operator is translated to `remove [A(A, x, A), x, A]`.

- `right: A ::= A x A, A ::= A y A`, the right associativity of two binary operators is translated to: `remove [A(A, x, A), y, A]` and `remove [A(A, y, A), x, A]`. Note that both operators right-associative themselves, *i.e.*, the previous rule applies to both operators.

- `A ::= A x A > A ::= A y A`, the priority between two binary operators, where `x` has a higher priority than `y` is translated to: `remove [A, x, A(A, y, A)]` and `remove [A(A, y, A), x, A]`.

- `A ::= x A > A ::= A y A`, the priority between a left-unary operator and a binary operator, where the left-unary operator has a higher priority, is translated to: `remove [x, A(A, y, A)]`.

- `A ::= A y A > A ::= x A`, the priority between a binary operator and a left-unary operator, where the binary operator has a higher priority, is translated to: `remove [A(x, A), y, A]`.

- `A ::= A x > A ::= A y A`, the priority between a right-unary operator and a binary operator, where the right-unary operator has a higher priority, is translated to: `remove [A(A, y, A), x]`.

- `A :: A y A > A ::= A x`, the priority between a binary operator and a right-unary operator, where the binary operator has a higher priority, is translated to: `remove [A, y, A(A, x)]`.

5.4 Implementation

In Section 5.1 we described our disambiguation mechanism without exploring the implementation details. Our Java implementation of GLL is not aware of Tom and creates SPPF nodes as pure Java objects. Creating SPPF nodes as Tom terms is difficult, if not impossible. Tom terms are immutable, meaning that after the construction, they can no longer change. A change in a term , in fact, returns a new term. In SPPF construction, however, children are often added to a node without any specific order, which updates the parent node. In addition, there are always many references to SPPF nodes, from GSS for example. Implementing SPPF nodes as immutable objects implies that each time a new child is added to an SPPF node, all references should be updated, and in fact, a complete new SPPF should be created.

Although Tom uses maximum sharing and has a memory efficient strategy for creating a new term from an existing term, updating all the references is neither efficient nor easy.

Nevertheless, for pattern matching using Tom, we need to provide a bridge between SPPF nodes as Java objects and the Gom type defined in Listing 5.2. As a first approach we consider the conversion of an SPPF to a Tom term after the SPPF construction has finished. The conversion is straightforward, and as Tom provides maximum sharing of subterms, the storage of a converted Tom term is memory efficient. The problem is the traversal and rewriting strategies, as Tom terms are trees, and not graphs. When a subterm is being rewritten, the subterm is only rewritten in the current traversing path. If the subterm is shared by other nodes in another path, the other node still has a pointer to the original subterm.

As an Example, consider the SPPF in Figure 5.6 (a), in which both nodes B and C share the node D. From Tom's point of view, the SPPF is seen as in Figure 5.6 (b). Let us now consider the rewrite rule $D \rightarrow E$. Applying the rule to the node D under B results in the SPPF in Figure 5.6 (c). As can be seen, the rewriting has only taken place in the current path, and other node, node C, maintains its original subterm. Rewriting a node in all paths is not feasible as the number paths in SPPFs containing many ambiguity nodes is exponential.

(a) Memory view   (b) Tom view   (c) After rewriting
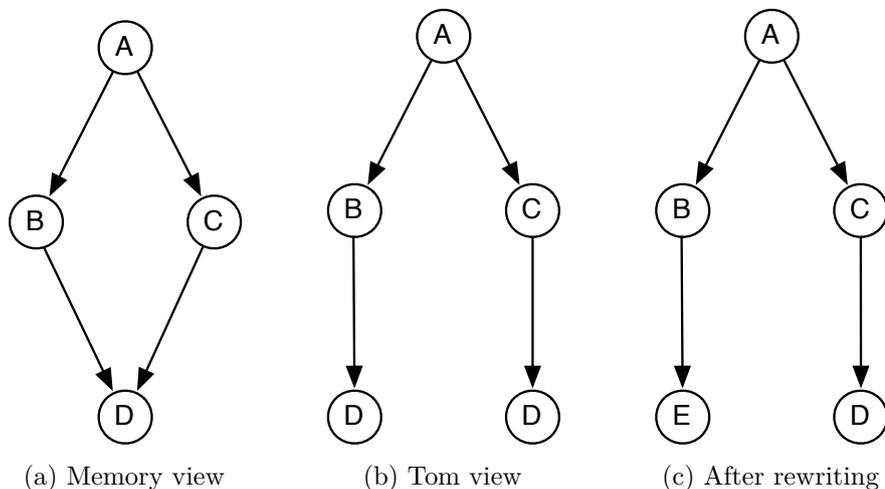
Figure 5.6: Different SPPF representations

The first approach, converting an SPPF after parsing is finished, is only suitable for small SPPFs or SPPFs containing few ambiguities. This makes it virtually unusable in real-world settings. A second approach is to keep the SPPF as Java objects and use a Tom mapping to view the Java structure as a Tom term. The price paid here is performance, as Java's List type

is not as efficient as Gom's generated List type in pattern matching and rewriting. The difference in performance was, however, not noticeable in the whole disambiguation process. The Tom mapping for SPPF is given in Appendix B.

Using a mapping for SPPF, one cannot simply rewrite a term as before, and should manipulate the matched node manually. This is to avoid creating immutable objects. For example, the rewrite rule in Listing 5.3 is changed to the rule in Listing 5.11. As can be seen, instead of creating a new term, the `setChildren` method of the matched node, identified by `node@`, is called in the body of the rule. The last point about the Tom mapping is that Tom strategies do not just simply work with mapping. To be able to use strategies with mapping, an specific *Introspector* should be used. An introspector is a Java object providing Tom strategies with the access to children of a term through a standard interface. The introspector for SPPF is provided in Appendix C.

```
node@SymbolNode("E", concNode(z1*,
  PackedNode(concNode(SymbolNode[name="E"],
    SymbolNode[name="+"],
    SymbolNode("E", concNode(SymbolNode[name="E"],
                             SymbolNode[name="+"],
                             SymbolNode[name="E"])))))
)), z2*)) -> { `node.setChildren(`concNode(z1*, z2*)); }
```

Listing 5.11: A rewrite rule in Tom for preferring the left-associative derivation

The disambiguation rules and patterns presented so far assume that unnecessary packed and intermediate nodes are removed from an SPPF. Moreover, it is assumed that the EBNF to BNF conversion has been performed before the pattern matching. Performing these transformations, both removing unnecessary nodes and EBNF to BNF conversion, on an SPPF containing ambiguities may not be efficient. Due to many shared subtrees under ambiguous nodes, transforming an ambiguous SPPF may lead to the exponential worst case complexity. To overcome this, the removal of unnecessary and intermediary EBNF nodes is performed only under the direct subtrees of the deepest ambiguity node prior to pattern matching. By performing the disambiguation bottom up, we ensure that the transformations are applied only on trees, without sharing, in linear time. To sum up, the disambiguation process is as follows:

1. Find the deepest ambiguity node. An ambiguity node is a symbol or intermediate node having more than one packed node as children.

2. Remove unnecessary packed and intermediate nodes under the am-

biguous packed nodes. Unnecessary packed nodes are the packed nodes which do no represent an ambiguity, *i.e.*, are the only child. Unnecessary intermediate nodes are the intermediate nodes which are not ambiguous, *i.e.*, have only one packed node as child.

3. Perform the EBNF to BNF conversion on the subtree under the ambiguous node. Note that no ambiguous intermediary EBNF node may appear here as the deepest ambiguity node is currently being processed.

4. Mark the transformed nodes so that these nodes are not checked for transformation while resolving ambiguities in higher levels of the SPPF.

5. Select a rewrite rule and apply it as long as it can match a packed node. If a rewrite rule cannot be applied anymore, go to the next rewrite rule. After applying all the available rewrite rules only one packed node should be remained, otherwise the disambiguation process fails.

6. If the disambiguation process is successful, replace the remained packed node with its children. The disambiguation process continues bottom up in an attempt to disambiguate the whole SPPF.

5.5 Disambiguation as a Term Rewriting System

In this section, we introduce the notion of a term rewriting system (TRS) in an informal way, and then examine our disambiguation mechanism as a TRS. A detailed, formal account can be found in [19].

A *term rewriting system* is a set of terms and their subterms, T, and a set of rewrite rules applying on the terms (and their subterms). Terms are usually defined using a signature, *e.g.*, by a formalism such as Gom. A rewrite rule is a function from T to T and is written as $t_i \rightarrow t_j$, stating that the term t_i can be rewritten to t_j. A TRS is *terminating* if the rewriting of terms (and their subterms) using the set of rules can not infinitely continue, *i.e.*, at one point no rewrite rule can be applied on a term anymore. A term which can no loner be rewritten is called a *normal* form. A TRS is *confluent*, if after applying the rules in any arbitrarily order, the same normal form is produced. More formally, if t is a term, and $t \overset{*}{\Rightarrow} u$ and $t \overset{*}{\Rightarrow} w$, where $\overset{*}{\Rightarrow}$ denotes rewriting in zero or more steps, and u and w are in normal forms, then the TRS is confluent if and only if $u = w$.

Our disambiguation mechanism can be considered as a TRS, where each SPPF is a term, and disambiguation rules are its rewrite rules. As discussed before, there are two types of rules: remove rules, written as *remove*(P),

rewriting a packed node matching the pattern P to nothing, and prefer rules, $prefer(P_i, P_j)$, preferring the packed node matching P_i over the packed node matching P_j.

The TRS corresponding to the disambiguation of an SPPF is terminating. In a fully built SPPF, there are at most n^2 symbol and intermediate nodes, each having at most n packed nodes [31]. Considering the worst case, in which all symbol and intermediate nodes except for the leaves are ambiguous, there are at most $n^2 - n$ ambiguity nodes. The number of disambiguation rewrite rules is also finite, thus in the disambiguation, a finite number of rewriting will be performed. Moreover, each rewrite rule which matches a packed node decreases the size of the SPPF, removing one packed node. Unsuccessful rewrite rules, which cannot match a packed node, leave the SPPF intact. However, there are a finite number of rewrite rules, and if they cannot disambiguate an ambiguity node, the disambiguation process prematurely terminates. In a successful disambiguation, ambiguity nodes are disambiguated one by one, until there is no disambiguation node, and the process terminates. In an unsuccessful disambiguation, at least one ambiguity node cannot be disambiguated, which terminates the whole process.

If only *remove* rules are used in disambiguation, the process is confluent. The reason is that the removal rules act independently of each other, thus the order in which they are applied does not matter. In a successful disambiguation of a rule, each packed node should be removed by one of the *remove* rules, regardless of in which order they are applied. In an unsuccessful disambiguation, at least one of the packed nodes cannot be matched. In both cases, the final result is the same.

However, if there are *prefer* rules, the disambiguation may not be confluent. We demonstrate this by the following example grammar:

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

The SPPF corresponding to parsing a single `a` is shown in Figure 5.7.

Let us consider three patterns named A, B, and C matching symbol nodes with the same name, and the rewrite rules $prefer(A, B)$ and $prefer(B, C)$. If, first, $prefer(A, B)$ is applied on the SPPF in Figure 5.7, node B is eliminated and node A remains. As no rules can be applied anymore, the disambiguation fails. However, if first $prefer(B, C)$ is applied, node B remains, and then by applying $prefer(A, B)$, node A remains. This disambiguates the SPPF.

44
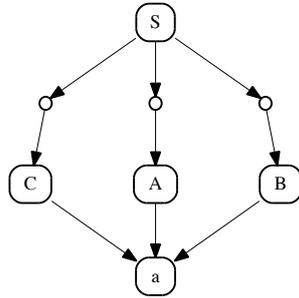
Figure 5.7: An ambiguous SPPF

As shown by this example, the lack of transitivity may lead to successful disambiguation in some orderings of applying rules, but not in some others.

Let us again consider the three patterns, A, B, and C, from the previous example and the rewrite rules $prefer(A, B)$, $prefer(B, C)$, and $prefer(C, A)$. There is a circular relationship between these patterns. Applying the rules $prefer(A, B)$ and $prefer(C, A)$ will result in a derivation tree containing C, but applying $prefer(C, A)$ and $prefer(B, C)$ will result in a derivation tree containing B. This counter example shows that a circular relationship between patterns may lead to different final trees.

Our current conclusion is that if prefer rules are not antisymmetric, not transitive, or there is a circular relationship between them, the disambiguation may not be confluent, but we cannot say with certainty when the disambiguation is confluent, if ever. We shall extend these theoretical analysis in the future.

Chapter 6

Disambiguation by Example

In this chapter we illustrate the applicability of our disambiguation mechanism, by applying it on complex examples. We provide examples of ambiguities in programming languages and island grammars, and show how these ambiguities can be resolved. This chapter shows how hard ambiguity cases can be resolved by our method, thus answering the third research question in Chapter 1.

6.1 Complex Ambiguities in Programming Languages

The examples presented in this section are taken from the SDF Disambiguation Medkit [37].

6.1.1 Postfix Expressions with Guarded Children

---

context-free syntax

```
E ::= E "+" E
  | E  "(" E ")"
  | "e"
```

Listing 6.1: A grammar for postfix expressions

Consider the grammar of postfix expressions in Listing 6.1. For this grammar, the input string `e+e(e+e)` is ambiguous whose associated SPPF is shown in Listing 6.1. The first derivation, `(e+e)"(" (e+e) ")"` the one under the left packed node, is not desired because `e+e` before the opening parenthesis is

reduced to `E`. Based on this pattern `E(E,E)` should bind stronger than `E + E`.
We resolve the ambiguity by defining the illegal pattern as follows:

<div align="center">

`remove [ E(E, "+", E), "(", E, ")"]`

</div>
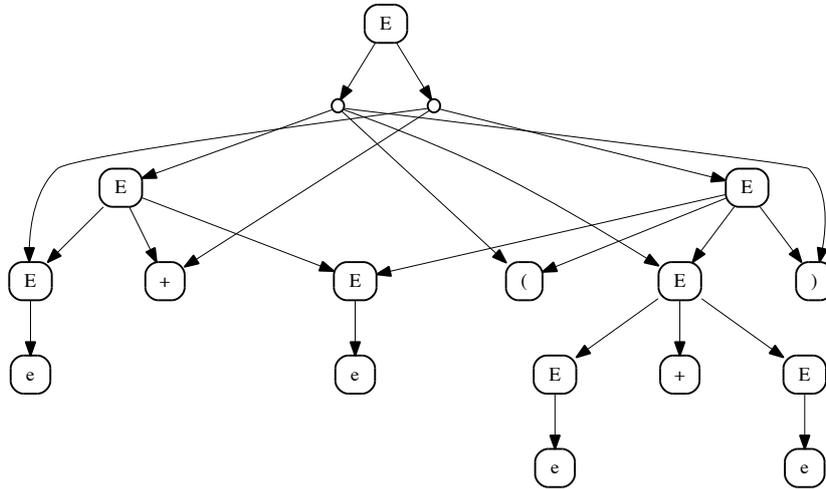
Figure 6.1: The SPPF resulting from parsing `e+e(e+e)`

6.1.2 Overloaded Commas

Consider the expression grammar in Listing 6.2. For this grammar, the
input `f(e,e)` is ambiguous whose associated SPPF is shown in Figure 6.2. As
can be seen, the list of arguments, *i.e.*, `e,e`, can be recognized in two different
ways: as a list of comma-separated `E`s derived from `( E (","  E)* )?` or through
the binary operator `E "," E`. The former derivation should be preferred to the
latter.

---

context-free-syntax

```
E ::= E "," E
    | "f" "(" ( E  (","  E)* )? ")"
    | "e"
```

Listing 6.2: A grammar for expression languages with overloaded commas

A naive way of resolving the ambiguity in Figure 6.2 is to disallow the
presence of a subtree `E(E, ",", E)` inside `f()` by the following rewrite rule:

<div align="center">

`remove ["f"(_*, E(E, ",", E), _*]`

</div>

Although this rewrite rule works for the input `f(e,e)`, it does not work for an
input with more `"e"`s such as `f(e,e,e)`. The reason is that some ambiguities
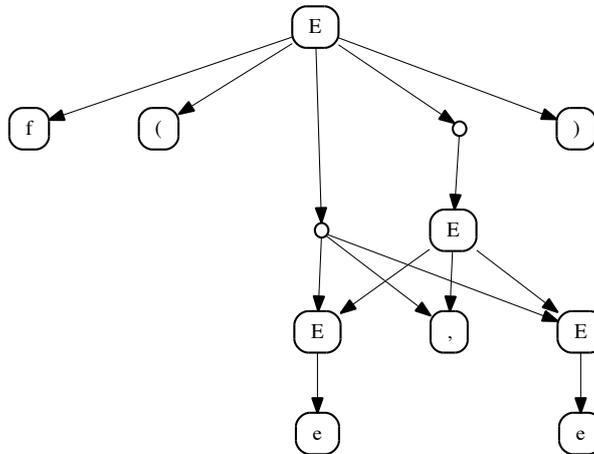
Figure 6.2: The SPPF resulting from parsing `f(e,e)`

happen in deeper nodes, thus the pattern will not apply. To match the deeper ambiguity nodes we should write the pattern without considering `f`. In deeper ambiguity nodes, however, both derivations `E ",", E` and `E (","` `E)*` are valid, but if they both occur under an ambiguous node, one of them should be removed. This ambiguity can be resolved by a preference rewrite rule which selects a derivation from `E (","` `E)*` over the one from `E "," E`:

```
prefer [_*, E(E, ",", E), _*], [_*, E, ",", E, _*]
```

In this example, some packed nodes may have additional children before or after the described pattern. To capture them, we add `_*` before and after the actual pattern.

6.1.3 The Dangling Else Ambiguity

Listing 6.3 shows a grammar for conditional statements. This grammar illustrates a classic ambiguity happening in most programming languages. The ambiguity, which is called "the dangling else" ambiguity, occurs in nested **if-then-else** statements, where the inner **else** can be either interpreted as belonging to the inner **if** or to the outer **if**. For example, consider the statement `"if expr then if expr then other else other"`. The SPPF resulting from parsing this example is shown in Figure 6.3.

```
E::= "expr"
S ::= "if" E "then" S+
    | "if" E "then" S+ "else" S+
    | "other"
```

Listing 6.3: A grammar for conditional statements
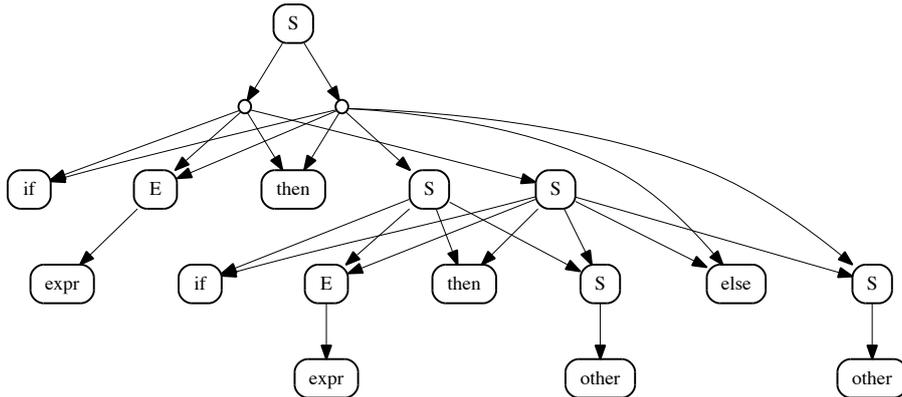
Figure 6.3: The dangling else ambiguity

As can be seen from Figure 6.3, the else statement under the left packed node belongs to the inner if, wheres the else statement under the right packed node belongs to the outer if. The left and right packed nodes correspond to the derivations `"if expr then (if expr then other else other)"` and `"if expr then (if expr then other) else other"`, respectively. Selecting one of these two derivations depends on the semantics of a programming language. For example, in Java, the derivation which binds an else to the closest if statement is selected. The rewrite rule for this selection is shown in Listing 6.4.

```
prefer ["if", E, "then", S],
       ["if", E, "then", S, "else", S]
```

Listing 6.4: Resolving the dangling else ambiguity

The rewrite rule states that if under an ambiguous node, under one packed node, there is an if-then statement, while under another one there is an if-then-else statement, the packed node containing the if- then should be preferred. The presence of if-then implies that the else part is recognized as a part of S, hence it is matched with an inner if statement.

The grammar of conditional statements defined in Listing 6.3 contains another ambiguity which is related to the length of the recognized then part. For example, consider the input `"if expr then if expr then other other"`, whose corresponding SPPF is depicted in Figure 6.4.

49
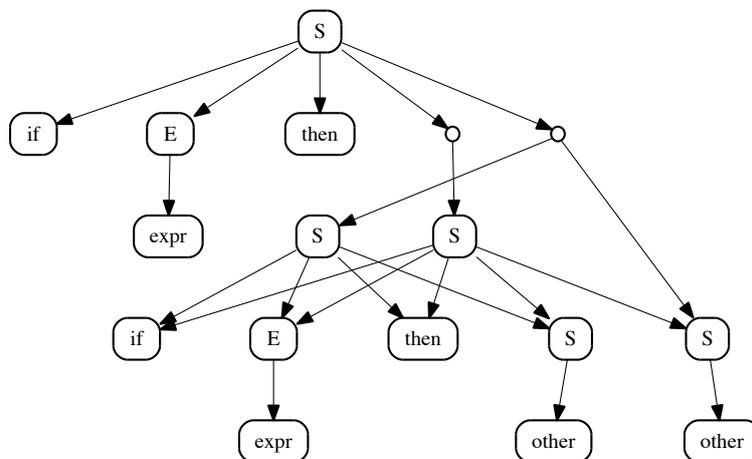
Figure 6.4: The ambiguity in the then part

As can be seen from Figure 6.4, the left packed node corresponds to a derivation in which the last statement, S, is recognized as part of then, *i.e.*, `"if expr then (if expr then (other other))"`. In contrast, the right packed node corresponds to two separate statements, in which the last S constitutes the second statement, *i.e.*, `"if (expr then (if expr then) other) other"`. The ambiguity in Figure 6.4 can be resolved by the rewrite rule shown in Listing 6.5.

```
prefer [S("if", E, "then", S), _*],
       [S("if", E, "then", S, _*)]
```

Listing 6.5: Resolving the ambiguity of the then part

The rewrite rule in Listing 6.5 indicates that if under an ambiguity node there is a packed node containing only one if statement, while under another packed node, the first child is an if statement followed by some other nodes, the packed node containing only one if statement should be preferred. Using the rules in Listings 6.4 and 6.5 we can disambiguate complicated, nested if-then-else statements.

6.1.4 Syntactic Overloading Between Identifiers and Keywords

In languages without reserved keywords, an ambiguity may occur between the identifier and keywords of the language when an identifier takes one of the keywords as its value. As an example, consider the grammar in Listing 6.7.

```
S ::= "return" E;
    | E "(" (E ("," E)*)? ")"

E ::= "(" E ")"
    | Identifier
```

Listing 6.6: A grammar for showing the ambiguity in syntactic overloading

The SPPF corresponding to the parsing of `"return (x)"` is shown in Figure 6.5. As can be seen under the left packed node, `return` is recognized as `E`, while under the right packed node, it is recognized as a keyword `"return"`. The ambiguity in Figure 6.5 can be resolved by a rewrite rule preferring the recognition of keywords, see Listing 6.7.
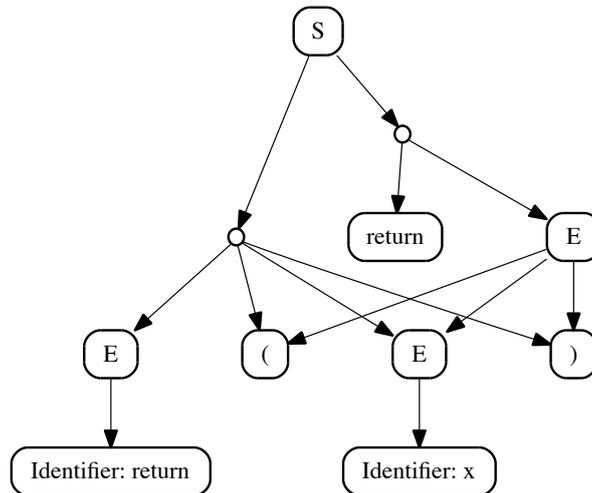
Figure 6.5: The SPPF corresponding to the parsing of `"return (x)"`

```
prefer ["return", E]
       [E, "(", E, ")"]
```

Listing 6.7: preferring the keyword `return` to identifiers

Using pattern matching for resolving this type of ambiguity allows more freedom than having reserved keywords. For example, the keyword `"return"` may be used in other places, but it will be rejected only if there is an ambiguity with recognizing a `"return"` `E` rule.

MEBNF also allows keyword restriction through lexical exclusions mechanism. Using lexical exclusions, keywords can be excluded from being re-

ported by the lexer to the parser as their recognized types. For example, in the example in Listing 6.7, one can exclude the recognition of `"return"` as an `"Identifier"` as follows:

---

lexical syntax

```
Identifier ::= [a-zA-Z_][a-zA-Z0-9_\-]* -/- { "return" }
```

Using the lexical exclusions leads to a faster parser, as the ambiguity is prevented in the first place. However, lexical exclusions reduce the freedom in choosing identifier names.

6.2   Ambiguities in Island-Grammar based languages

Island grammars [26, 33] are a method for describing the syntax of a language, concentrating only on relevant constructs. An island grammar consists of two sets of context-free rules. Rules for *islands* describing the relevant language constructs which should be fully parsed, and rules for *water* which describe the rest of the input which we are not concerned with. One of the applications of island grammars is in the parsing of embedded languages, where the embedded language and host language constructs are captured as island and water, respectively.

To illustrate the use of our disambiguation mechanism in island grammars for parsing embedded languages, we define a simple island grammar which extends general-purpose programming languages. This island grammar, which is a simplified version of Tom's `match` construct, is shown in Listing 6.8. The complete island grammar of Tom is presented in Appendix D.

---

context-free syntax

```
Program ::= Chunk*
Chunk ::= Water | Island
Island ::= "%match" "(" Chunk* ")"
```

lexical syntax

```
Water ::= Identifier | Integer | String | Char | SpecialChar
Identifier ::= [a-zA-Z_]+[a-zA-Z0-9\-_]*
Integer ::= [0-9]+
String ::= ["]([^"\\]|[\\]["trnu\\])*["]
Char ::= ['].['] | ['][\\] [btnfr"'\\] [']
SpecialChar ::= [; : + \- * / = & | < > ! % ? @ \[ \] \^ { } \. $ # ~]
```

Listing 6.8: The starting rules of an island grammar

As shown in Listing 6.8, the production rule defining `Program` contains `Chunk*`, meaning that other `Match` and host-language constructs can be nested inside a `Match` construct. In defining an island grammar, the overlaps and boundaries between the water and island tokens should be recognized. Therefore, we define fine-grained `water` tokens being as small as the building blocks of the supported host languages.

A water definition such as `Water ::= [^ \n \r \t]+`, which defines water as a sequence of non-whitespace characters, is inadequate in recognizing the boundaries. For example, consider the input `"%match(x)"`. Given the lexical definition for `Water`, no **match** construct can be recognized. The lexer reports a sequence of `"match"`, `"("`, and `"x)"` tokens to the parser. After consuming the first parenthesis, the parser asks for a `Water`, but the lexer returns the whole chunk of text until the next whitespace or the end of line character, creating the token `"x)"`. As no closing parentheses left, the parsing fails. For more information about the interaction between the parser and lexer, the reader is referred to Chapter 8.

In addition, the island grammar presented in Listing 6.8 is designed to be host-language agnostic, although it may not possible to completely achieve it. For being host-agnostic, the water definition should be flexible and capture the different varieties of tokens appearing in different supported host languages. For example, `SpecialChar` contains the union of all different symbols which appear in the host languages supported by our island grammar.

6.2.1 The Island-Water Ambiguity

The SPPF corresponding to the parsing of `"%match(x)"` is shown in Figure 6.6. As can be seen, an ambiguity has been occurred between the recognition of an island, under the right packed node, and a set of water, under the left packed node. We call this ambiguity the island-water ambiguity happening in almost all island grammars. If the ambiguity contains more than one water element, the ambiguity occurs under a node labeled `"Chunk_*"`, which is in fact an intermediary node resulting from the EBNF to BNF conversion. For more information on the EBNF to BNF conversion, see Section 3.2. The conversion, however, does not affect the writing of patterns, as patterns describe tree structures under packed nodes and not their parent, the ambiguous node.

The ambiguity in Figure 6.6 can be resolved by the rewrite rule in Listing 6.9. As can be seen, the rewrite rule prefers the packed node having a complete island to another packed node whose first child is a water element. Using this simple rule, we are able to resolve all the island-water ambiguities involving more than one water. In case an ambiguity occurs between an island and a
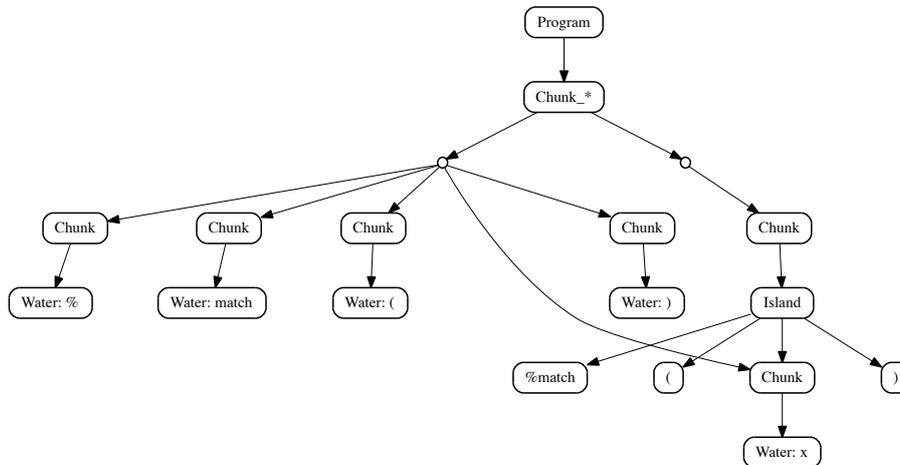
Figure 6.6: The island-water ambiguity

single water, the disambiguation rule would be: `prefer[Island], [Water]`. Note that in this case the ambiguity occurs under a node labeled `Chunk`.

```
prefer [Chunk(Island)],
       [Chunk(Water), _*]
```

Listing 6.9: Resolving the island-water ambiguity

6.2.2 The Island-Island Ambiguity

Figure 6.7 shows the SPPF resulting from the parsing of input `"match(x))"`. The rewrite rule of Listing 6.9 has already been applied to the SPPF, thus the island-water ambiguities are resolved.

As can be seen from Figure 6.7, an ambiguity has been occurred between a collection of islands and water elements under two packed nodes. We call this type of ambiguity the island-island ambiguity. Resolving the island-island ambiguity is difficult, mainly because it depends on the semantics of the language. For example, in Figure 6.7, the derivation under the left packed node is preferred since it has well-balanced parentheses under its island. Unfortunately, one cannot use our disambiguation syntax to check the well-balancedness property. For this, a simple Tom program should be written to manually check this. See Section 5.4 for details on how to attach a handwritten Tom specification into the disambiguation process.

For this specific island-island ambiguity, one can also rewrite the grammar to avoid the ambiguity. The idea is that we enforce the well-balancedness of parentheses in the grammar itself. For this, the definition of `Chunk*` will be

54
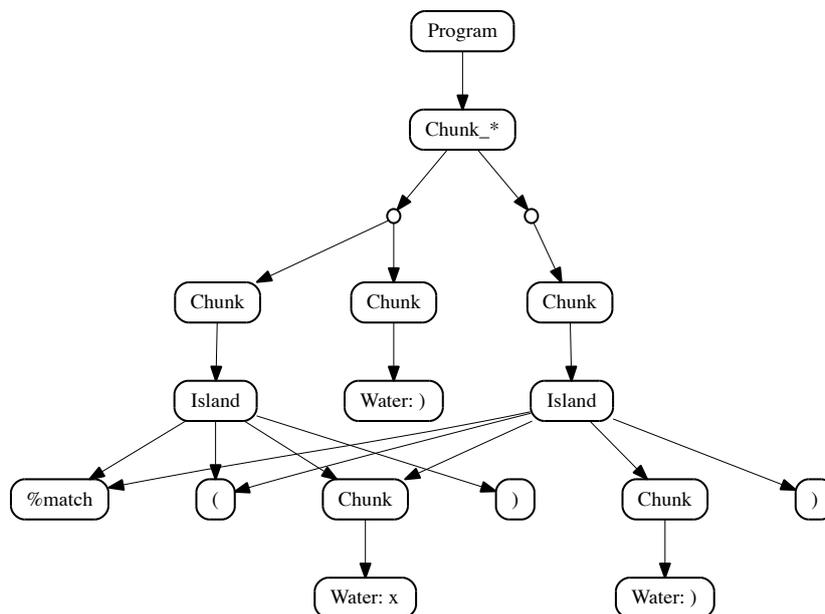
Figure 6.7: The island-island ambiguity

changed to `Chunk* ::= Water | "(" Water ")" | Island`, and parentheses are removed from the lexical definition of water. By these modifications, parentheses can only be recognized as part of an island or `"(" Water ")"`. With the new grammar the input `"%match())"` does not yield any parses as parentheses are not well-balanced in the water parts, but an input such as `"%match() f()"` can be correctly parsed, without introducing island-island ambiguities. The rewriting of the grammar approach is more limiting than pattern matching, as it enforces more restriction on water elements.

A third way of selecting a derivation in the island-island ambiguity is to use a heuristic. For this example, one can select the largest formed island, thus selecting the derivation under the second packed node. The parser of the host language will eventually complain about the unbalanced parenthesis. This way, the island grammar's parser does not assume anything about the host language, and the check for well-balancedness of parentheses is delegated to the host language's parser

A second usage of island grammars is in reverse engineering, especially the fact extraction from source code. Listing 6.10 gives an island grammar definition for extracting integer definitions and assignments from C code. The lexical definitions for `Water`, `Integer`, and `Identifier` is the same as in 6.8. The example is taken from [36].

```
Program ::= Chunk*

Chunk ::= Water | Island

Island ::= "const" "int" Identifier
        | Identifier "=" Integer
```

Listing 6.10: An island grammar for integer definitions and assignments

The SPPF corresponding to the parsing of `const int x = 1` is given in Figure 6.8. The rewrite rule of Listing 6.9 has already been applied to the SPPF, thus the island-water ambiguities are resolved. As shown in [36], SDF's prefer mechanism fails to resolve the island-water ambiguities in this example.
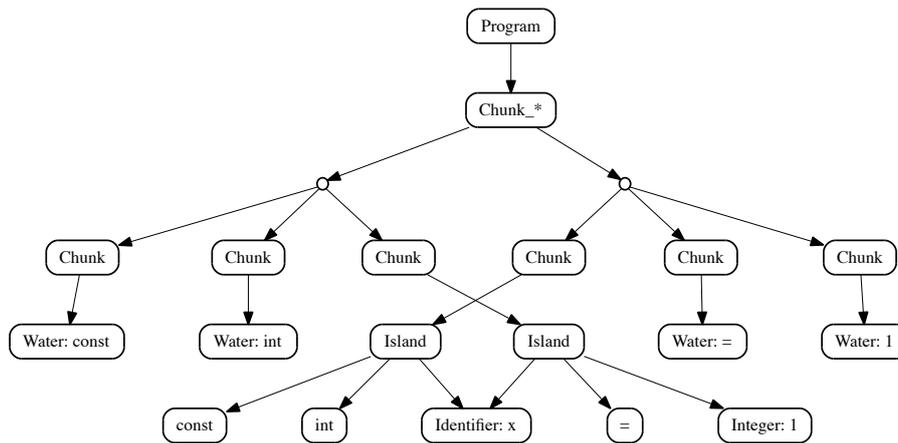
Figure 6.8: The island-island ambiguity resulting from parsing `const int x = 1`

As can be seen from Figure 6.8, under the left packed node, `x = 1` is recognized as an island, while under the right packed node `const int x` is an island. Again, the disambiguation of this island-island ambiguity depends on the semantics. Here, we simply prefer an assignment over a definition. The rewrite rule implementing this preference is shown in Listing 6.11. Using this rewrite rule the ambiguity in Figure 6.8 can be resolved.

```
prefer [_*, Chunk(Island(Identifier, "=", Integer)), _*],
       [_*, Chunk(Island("const", "int", Identifier)), _*]
```

Listing 6.11: Resolving the island-water ambiguity in Figure 6.11

Chapter 7

Experiments

In this chapter we present the result of applying our disambiguation mechanism to three case studies. The first case is the grammar of arithmetic expressions in Listing 5.8 and its associativity and priority relationships in Listing 5.9 as disambiguation rules. The reason for selecting this grammar was that it is fairly ambiguous, and more importantly, and we can easily control the number of ambiguities by increasing the size of input. The second case is mCRL2 [10], a formal language for analysis of behavior of systems. mCRL2 is a very ambiguous language, due to its many operators. The last case is Tom, an embedded language within Java and other general-purpose programming languages, providing advanced pattern matching and tree traversal functionalities. The reason for selecting Tom was to evaluate our disambiguation mechanism on island grammars. Tom has also been used to implement the disambiguation mechanism in this thesis.

7.1 Arithmetic Expressions

For conducting the experiments on the grammar of arithmetic, we generated random arithmetic expressions of different sizes and recorded the disambiguation results. The size of the generated examples ranges from 6 to 340 tokens, and in total 50 examples were executed. Moreover, the associativity and priority rules were translated into 30 removal disambiguation rules.

Table 7.1 summarizes the obtained disambiguation results. The data is divided into 10 categories, each represented by the average value for the metrics. As can be seen, inputs with an average size of 20 are almost immediately parsed and disambiguated. Even with 50 tokens, the total time for parting and disambiguation is acceptable. Longer expressions are extreme cases and rarely happen in practice.

Input size	Parsing Time	Disambiguation Time	(#)Ambiguity Nodes
20	153	70	47
53	514	866	334
88	1102	1455	944
124	1444	3663	1851
157	1852	7136	3001
198	2685	12733	4799
233	3614	18727	6597
263	4633	27816	8366
297	6278	42290	10814
329	8102	58310	13052

Table 7.1: The average disambiguation results. Times are in milliseconds
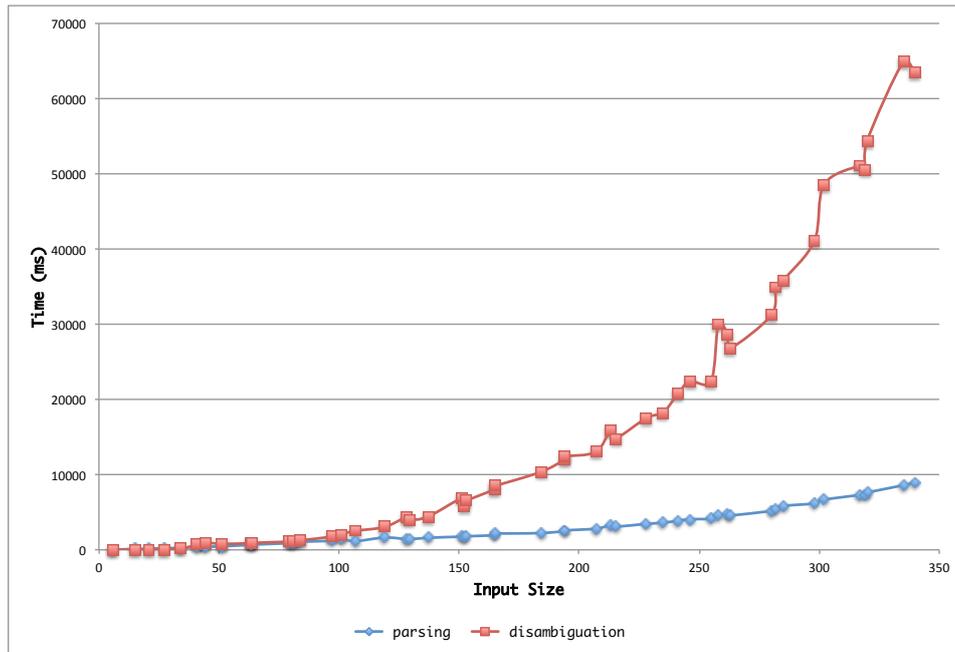
Figure 7.1: The parsing and disambiguation times for arithmetic expressions

The parsing and disambiguation times are plotted in Figure 7.1. As can be seen, as the size of input increases, and as result the number of ambiguity nodes, the disambiguation time increases. In the future, we will present a theoretical analysis of the complexity of our disambiguation mechanism. Moreover, one of the reasons that the parsing time is relatively low is that the grammar only has seven production rules, without any layout rules.

7.2 mCRL2

The second case study is the grammar of mCRL2. The EBNF grammar of mCRL2 can be found at mCRL2's website [10]. The grammar is rather large, with a significant number of operators. Converting the priority and associativity rules has resulted in about 600 Tom rules. Besides, there were a number of "Postfix Expressions with Guarded Children", see Section 6.1.1, ambiguities which have been manually resolved by rewrite rules. Another important point about the grammar of mCRL2 was the presence of unary operators with a prefix or postfix, *i.e.*, $A ::= \alpha\ x\ A$ or $A ::= A\ x\ \alpha$, where α is a sequence of grammar symbols, and x, the operator is is a terminal or nonterminal. The current translation technique in Section 5.3.2 cannot be used to generate rules for such rules. Therefore, the rules concerning the priority of unary and binary operators with a prefix or postfix are written manually. The complete set of disambiguation rules for mCRL2 can be found in Appendix E.
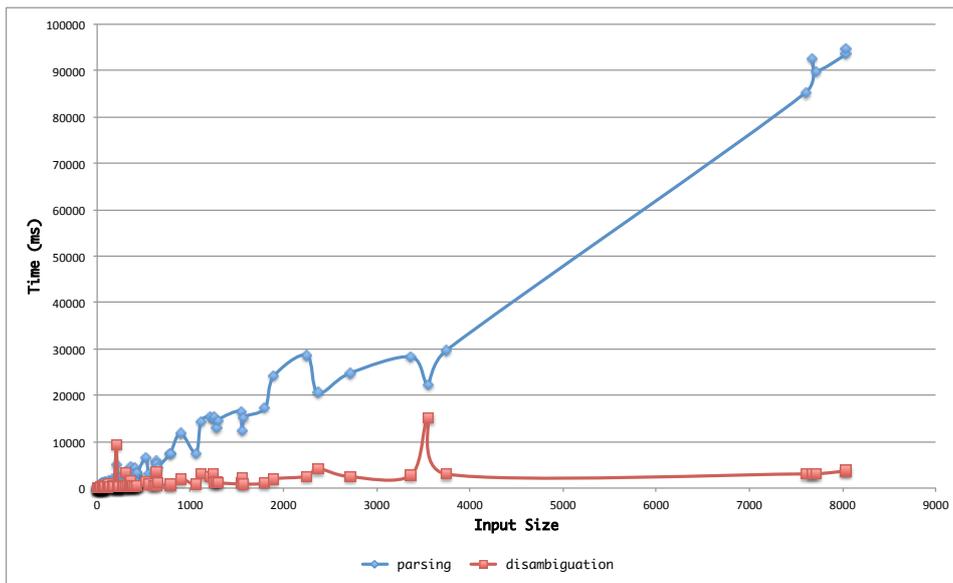
Figure 7.2: The parsing and disambiguation times for mCRL2

We have parsed and disambiguated all mCRL2 files present in the the source release of the software. In total 93 files have been parsed, ranging from 3 to 8037 tokens. Moreover the number of ambiguity nodes in the mCLRL2 example files ranges from zero to 897, with an average of 95 ambiguity nodes. The result of this experiment is plotted in Figure 7.2. As can be seen the disambiguation time for mCRL2 files is constant.

7.3 Tom

The last case study is parsing the island grammar of Tom. The island grammar is presented in Appendix F. Using the island grammar and its disambiguation rules, we are able to parse most of the Tom programs from the tom-examples package. This package, which is shipped with a source distribution of Tom, contains more than 400 examples (for a total of 70,000 lines of code) showing how Tom is used in practice. The size of these examples varies from 24 lines of code (679 characters) to 1,103 lines of code (30,453 characters).

The parsed examples which have a size of about 10,000 characters can be parsed and disambiguated in less than one second, and the disambiguation time of an instance is always lower than its parsing time. Moreover, from what we have observed, the parsing time for Tom instances is linear. For this study, we only focused on the core Tom examples, about 300 files. The other examples were related to XML processing and were not considered. The result of parsing the Tom examples is shown in Figure 7.3.
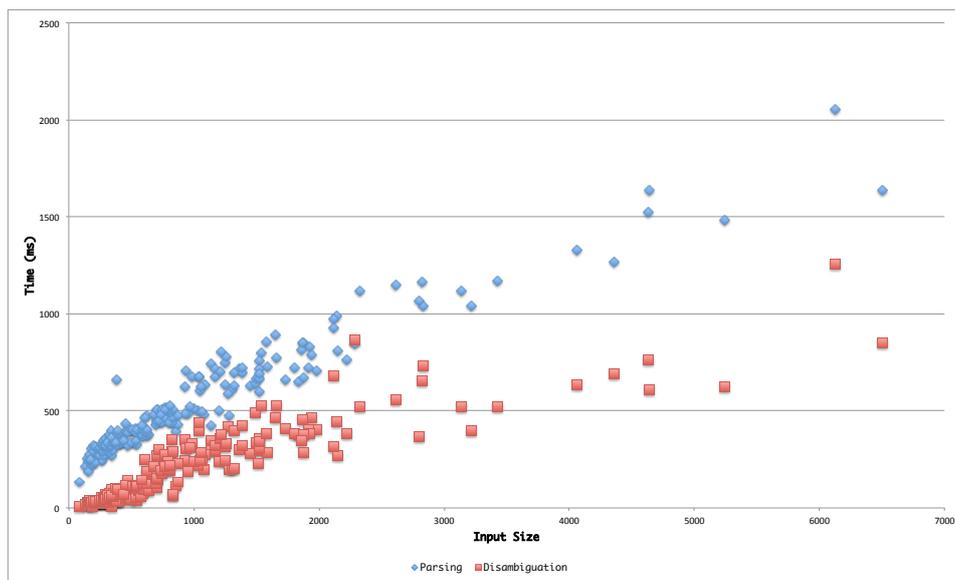
Figure 7.3: The parsing and disambiguation times for Tom

Chapter 8

JGLL Parser Generator

In this chapter we present JGLL, a parser generator for generating GLL-based Java parsers. JGLL, written in Java and Tom, uses Gom-generated data types. This provided us with a declarative way of performing different transformation tasks using Tom. JGLL generates parsers, which are based on an improved version of the GLL parsers presented in [24]. The improvements include optimization in data structures and faster lexing.

The rest of this chapter is organized as follows. Section 8.1 presents the architecture of JGLL. Next, in Section 8.2, we describe the interaction between the parser and lexer in generated parsers. We also demonstrate how generated parsers can deal with the overlapping between tokens and tokens with different types. We conclude this chapter by explaining the improvement in the lexer implementation.

8.1 Architecture

The architecture of JGLL parser generator is shown in Figure 8.1. As can be seen, A GLL parser is generated in six steps. The input to these steps is an MEBNF specification for a language L, and the output is a GLL parser for language L. The function of each step is described as follows:

1. **EBNF Parser** is a GLL parser for parsing MEBNF specifications. It parses the provided syntax of a language and produces an SPPF. If the syntax is not correct, an error message indicating the position where the error happened is shown to the user, and the parser generation process terminates.

2. **SPPF2EBNF** is a Tom specification translating the produced SPPF from the previous step to an EBNF term, as shown in 3.1. The SPPF

corresponding to parsing an MEBNF specification may be ambiguous. The disambiguation is performed in this step. Furthermore, an MEBNF specification has more information than context-free rules: lexical rules are translated into regular expressions. disambiguation rules and priorities into Tom rules, and EMF annotations into Java classes which process annotations. See Chapter 10 for more information on EMF integration.

3. **LoadModules** loads the imported modules from the loaded EBNF term produced in the previous step. If a module is already loaded, it will be reused. Otherwise, the process goes to Step 1, and the module is loaded. When all the modules are loaded, modularity rules are applied, and a EBNF term from all the loaded modules is produced. As explained in Section 3.6, currently, all reachable nonterminals are added to the importing module, without considering module names.

4. **EBNF2BNF** converts an EBNF specification to its equivalent BNF specification. The conversion is performed according to Section 3.2. Note that there is no explicit signature for BNF. An EBNF term is transformed into another EBNF term without specific EBNF constructs. See Section 3.1 for more information.

5. **BNFUtil** extracts first and follow sets, among other required information, from the provided BNF term. These information are fed into parser generator.

6. **Parser Generator** generated a Java parser and related components, such as lexer and disambiguation rules, using the provided information from Step 2 and 5. The generated parsers are in pure Java, and thus there is no dependency to Tom.

8.2 Parsing with a separate Lexer

Our GLL implementation uses a separate lexer, which is very simple and is driven by the parser. The lexer returns all possible token types seen at a particular point of the input. The token types are explicitly requested by the parser, and they may overlap. Being a top-down parser, GLL traverses the rules of a grammar, and at each grammar position, decides whether the tokens received from the lexer are relevant. This check is performed by testing the received tokens against the first and follow sets of alternates. All the relevant tokens at a position are consumed, and irrelevant ones are simply ignored. With this strategy, GLL parsers with a separate lexer are able to detect the overlapping between tokens and tokens with multiple types. These features are essential in parsing embedded languages.
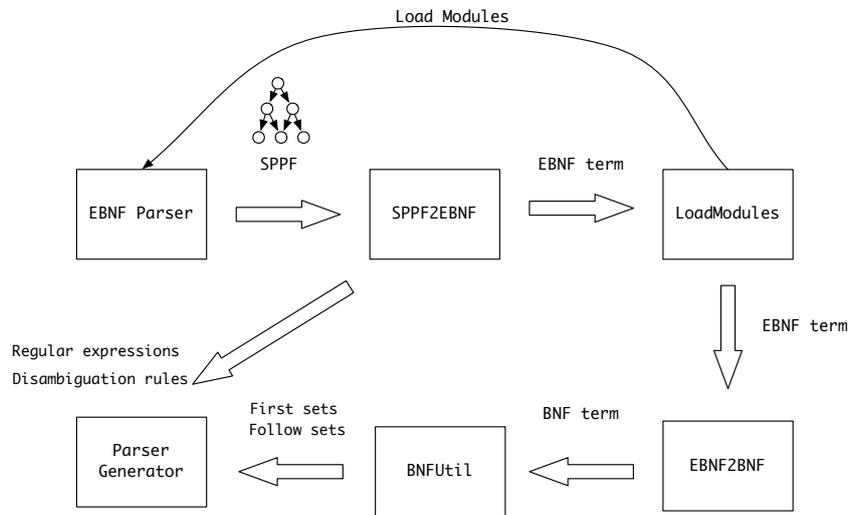
Figure 8.1: The architecture of the JGLL parser generator

As an example, consider the rules from Listing 6.8, which describes the starting rules of an island grammar. The generated parsing code for choosing between an `Island` or `Water` is shown in Listing 8.1.

```java
1  private void parse_Chunk_Label() {
2      if (test(lexer.get(ci, "%match"))) {
3          add(Water_Label, cu, ci, SPPFNode.DUMMY);
4      }
5      if (test(lexer.get(ci, "Water"))) {
6          add(Island_Label, cu, ci, SPPFNode.DUMMY);
7      }
8      label = 0;
9  }
```

Listing 8.1: A Java function associated with

The `test` method in Listing 8.1 checks whether the set of tokens returned by the lexer is not empty. The first parameter of this method, c_i, denotes the current input position, and the second parameter is the token type. The `add` method creates a descriptor for processing a grammar label. In Listing 8.1, if the first `if` at Line 2 evaluates to true, the grammar label corresponding to `Water`, `Water_Label` is added to the set of descriptors. Similarly, at Line 6, the label for processing `Island` is added.

Let us consider the input `%match`. When the parser executes the `parse_Chunk_Label` method, it requests two different token types: the keyword `%match` and `Water`. As Lexer can match both token types, returning `%match` and `%`, respectively, both island and water nonterminals will be processed. This example shows

how overlapping between tokens can be recognized using a separate lexer.

For showing how a GLL parser with a separate lexer treats tokens with different types, we use the grammar in Listing 8.2, which defines an if statement. In this grammar, keywords are not reserved.

---

context-free syntax

```
Program ::= S*
S ::= "if" E "then" S
S ::= E "=" E
```

lexical syntax

```
E ::= [a-z]+
```

Listing 8.2: A grammar for if statements without reserved keywords

The input `"if if then then = if"` can be successfully parsed, without introducing any ambiguity. The parse tree corresponding to this string is shown in Figure 8.2. Other parses, in which the second `"if"` has been interpreted as the beginning of an if statement, were unsuccessful.
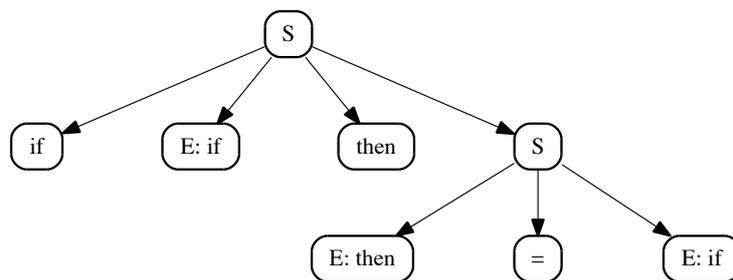
Figure 8.2: Parse tree for `"if if then then = if"`

GLR parsers cannot deal with such cases without additional help. For example, in the Schrödinger's Token method [15], it is shown how to modify the lexer to report the type of a token with multiple types as the Schrödinger type, which is in fact a meta type representing all the actual types. When the parser receives a Schrödinger's token, for each of its actual types, parsing will be continued in parallel.

Scannerless GLR parsers [38] can successfully deal with this example. In scannerless parsing, there is no explicit lexer present, so the parser directly works on the character level. This has the advantage of giving the parser the opportunity to decide about the type of a token based on the context in which the token's characters appear. Scannerless parsing is a powerful paradigm for dealing with tokens with multiple types and overlapping, but

there are some disadvantages to this approach. First, because the parser works on the character level, the size of the parse forest is bigger. Second, in recognizing sequence of characters as a token, ambiguities may occur. For example, the string `"if"` can be interpreted as two tokens, `"i"` and `"f"`, or a single token `"if"`. These ambiguities should be explicitly addressed. Using GLL with a separate lexer, one does not require any specific treatment for tokens which overlap or have different types. Our parser implementation is as powerful as a scannerless parser, and at the same time, has the benefits of a separate lexer.

8.3   Lexer Implementation

The original implementation of [24] used Java's regular expression library as the underlying implementation of the lexer. Our early investigations showed that the lexer is a performance bottleneck. The reason is that Java's regular expressions is an NFA-based library, and as NFA-based regular expressions use backtracking for matching a pattern, the matching may become very slow. We replaced Java's regular expression with `dk.brics.automaton` [25] regular expression library, which is a DFA-based. DFA-based regular expressions do not provide features such as capturing groups, which are also not needed for a lexer, but they can match a pattern linear in the length of a string, independently of the complexity of the pattern. This led to a considerable performance gain in the whole parsing process.

8.4   Attaching Custom Disambiguation Rules

In Section 6.2, we mentioned that one can resolve the island-island ambiguity by using a custom Tom program. In this section, we show how to do this. Listing 8.3 shows the Java code for parsing and disambiguating an input. The `parser` in line 1 refers to an instance of a GLL parser. If parsing the provided input was successful, *i.e.*, the returned SPPF from the `parse` method was not null in line 3, the disambiguation phase starts. Disambiguation rules can be added to a `BottomUpDisambigutor` instance, as shown in Line 10 and 11. In this example, two disambiguation rules are added. The `disambiguate` method, in line 14, tries to disambiguate the SPPF using the provided rules. If the rules could not successfully disambiguate an SPPF, an exception is thrown. Otherwise, if the disambiguation was successful, the SPPF will only contain symbol nodes. A non-ambiguous SPPF can be processed by any Java or Tom programs.

```
1  SymbolNode sppf = parser.parse(input);
2
3  if (sppf == null) {
4      ParseError parseError = parser.getParseError();
5      System.out.println(parseError);
6      System.exit(1);
7  }
8
9  BottomUpDisambiguator disambiguator = new BottomUpDisambiguator();
10 disambiguator.addDisambiguationRule(new DisambiguationRule1());
11 disambiguator.addDisambiguationRule(new DisambiguationRule2());
12
13 try {
14     disambiguator.disambiguate(sppf);
15 } catch (DisambiguationFailure e) {
16     e.printStackTrace();
17 }
```

Listing 8.3: Needed steps for parsing and disambiguation

A custom disambiguator can be added to the `disambiguator` instance. To do this, the user should implement the `DisambiguationRule` interface, shown in Figure 8.4. The interface provides a single method, `apply` which takes an SPPF node and disambiguates the subtree under it. This interface can be implemented using pure Java or a combination of Java and Tom.

```
1  public interface DisambiguationRule {
2      public void apply(SPPFNode node);
3  }
```

Listing 8.4: The `DisambiguationRule` interface

Chapter 9

The Eclipse Plugin

Modern IDEs such as IntelliJ IDEA [9] and Eclipse [3] provide sophisticated environments for editing the source code of Java, and other general-purpose programming languages. These environments provide editors capable of syntax highlighting, code completion, and live error reporting. Most domain-specific languages, however, do not enjoy such tooling, as creating similar environments for domain-specific languages is very expensive. It is, therefore, desirable to generate such editing environment with a set of standard features, such as syntax highlighting, from a DSL's specification. The generated editor can be used as the basis for a full-fledged IDS for the DSL.

In this chapter we show how to create an Eclipse-based editor from a generated MEBNF grammar. The editor is based on the IDE Meta-Tooling Platform (IMP) [18, 8]. IMP provides an automated, yet customizable platform for creating Eclipse-based IDEs. In the rest of this chapter we first introduce IMP in Section 9.1 and how it facilitates IDE development. Then, in Section 9.2, the integration of GLL and IMP is explained. We conclude this chapter by demonstrating some screenshots from an IDE for MEBNF.

9.1   IMP

IMP is an Eclipse project aiming at making the creation of IDEs for custom languages easier. The ultimate goal of IMP is to provide a similar environment to Eclipse Java Development Tools (JDT) [4] for DSLs. IMP produces an Eclipse plugin project with a set of IDE services, each providing a facility such as syntax highlighting, code folding, and many other features found in popular IDEs. IDE services rely on the Abstract Syntax Tree (AST) of a language. IMP makes few assumptions about the type of AST nodes and treat them as instances of Java `java.lang.Object` classes. This provides great

flexibility in attaching custom parsers and AST node types.

The core part of an IMP plugin is a `ParseController`. A parser controller implements a `parse` method which receives an string as input and returns the root of the produced AST. The `parse` method is called by IMP at each update of the editor, and the text in current active editor is provided as input. Furthermore, a `ParserController` implements a `getTokenIterator` method providing an `iterator` for the leaves of a parse tree. The produced AST or `iterator` is then passed to different IDE services for further processing. After the processing, IMP updates the UI components to reflect the changes.

From the set of IMP IDE services, we implemented the following IDE services. The actual implementation of these services using a GLL parser is presented in Section 9.2.

- **Token Colorer** or syntax highlighter provides the editor with information to color a piece of text based on its token type. This service uses the token iterator from the `ParserController`.

- **Tree-Model Builder and Label Provider** creates a labeled tree from the AST. The labeled tree can, for example, be used in the outline view, giving an overview of the structure of a program.

- **Text Folding** provides the editor with information to collapse a block of code. An example is to fold Java methods in JDT.

9.2 Integrating GLL and IMP

In this section we discuss the technical details of integrating GLL parsers with IMP. The examples in this section are from an editor for the MEBNF syntax, *i.e.*, an editor for editing MEBNF specifications.

9.2.1 Implementation of Parser Controller

Listing 9.1 shows a parse controller implementation for MEBNF. The class extends `ParseControllerBase` which is provided by IMP as a base class for parse controllers. The `parse` method, line 6, receives the input as an argument, parses the input, and then disambiguates the resulting SPPF. If a parse error occurs, *i.e.*, `sppf == null`, the `handleSimpleMessage` from the provided `handler` object is called. This results in showing the error message in the editor window. As a produced SPPF may be ambiguous, it is disambiguated before returning the root of the SPPF. If disambiguation fails, *e.g.*, the corresponding disambiguation rule is not present, an error is shown to the user, see line 20.

```
 1  public class MEBNFParseController extends ParseControllerBase {
 2
 3      private MEBNFParser parser;
 4      private BottomUpDisambiguator disambiguator;
 5
 6      public Object parse(String input, IProgressMonitor monitor) {
 7
 8          SymbolNode  sppf = parser.parse(input);
 9
10          if(sppf == null) {
11              ParseError error = parser.getParseError();
12              handler.handleSimpleMessage(error.toString(), error.getInputIndex(),
13                                          error.getInputIndex(), 0, 0, 0, 0);
14              return null;
15          }
16
17          try {
18              disambiguator.disambiguate(sppf);
19          } catch(DisambiguationFailure e) {
20              handler.handleSimpleMessage(e.toString(), e.getInputIndex(),
21                                          e.getInputIndex(), 0, 0, 0, 0);
22          }
23          fCurrentAst = sppf;
24          return sppf;
25      }
26
27      public Iterator<SymbolNode> getTokenIterator(IRegion region) {
28          if(fCurrentAst == null) {
29              return null;
30          }
31          return SPPFUtil.getTerminalIterator((SPPFNode) fCurrentAst);
32      }
```

Listing 9.1: A parser controller implementation for MEBNF

The getTokenIterator method returns an java.util.Iterator instance from the
set of tokens. The set of tokens is calculated by SPPFUtil which is a Tom
program. Listing 9.2 shows how the getTokenIterator method is implemented.
As can be seen, a TopDown strategy is employed to gather all SPPF nodes
having no children, Line 13.

```
public static Iterator<SymbolNode> getTerminalIterator(SPPFNode node) {
    List<SymbolNode> terminals = new ArrayList<SymbolNode>();
    try {
        `TopDown(GetTerminals(terminals)).visitLight(node, new SPPFIntrospector());
    } catch (VisitFailure e) {
        e.printStackTrace();
    }
    return terminals.iterator();
}


%strategy GetTerminals(list:List) extends Identity() {
    visit SPPFNode {
        s@SymbolNode(name, concNode()) -> { list.add(`s); }
    }
}
```

Listing 9.2: Getting the token iterator using Tom

9.2.2 Token Coloring

Token coloring is done through the ITokenColorer interface, provided by IMP.
A language-specific class should implement the getColoring method of the
ITokenColorer interface in order to provide coloring rules for the tokens of the
language.

```
1  TextAttribute keywordAttribute = new TextAttribute(
2                      display.getSystemColor(SWT.COLOR_DARK_MAGENTA),
3                      null, SWT.BOLD);
4
5  TextAttribute commentAttribute = new TextAttribute(
6                      display.getSystemColor(SWT.COLOR_DARK_GREEN),
7                      null, SWT.NORMAL);
8
9  public TextAttribute getColoring(IParseController controller, Object o) {
10     if (o == null) { return null; }
11
12     SymbolNode token = (SymbolNode) o;
13
14     if(keywords.contains(token.getSymbolName())) {
15         return keywordAttribute;
16     }
17     if("Comment".equals(token.getSymbolName())) {
18         return commentAttribute;
19     }
20     return super.getColoring(controller, token);
21 }
```

Listing 9.3: Token coloring for MEBNF's keywords

Listing 9.3 shows how this method is implemented for MEBNF's keywords. Line 1 and 5 declare text attributes specifying font properties such as color and type. The implementation of the `getColoring` method begins at Line 9. The provided `Object` parameter to this method is the token to be colored. The tokens are obtained through the MEBNFParseController in Listing 9.1. Line 14 and 17 check the token type, by calling the method `getSymbolName`, and select a text attribute accordingly. If no match is found, the default text attribute is returned from Line 20.

9.2.3 Tree-Model Builder and Label Provider

IMP's tree-model builder provides means for creating a tree model from the AST. This tree, augmented by a label provider assigning a label to each node of the tree-model, can be used to create an outline view for the source code. An outline view provides a tree user interface for navigating the structure of code. For example, in JDT, the outline view gives and overview of the methods, constructors, and fields defined in a class.

A tree model in IMP is specified by instances of `ModelTreeNode`. A `ModelTreeNode` is a simple Java object defining a tree node holding an AST node and keeping references to its parent and children. To implement a tree model, the AST should be traversed and the desired parts be added to a tree model. For example, the Tom program in Listing 9.4 creates a tree model from MEBNF's lexical and context-free sections.

```
ModelTreeNode root = new ModelTreeNode(node);

%match(node) {

SymbolNode("Section",
           concNode(_*, section@SymbolNode[name="ContextFreeSection"], _*)) -> {
    ModelTreeNode m = new ModelTreeNode(`section);
    root.addChild(m);

}

SymbolNode("Section",
           concNode(_*, section@SymbolNode[name="LexicalSection"], _*)) -> {
    ModelTreeNode m = new ModelTreeNode(`section);
    root.addChild(m);
}
```

Listing 9.4: Creating tree models from MEBNF's AST

In Listing 9.4, `node` refers to a SPPF node having the label `Section`, corresponding to the `Section` nonterminal in Listing 3.2. A `ModelTreeNode` named root holding the root node of AST. The **match** construct, matches the SPPF

nodes, and if a node labeled `ContextFreeSection` or `LexicalSection` is found, a new `ModelTreeNode` holding the matched section is created. The newly created child tree model object is added to the root node as a child.

9.2.4 Text Folding

The text folding service is provided by IMP's `FolderBase` class. To provide language-specific text folding, the `sendVisitorToAST` method of the `Folder` class should be implemented. In the implemented method, AST nodes associated with foldable blocks should be passed to `makeAnnotation()`. Listing 9.5 shows how to add make MEBNF's sections foldable. The provided `ast` is the root of AST. In this example, we only need to examine direct children of the root node. Therefore, a simple for loop suffices. For complex, nested text folding, one should examine deeper nodes in the AST.

```
 1  public class MEBNFFoldingUpdater extends FolderBase {
 2
 3      protected void sendVisitorToAST(HashMap<Annotation, Position> newAnnotations,
 4                                      List<Annotation> annotations, Object ast) {
 5
 6          for (SPPFNode child : ((SPPFNode) ast).getChildren()) {
 7
 8              SymbolNode node = (SymbolNode) child;
 9              if(node.getSymbolName().equals("Section")) {
10                  makeAnnotation(node);
11              }
12          }
13      }
14  }
```

Listing 9.5: Making MEBNF's sections foldable

9.3 An IDE for MEBNF

In this section we show a number of screenshots from an IMP-based Eclipse plugin for MEBNF. Figure 9.1 shows an overview of the plugin. As can be seen, the plugin provides syntax highlighting, an outline, and text folding. In the figure, the `Declaration` node in the outline view is selected, and as a result, the `Declaration` nonterminal in the editor is highlighted. In addition, the annotations associated with the alternates of `Statement` are folded.

Error messages appear in the editor as a parse error occurs. For example, Figure 9.2 shows a parse error caused by inserting a character `t` after the definition of the `Program` nonterminal.
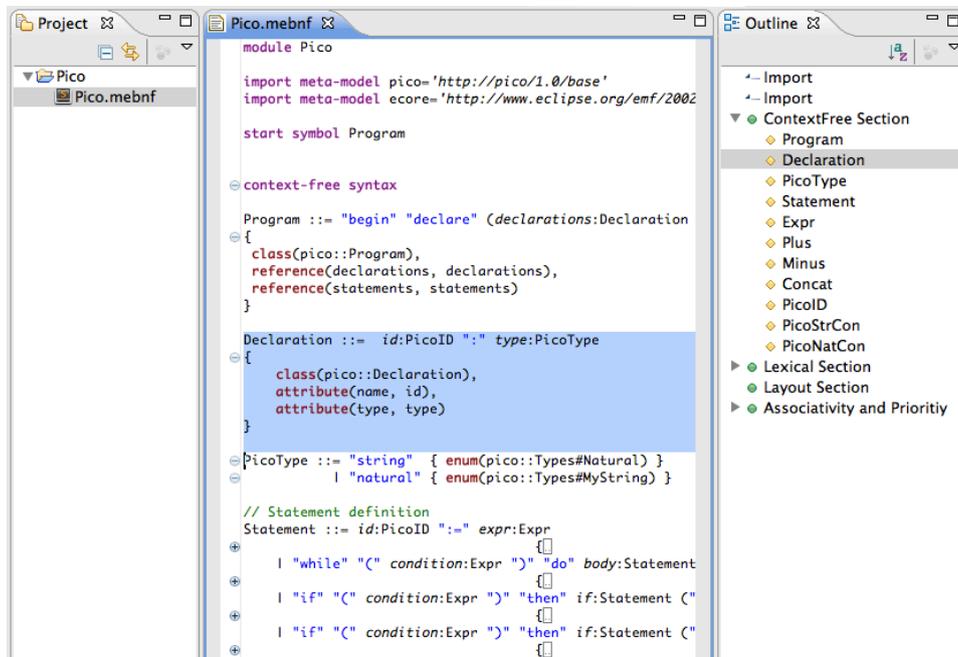
Figure 9.1: An overview of an IMP-based Eclipse plugin for MEBNF.
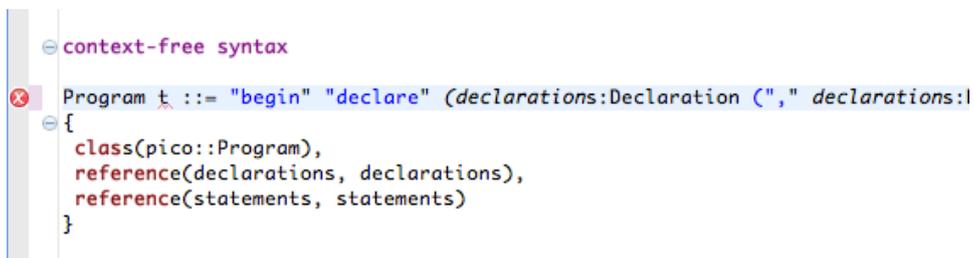
Figure 9.2: Error messages appear in the editor as soon as a parse error occurs

Chapter 10

EMF Model Generation

One of the initial goals of this project was to provide a mapping from parse trees to EMF models. The first effort lead to the thesis "mlBNF: A syntax formalism for domain-specific languages" [24]. The thesis describes a syntax formalism for defining domain-specific languages, in BNF, and provides a mapping to EMF models through annotations. Two shortcomings of mlBNF, no EBNF support and no disambiguation mechanism, have been addressed in this work. Together with the Eclipse plugin introduced in Chapter 9, we provide a viable alternative to the state of the art tools in the area of metamodeling and concrete syntax, such as XText [13] and EMFText [6].

In this chapter, we introduce a revised version of the annotations in [24] for defining the mappings to EMF. The annotations, resembling attribute grammars, define how an SPPF should be evaluated to create corresponding EMF models. The revised annotations and an example of how they can be used is given in Section 10.1. The EBNF support and disambiguation mechanism impose difficulties in the evaluation of annotations. We conclude this chapter by discussing these implementation challenges in Section 10.2.

10.1 Concrete Syntax for Annotations

For defining a mapping from an SPPF to EMF models, we provide the concrete syntax of MEBNF, shown in Listing 3.3, with an annotation facility. For this, an `Annotations` nonterminal is added to each context-free rule. In addition, a separate import type, for importing meta-models, and a label for symbols of the grammar is provided. The labels are used during the evaluation of annotations to access a specific part of the parse tree. The resulting grammar from these changes is shown in Listing 10.1.

```
Import ::= "import" ModuleName Retract*
         | "import" "meta-model" NameSpace "=" URI

ContextFreeRule ::= Nonterminal "::=" Symbol* Annotations?
                      ("|" Symbol* Annotations?)*

Annotations ::= "{" (Annotation ("," Annotation)*)? "}"

Symbol ::= (Label ":")? Nonterminal
```

Listing 10.1: Augmenting the MEBNF concrete syntax with annotation support

The rules defining an `Annotation` itself are given in Listing 10.2. As can be seen, each alternate of a nonterminal can be annotated with one of the six provided annotation types. An annotation starts with its name, followed by a set of parameters, denoted by `Name`.

context-free syntax

```
Annotation ::= Class | Attribute | Reference | DataType | Enum | Propagate

Class ::= "class" "(" Name ( "::" Name )+ ")"
Attribute ::= "attribute"
"(" (LiteralAttribute | ReferenceAttribute | EnumAttribute) ")"
LiteralAttribute ::= Name "," String
ReferenceAttribute ::= Name "," Name
EnumAttribute ::= Name "," Name "::" Name ( "::" Name )* "#" Name
Reference ::= "reference" "(" Name "," Name ")"
DataType ::= "type" "(" Name ("::" Name)+ ")"
Enum ::= "enum" "(" Name "::" Name ( "::" Name )* "#" Name ")"
Propagate ::= "propagate" "(" Name ")"
```

lexical syntax

```
Name ::= Id
```

Listing 10.2: Concrete syntax for defining annotations

```
1   import meta-model expr  = 'http://arithmetics/1.0/'
2   import meta-model ecore = 'http://www.eclipse.org/emf/2002/Ecore'
3
4   E ::= lhs:E "+" rhs:E
5          {
6                class(expr::BinaryExpression),
7                reference(leftHandSide, lhs),
8                reference(rightHandSide, rhs),
9                attribute(operator, expr::BinaryOperators::ADDITION)
10         }
11     | lhs:E "-" rhs:E
12         {
13                class(expr::BinaryExpression),
14                reference(leftHandSide, lhs),
15                reference(rightHandSide, rhs),
16                attribute(operator, expr::BinaryOperators::SUBTRACTION)
17         }
18     | lhs:E "*" rhs:E
19         {
20                class(expr::BinaryExpression),
21                reference(leftHandSide, lhs),
22                reference(rightHandSide, rhs),
23                attribute(operator, expr::BinaryOperators::MULTIPLICATION)
24         }
25     | lhs:E "/" rhs:E
26         {
27                class(expr::BinaryExpression),
28                reference(leftHandSide, lhs),
29                reference(rightHandSide, rhs),
30                attribute(operator, expr::BinaryOperators::DIVISION)
31         }
32     | "-" expr:E
33         {
34                class(expr::UnaryExpression),
35                reference(expression, expr),
36                attribute(operator, expr::UnaryOperators, MINUS)
37         }
38     | lhs:E "^" rhs:E
39         {
40                class(expr::BinaryExpression),
41                reference(leftHandSide, lhs),
42                reference(rightHandSide, rhs),
43                attribute(operator, expr::BinaryOperators::EXPONENTIATION)
44         }
45     | number:Number
46         {
47                class(expr::IntegerExpression),
48                attribute(value, number)
49         }
50
51  Number ::= Digit { type(ecore::EBigInteger) }
```

Listing 10.3: EMF mapping annotations for arithmetics expressions

Listing 10.3 shows the EMF mapping for the grammar of arithmetic expressions, shown in Listing 5.8. In lines 3 and 4, two meta-models are imported. Imported meta-models have a namespace, which can be used to refer to them in annotations. The imported meta-models, the Arithmetics meta-model, shown in Figure 10.1, and the standard Ecore EMF meta-model, is used to instantiate EMF models.
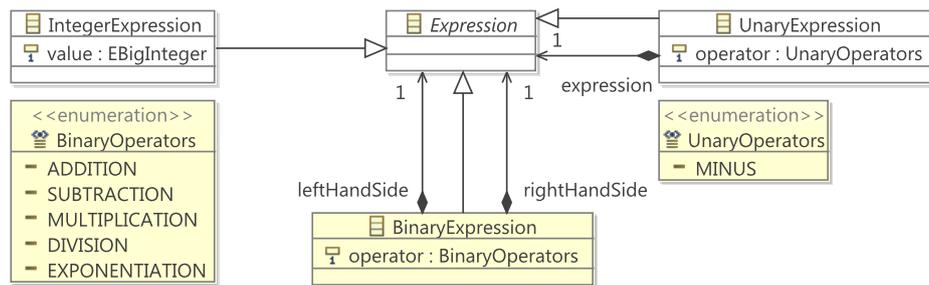
Figure 10.1: The Arithmetics meta-model

In lines 4-50, the alternates of `E` are defined. The first alternate contains three annotations. The *class* annotation in line 6 is used to instantiate the class `BinaryExpression` from the Arithmetics meta-model, indicated by the meta-model's namespace, `expr` before ::. In lines 8 and 9, *reference* annotations are used to instantiate references of an instance of the `BinaryExpression` class. Reference annotations take two parameters. The first parameter is a meta-model reference identifier, and the second one is a symbol label. The meta-model reference identifier is used to refer to a reference of a meta-class. The symbol label is used to indicate the part of the parse tree corresponding to the alternate of a context-free rule. This part of the tree should be traversed and evaluated. The meta-model reference will then be instantiated with the result of this traversal. For the reference annotation in line 7, the `leftHandSide` reference from the `BinaryExpression` class is instantiated with the value of the traversal of the tree corresponding to `lhs`.

The *attribute* annotation in line 10 is used to instantiate the `operator` attribute of an instance of the `BinaryExpression` class. There are three kinds of attribute annotations. 1) *Enumeration attribute* annotations, instantiating an attribute of a class instance with an enumeration value, *literal attribute* annotations, instantiating an attribute with a literal value, and 3) *reference attribute* annotations, instantiating an attribute with the result of a traversal. The attribute annotation in line 10 is an example of an enumeration attribute annotation. The literal `Addition` of the `BinaryOperators` enumeration is assigned to the `operator` attribute of the `BinaryExpression` class. An example of a reference attribute annotation is given in line 48.

Line 51 shows the usage of a *datatype* annotation, denoted by `type`. Datatype annotations indicate that the result of evaluation of a tree is casted to the indicated data type. In this case, the literal value of a `Number` is casted to a `EBigInteger`, which is a standard Ecore datatype. As a result, the `value` attribute of an instance of an `IntegerExpression`, as defined in lines 45-59, is instantiated with an instance of EBigInteger. The *enum* and *propagate* annotations are not discussed here. For more information on how to use these annotations, see [24]. Moreover, as another example, an ambiguous version of the Pico language grammar, written in EBNF, and its mapping to EMF is provided in Appendix D. The pico example has also been used in [24]. Therefore it can be used for comparing the new and old versions of mappings to EMF.

10.2   Implementation

As explained in Section 2.2, the SPPF keeps the information related to grammar positions in packed nodes. These information are essential in retrieving the labels and annotations of nodes for the conversion of an SPPF to an EMF model. In our disambiguation mechanism, we remove the packed nodes and intermediate nodes which are not part of an ambiguity node. This removal leads to the loss of position information in an SPPF. To overcome this, we move the grammar position information from the packed nodes, which should be removed, to appropriate symbol nodes.

A packed node can be removed only if its the only child. If a packed node has packed node siblings, it is part of an ambiguity node, and will be removed after disambiguation. When a packed node is removed, its grammar position information is added to its parent symbol node. This information is used later to add retrieve the labels and annotations for the symbol nodes.

In principle, the label of a symbol node is retrieved using the position information of its parent, while the annotation information come from its child. We, however, move the position information of a packed node to its parent, and not its child. The reason is that symbol nodes may be shared and we cannot simply push down the position information to all the children. This leads a symbol node having multiple positions. After ambiguities are resolved, and a correct derivation has been chosen, we can retrieve the labels of a symbol node using the information from its parent. In an unambiguous SPPF, only some ϵ nodes can be shared. Moreover, we remove the ϵ nodes during while removing intermediary EBNF nodes, see Section 3.2. As a result, in an unambiguous SPPF, each symbol node has only one parent. Furthermore, the annotations are retrieved from the position information of a symbol node itself.

Another important implementation note is how to treat intermediary EBNF nodes. These nodes, similar to any other symbol node, contain the position information after the removal of unnecessary packed and intermediate nodes. Therefore, while removing the EBNF intermediary nodes, in Section 3.2, the position information should be pushed down to their children, so that their corresponding labels and annotations can be retrieved.

After the label and annotation information are added to symbol nodes of an unambiguous SPPF, one can traverse the SPPF, using Java or Tom, and generate models using these information. The generation of models from an annotated SPPF is basically the same as in [24].

Chapter 11

Future Work

In this chapter we discuss how the work presented in this thesis can be extended in the future. Each section discusses a possibility of improving this work, or introduces a new idea for future research, based on what have already been done.

11.1 Extending the syntax of disambiguation rules

The presented syntax for disambiguation rules in Section 5.2 can be extended to support more complicated patterns. The following extensions to this syntax is desirable.

- EBNF style modifiers, such as ?, ∗, + for matching specific numbers of terminals and nonterminals.

- A depth search operator for matching pattens in arbitrary depth of an SPPF.

- A not follow operator, disallowing a specific sequence of terminals and nonterminals in an SPPF.

11.2 A Framework for Parsing Language Embeddings

Parsing language embeddings and extensions poses a considerable challenge from the parsing point of view. One of our goals in developing GText, and its disambiguation mechanism, was to provide an environment to facilitate

experimenting with hard, ambiguous grammars, such as grammars of language embeddings. In this thesis we showed how to parse Tom, using island grammars, and disambiguate it, using term rewriting within the SPPF. In the future, this approach should be tested on other language combinations, such as Java+SQL and Java+XML. The ultimate goal is to provide a generic framework for language extensions. Using GLL to parse layout-sensitive languages, which is another difficult parsing topic, is another possibility for future work.

11.3 Embedding Precedence and Associativity Rules Inside the Parser

Grammars with recursive binary and unary operators, such as the grammar of mCRL2, considerably slow down the performance of parsing. For these specific ambiguities, which often occur in practice, we should try to perform the disambiguation while parsing, and disallow the accumulation of ambiguity nodes in the first place. One possible approach for disambiguating binary and unary operators while parsing is to use more look ahead symbols, *i.e.*, implementing GLL(k) parsers. It is, however, not clear how more look ahead symbols will affect the complexity of GLL. Another possible approach is to internally transform the grammar of binary operators to an equivalent unambiguous grammar. This approach will produce different parse trees than expected by the user. Therefore, one should provide a mechanism to transform parse trees corresponding to unambiguous grammars into the parse trees of the original ambiguous grammar.

11.4 Improving the Modularity in MEBNF

As discussed in Section 3.6, both modularity paradigms, importing with and without considering the namespace, have their own shortcomings. While importing by considering module names allows two nonterminals with the same name in two modules be totally separated, it does not allow grammar extension. On the other hand, importing without preserving the namespace, may lead to unexpected results, as the production rules of all nonterminals with the same name will be merged. A better approach may be the combination of these two import mechanisms, using separate import commands. Improving the modularity mechanism is of high priority in future work.

11.5   Performance Improvements

Our GLL implementation is not thoroughly optimized. Specifically, the SPPF construction phase has shown performance problems while constructing parse forests for fairly ambiguous grammars, *e.g.*, for the grammar of mCRL2 which uses a considerable number of binary and unary operators. Moreover, we have not yet tested our GLL implementation with considerably large grammars, *e.g.*, the grammar of Java. Such large grammars will likely cause performance problems, as their SPPF will grow very large. An important part of future work will be the optimization of our GLL implementation.

11.6   Error Recovery for GLL Parsers

Chapter 4 introduces a simple error reporting mechanism, which works fairly well. The current mechanism can report the location of a parse error and the grammar pointer where the error occurred. The mechanism, however, provides no help in recovering from the error. A possible research path for the future can be the development of error recovery mechanisms for GLL parsers. In addition, a study should be conducted on the use of complex heuristics in detecting the location of parse errors.

11.7   Improving the Eclipse Plugin for MEBNF

The Eclipse plugin presented in Chapter 9 is just a proof of concept, and not much effort has put into making it usable for production use. Moreover, some essential features such as hyperlinking and building is missing. With hyperlinking, the user can, for example, jump to a nonterminal definition by clicking on the nonterminal name in the body of a context-free rule. In addition, hyperlinking should support modularity, *e.g.*, clicking on a nonterminal, defined in another module, should open the module and show the nonterminal definition. A building service for MEBNF can compile the parsed MEBNF files and provide semantic errors, *e.g.*, reporting nonterminals which do not have a definition.

A next step after completing the Eclipse plugin for EMBNF would be the automatic generation of Eclipse plugins for MEBNF grammars. A default set of features such as syntax highlighting and error reporting should be provided. In addition, the generated plugin should be easily customizable, so that the user can augment the generated plugin with language-specific features.

11.8   Parse Tree Visualization in Eclipse

In order to make the process of developing languages easier, a parse tree visualization should be integrated into the Eclipse plugin for MEBNF. Currently, we generate a GraphViz dot [7] file from an SPPF and then manually generate a graph from the dot file. This process is slow, and more importantly, the produced graph is a large image, thus cannot be rearranged or filtered. It will be a very useful feature to provide an Eclipse-based SPPF visualization tool, capable or filtering nodes, *e.g.*, removing layout nodes, showing ambiguity nodes, and rearranging SPPF nodes.

11.9   Scala Support

As shown in Section 8.4, the SPPF mapping in Appendix B can be used to write arbitrary disambiguation or SPPF processing code in Tom. Tom is a very powerful term rewriting and pattern matching language. However, it is not a major language and not many developers know Tom. Moreover, the tooling for Tom is far from perfect. Scala, a Java Virtual Machine language, provides identical term rewriting features to Tom, and is becoming a mainstream language. It will be essential for the widespread usage of our workbench to provide Scala support, next to Tom and Java support, for SPPF processing.

Chapter 12

Conclusions

In this thesis we presented $\mathcal{G}$text, a language workbench based on GLL and term rewriting. $\mathcal{G}$text is composed of three main components. The first component is the JGLL parser generator, introduced in Chapter 8, being used to generate Java-based GLL parsers. JGLL is based on the presented context-free syntax formalism, MEBNF, introduced in Chapter 3. The second component is the Eclipse plugin, presented in Chapter 9, providing an environment for editing MEBNF grammars. In the future, we plan to generate Eclipse plugins with a basic set of features, such as syntax highlighting, from an MEBNF specification. The third component of our language workbench is the mapping of concrete syntax to EMF models, through the annotations introduced in Chapter 10.

The main research question in this thesis was how to disambiguate an SPFF, produced by a GLL parser. In Chapter 5 we introduced our disambiguation mechanism which is based on rewriting within the parse forest, without modifications in the parser. We provided real-world hard cases which can be disambiguated with our technique in Chapter 6. To validate our disambiguation mechanism, we used three case studies in Chapter 7. The results of these case studies suggest that our disambiguation mechanism is suitable for 'real' languages with a reasonable number of ambiguities. With further optimizations in the parser, and embedding the priority and associativity rules within the parser, we expect to obtain performance improvements in the future.

$\mathcal{G}$text, as presented in this thesis, is a prototype of a language workbench, and there is still a long way to prepare it for production use. Nevertheless, this thesis provided the basis for developing a language workbench based on GLL. With work that will be done in the future, we hope that one day we introduce $\mathcal{G}$text as a mainstream language workbench.

Bibliography

[1] ANTLR. http://www.antlr.org/, last visited: Feb 2012.

[2] DParser. http://dparser.sourceforge.net/, last visited: June 2012.

[3] Eclipse. http://www.eclipse.org/, last visited: August 2012.

[4] Eclipse Java Development Tools (JDT). http://www.eclipse.org/jdt/, last visited: August 2012.

[5] EMF. http://www.eclipse.org/modeling/emf/, last visited: May 2012.

[6] EMFText. http://www.emftext.org/, last visited: August 2012.

[7] Graphviz. http://www.graphviz.org/, last visited: August 2012.

[8] IMP: The IDE Meta-Tooling Platform. http://www.eclipse.org/imp/, last visited: August 2012.

[9] IntelliJ IDEA. http://www.jetbrains.com/idea/, last visited: August 2012.

[10] mCRL2. http://www.mcrl2.org/, last visited: August 2012.

[11] Rascal-meta programming language. http://www.rascal-mpl.org/, last visited: August 2012.

[12] Spoofax language workbench. http://strategoxt.org/Spoofax, last visited: August 2012.

[13] Xtext. http://www.eclipse.org/Xtext/, last visited: August 2012.

[14] A.V. Aho, M.S. Lam, R. Sethi, and J.D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison Wesley, August 2006.

[15] J. Aycock and R.N. Horspool. Practical Earley Parsing. *Comput. J.*, 45(6):620–630, 2002.

[16] E. Balland, P. Brauner, R. Kopetz, P.E. Moreau, and A. Reilles. Tom: piggybacking rewriting on java. In *Proc. 18th international conference on Term rewriting and applications*, RTA'07, pages 36–47, Berlin, Heidelberg, 2007. Springer-Verlag.

[17] E. Balland, C. Kirchner, and P.E. Moreau. Formal Islands. In Michael Johnson and Varmo Vene, editors, *11th International Conference on Algebraic Methodology and Software Technology*, volume 4019 of *LNCS*, pages 51–65, Kuressaare, Estonia, July 2006. Springer-Verlag.

[18] P. Charles, R.M. Fuhrer, and S.M. Sutton Jr. IMP: a meta-tooling platform for creating language-specific ides in eclipse. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, ASE '07, pages 485–488, New York, NY, USA, 2007. ACM.

[19] N. Dershowitz. Term rewriting systems by "Terese" (M. Bezem, J.W. Klop, and R. de Vrijer, eds.), Cambridge University Press, Cambridge Tracts in Theoretical Computer Science 55, 2003, hard cover: Isbn 0-521-39115-6, xxii+884 pages. *Theory Pract. Log. Program.*, 5(3):395–399, may 2005.

[20] M. Fowler. *Domain-Specific Languages.* Addison-Wesley Professional, 2010.

[21] J. Heering, P. Hendriks, P. Klint, and J. Rekers. The syntax definition formalism SDF - reference manual. *SIGPLAN Notices*, 24(11):43–75, 1989.

[22] J.E. Hopcroft, R. Motwani, and J.D. Ullman. *Introduction to Automata Theory, Languages, and Computation (3rd Edition).* Addison-Wesley Longman Publishing Co., In. Boston, MA, USA, 2006.

[23] S.C. Johnson. Yacc: Yet Another Compiler-Compiler. http://dinosaur.compilertools.net/yacc/index.html, last visited: July 2012.

[24] M. Manders. mlBNF - a syntax formalism for domain specific languages, April 2011. MSc Thesis, Eindhoven University of Technology. http://alexandria.tue.nl/extra1/afstversl/wsk-i/manders2011.pdf.

[25] A. Møller. dk.brics.automaton – finite-state automata and regular expressions for Java, 2010. http://www.brics.dk/automaton/.

[26] L. Moonen. Generating robust parsers using island grammars. In *Reverse Engineering, 2001. Proceedings. Eighth Working Conference on*, pages 13–22, 2001.

[27] P.E. Moreau, C. Ringeissen, and M. Vittek. A pattern matching compiler for multiple target languages. In *Compiler Construction*, volume 2622 of *Lecture Notes in Computer Science*, pages 61–76. Springer Berlin / Heidelberg, 2003.

[28] E. Post. Island grammars in ASF+SDF, Summer 2007. MSc Thesis, University of Amsterdam. http://homepages.cwi.nl/~paulk/theses/ErikPost.pdf.

[29] J. Rekers. *Parser Generation for Interactive Environments*. PhD thesis, University of Amsterdam, Amsterdam, The Netherlands, January 1992.

[30] E. Scott and A. Johnstone. Modelling GLL parser implementations. In *Proc. of the Third international conference on Software language engineering*, SLE'10, pages 42–61, Berlin, Heidelberg, 2011. Springer-Verlag.

[31] E. Scott and A. Johnstone. GLL parse-tree generation. *Science of Computer Programming*, 2012. To appear.

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

[33] A. v. Deursen and T. Kuipers. Building documentation generators. In *ICSM*, pages 40–49, 1999.

[34] M. v.d. Brand, P. Klint, and J. Vinju. The language specification formalism ASF+SDF, September 2008. `http://www.meta-environment.org/doc/books/extraction-transformation/asfsdf/asfsdf.pdf`.

[35] M. v.d. Brand, J. Scheerder, J.J. Vinju, and E. Visser. Disambiguation Filters for Scannerless Generalized LR Parsers. In *CC*, pages 143–158, 2002.

[36] R. v.d. Leek. Implementation Strategies for Island Grammars, May 2005. MSc Thesis, Delft University of Technology, `http://swerl.tudelft.nl/twiki/pub/Main/RobVanDerLeek/robvanderleek.pdf`.

[37] J. Vinju. SDF Disambiguation Medkit for Programming Languages, 2007. `http://www.meta-environment.org/doc/books/syntax/sdf-disambiguation/sdf-disambiguation.pdf`, last visited: July 2012.

[38] E. Visser. Scannerless Generalized-LR Parsing. Technical Report P9707, Programming Research Group, University of Amsterdam, July 1997.

Appendix A

MEBNF's Concrete Syntax

---

module MEBNF

import Annotations
import Disambiguation
import Basics

start symbol Module

context-free syntax

```
Module ::=  "module" ModuleName Import* StartSymbolSection?  Section*

Import ::= "import" ModuleName Retract* | "import" "meta-model" NameSpace "=" URI

Retract ::= "retract" (ContextFreeRule | LexicalRule)*

Section ::= LexicalSection | ContextFreeSection | DisambiguationSection

StartSymbolSection ::= "start" "symbol" Nonterminal

ContextFreeSection ::= "context-free" "syntax" ContextFreeRule*

ContextFreeRule ::= Nonterminal "::=" Symbol* ("|" Symbol*)*

Symbol ::= Symbol "*"
         | Symbol "+"
         | Symbol "?"
         | Keyword
         | Nonterminal
         | "(" Alt ")"

Alt ::=  Alt "|" Alt
      |  Symbol+

LexicalSection ::= "lexical" "syntax" LexicalRule*

LexicalRule ::= Nonterminal "::=" RegularExpression LexicalExclusion?

LexicalExclusion ::= "-/-" "{" RegularExpression
                          ("," RegularExpression)* "}"
```

```
RegularExpression ::= RegularExpression RegularExpression
                    | RegularExpression "|" RegularExpression
                    | RegularExpression "*"
                    | RegularExpression "+"
                    | RegularExpression "?"
                    | "(" RegularExpression ")"
                    | Nonterminal | Keyword | CharClass | "."

CharClass ::= "[" Not? Character+ "]"

Character ::= Char
            | EscapedChar
            | WordChar "-" WordChar
            | Digit "-" Digit
```

lexical syntax

```
Label ::= Id
ModuleName ::= Id
Name ::= Id
URI ::= ['][^']*[']
Char ::= . -/- {[\\], [\-], [\ ], [\t], [\r], [\n], [\[], [\]], [\^], [\.]}

Nonterminal ::= Id -/- { "lexical", "context-free", "syntax",
                         "module", "import",
                         "start", "retract", "symbol" }
Keyword ::= String
EscapedChar ::= [\\][\\ \- \ t r n \[ \] \^ \.]
WordChar ::= [a-zA-Z]
Not ::= [\^]
```

layout syntax

```
Whitespace ::= [\ \t \n \r]+
Comment ::= [/][/][^\r\n]*
```

Listing A.1: The MEBNF module

---

module Annotations

import Basics

start symbol Annotation

context-free syntax

```
Annotation ::= Class | Attribute | Reference | DataType | Enum | Propagate

Class ::= "class" "(" Name ( "::" Name )+ ")"

Attribute ::= "attribute"
              "(" (LiteralAttribute | ReferenceAttribute | EnumAttribute) ")"

LiteralAttribute ::= Name "," String

ReferenceAttribute ::= Name "," Name

EnumAttribute ::= Name "," Name "::" Name ( "::" Name )* "#" Name

Reference ::= "reference" "(" Name "," Name ")"
```

89

```
DataType ::= "type" "(" Name ("::" Name)+ ")"

Enum ::= "enum" "(" Name "::" Name ( "::" Name )* "#" Name ")"

Propagate ::= "propagate" "(" Name ")"
```

lexical syntax

```
Name ::= Id
```

Listing A.2: The Annotations module

module Disambiguation

import Basics

start symbol DisambiguationSection

context-free syntax

```
DisambiguationSection ::= "disambiguation" "rules" Rules*

Rule ::= "remove" PackedNode
       | "prefer" PackedNode "," PackedNode

PackedNode ::= "[" (SymbolNode | Any) ("," (SymbolNode | Any) )* "]"

SymbolNode ::= SymbolName
             | SymbolName "(" ( SymbolNode | Any) ("," (SymbolNode | Any) )* ")"
```

lexical syntax

```
Any ::= [_][*]?
SymbolName ::= String | Id
```

Listing A.3: The Disambiguation module

module Priorities

import Basics

start symbol PrioritiesSection

context-free syntax

```
PrioritiesSection ::= "associativity" "and" "priority" Group*

Group ::= "{" (AssociativityGroup (">" AssociativityGroup)*)? "}"

AssociativityGroup ::=  (Modifier ":")? OperatorRule ("," OperatorRule)*

OperatorRule ::= BinaryOperatorRule | LeftUnaryOperatorRule | RightUnaryOperatorRule
BinaryOperatorRule ::= Head "::=" Operand Operator? Operand
LeftUnaryOperatorRule ::= Head "::=" Operator Operand
RightUnaryOperatorRule ::= Head "::=" Operand Operator

Operator ::= String | Id
```

lexical syntax

```
Head ::= Id
Operand ::= Id
Modifier ::= "left" | "right"
```

Listing A.4: The Priorities Module

module Basics

context-free syntax

lexical syntax

```
Id ::=[a-zA-Z][a-zA-Z0-9\-_]*
Digit ::= [0-9]+
String ::= ["]([^"\\]|[\\]["trnu\\])*["]
```

Listing A.5: The Basics module

Appendix B

Tom mapping for SPPF

```
%include {int.tom}

%typeterm SPPFNode {
  implement { nl.tue.win.set.jgll.sppf.SPPFNode }
  is_sort(t) { $t instanceof nl.tue.win.set.jgll.sppf.SPPFNode }
  equals(t1,t2) { ($t1.equals($t2)) }
}


%op SPPFNode SymbolNode(name:String, children:NodeList) {
  is_fsym(t) { ($t instanceof nl.tue.win.set.jgll.sppf.SymbolNode) }
  get_slot(name,t) { ((SymbolNode)$t).getSymbolName() }
  get_slot(children,t) { ($t.getChildren()) }
}

%op SPPFNode PackedNode(children:NodeList) {
  is_fsym(t) { ($t instanceof nl.tue.win.set.jgll.sppf.PackedNode) }
  get_slot(children,t) { ($t.getChildren()) }
}

%op SPPFNode IntermediateNode(children:NodeList) {
  is_fsym(t) { ($t instanceof nl.tue.win.set.jgll.sppf.IntermediateNode) }
  get_slot(children,t) { ($t.getChildren()) }
}

%typeterm NodeList {
  implement      { java.util.List<nl.tue.win.set.jgll.sppf.SPPFNode> }
  is_sort(t)     { $t instanceof java.util.List<?> }
  equals(l1,l2)  { $l1.equals($l2) }
}

%oplist NodeList concNode(SPPFNode*) {
  is_fsym(l)       { ($l!= null) && ($l instanceof java.util.List<?>) }
  make_empty()     { new java.util.LinkedList<nl.tue.win.set.jgll.sppf.SPPFNode>() }
  make_insert(o,l) { concListAppend($o,$l) }
  get_head(l)      { $l.get(0) }
  get_tail(l)      { concListgetTail($l) }
  is_empty(l)      { ($l.size()==0) }
}

private static <T> java.util.List<T> concListAppend(T o, java.util.List<T> l) {
```

92

```java
  java.util.List<T> res = new java.util.LinkedList<T>(l);
  res.add(0,o);
  return res;
}

private static <T> java.util.List<T> concListgetTail(java.util.List<T> l) {
  java.util.List<T> res = new java.util.LinkedList<T>(l);
  res.remove(0);
  return res;
}
```

Appendix C

An Introspector for SPPF mapping

```java
package nl.tue.win.set.jgll.sppf;

import tom.library.sl.Introspector;

public class SPPFIntrospector implements Introspector {

    @Override
    public Object getChildAt(Object o, int i) {
        return ((SPPFNode) o).getChildren().get(i);
    }

    @Override
    public int getChildCount(Object o) {
        return ((SPPFNode) o).getChildren().size();
    }

    @Override
    public Object[] getChildren(Object o) {
        return ((SPPFNode) o).getChildren().toArray();
    }

    @SuppressWarnings("unchecked")
    @Override
    public  Object setChildAt(Object o, int i, Object child) {
        return o;
    }

    @SuppressWarnings("unchecked")
    @Override
    public Object setChildren(Object o, Object[] children) {
        return o;
    }
}
```

Appendix D

An Ambiguous Pico Grammar in EBNF

---

module Pico

import meta-model pico = 'http://pico/1.0/base'
import meta-model ecore = 'http://www.eclipse.org/emf/2002/Ecore'

start symbol Program

context-**free syntax**

```
Program ::= "begin" "declare" (declarations:Declaration ("," declarations:Declaration)*)? ";" statements:Statement* "end"
{
    class(pico::Program),
    reference(declarations, declarations),
    reference(statements, statements)
}

Declaration ::=  id:PicoID ":" type:PicoType
{
    class(pico::Declaration),
    attribute(name, id),
    attribute(type,type)
}

PicoType ::= "string"  { enum(pico::Types#Natural) }
          | "natural" { enum(pico::Types#MyString) }

Statement ::= id:PicoID ":=" expr:Expr ";"
        {
            class(pico::AssignStatement),
            attribute(name,id),
            reference(expression,expr)
        }

    | "while" "(" condition:Expr ")" "do" body:Statement* "end"
        {
            class(pico::LoopStatement),
            reference(condition,condition),
            reference(body,body)
```

95

```
        }

    | "if" "(" condition:Expr ")" "then" if:Statement* "end"?
        {
            class(pico::ConditionalStatement),
            reference(condition,condition),
            reference(trueBranch,if)
        }

    | "if" "(" condition:Expr ")" "then" if:Statement* "else" else:Statement* "end"?
        {
            class(pico::ConditionalStatement),
            reference(condition,condition),
            reference(trueBranch,if),
            reference(falseBranch,else)
        }

Expr ::= natcon:PicoNatCon
        {
            class(pico::pico_expressions::NaturalExpression),
            attribute(value,natcon)
        }
    | strcon:PicoStrCon
        {
            class(pico::pico_expressions::StringExpression),
            attribute(value,strcon)
        }
    | id:PicoID
        {
            class(pico::pico_expressions::VariableReference),
            attribute(name,id)
        }

    | lhs:Expr operator:Plus rhs:Expr
        {
            class(pico::pico_expressions::BinaryExpression),
            attribute(operator,operator),
            reference(lhs,lhs),
            reference(rhs,rhs)
        }

    | lhs:Expr operator:Minus rhs:Expr
        {
            class(pico::pico_expressions::BinaryExpression),
            attribute(operator,operator),
            reference(lhs,lhs),
            reference(rhs,rhs)
        }

    | lhs:Expr operator:Concat rhs:Expr
        {
            class(pico::pico_expressions::BinaryExpression),
            attribute(operator,operator),
            reference(lhs,lhs),
            reference(rhs,rhs)
        }
    | "(" expr:Expr ")"
        {
            propagate(expr)
        }
```

```
Plus ::= "+"  { enum(pico::pico_expressions::Operators#Addition) }

Minus ::= "-"  { enum(pico::pico_expressions::Operators#Subtraction) }

Concat ::=  "||" { enum(pico::pico_expressions::Operators#Concatenation) }

PicoID ::= Id  { type(ecore::EString) }
PicoStrCon ::= StrCon { type(ecore::EString) }
PicoNatCon ::= NatCon { type(ecore::EIntegerObject) }
```

lexical syntax

```
Id ::= [A-Za-z][A-Za-z0-9]*
NatCon ::= [0-9]+
StrCon ::= ["]([^"\\]|[\\]["trnu\\])*["]
```

layout syntax

```
Whitespace ::= [\ \t \n \r]+
MultilineComment ::= [/][*]([^*]|[*]+[^/*])*[*]+[/]
SingleLineComment ::= [/][/][^\r\n]*
```

disambiguation rules

prefer ["if", "(", Expr, ")", "then", _*],
       ["if", "(", Expr, ")", "then", _*, "else", _*]

prefer [Statement("if", "(", Expr, ")", "then", Statement), _*],
       [Statement("if", "(", Expr, ")", "then", Statement, _*)]

associativity and priority
```
{
left: Expr ::= Expr Concat Expr >
left: Expr ::= Expr Minus Expr, Expr ::= Expr Plus Expr
}
```
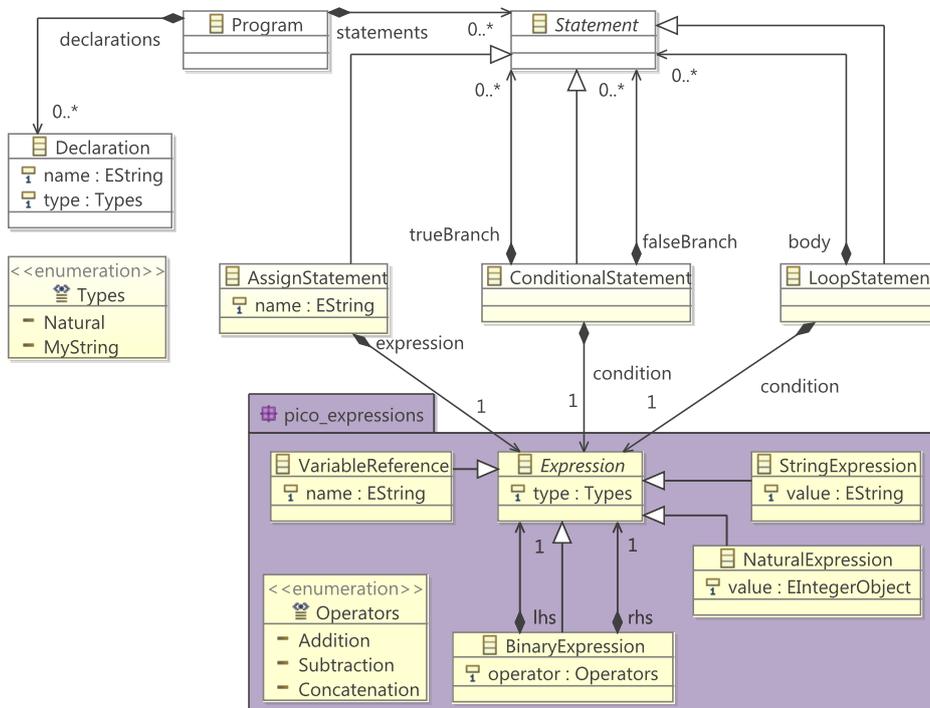
Figure D.1: A metamodel for Pico

Appendix E

Disambiguation rules for mCRL2

---

disambiguation rules

remove [ProcExpr("sum", VarsDeclList, ".", ProcExpr), ".", ProcExpr]

remove ["sum", VarsDeclList, ".", ProcExpr(ProcExpr, "+", ProcExpr)]

remove [ProcExpr("sum", VarsDeclList, ".", ProcExpr), "|", ProcExpr]

remove [ProcExpr("sum", VarsDeclList, ".", ProcExpr), "@", DataExprUnit]

remove [DataExpr(DataExpr, _, _, DataExpr), "(", DataExprList, ")"]

remove [DataExpr("lambda", VarsDeclList, ".", DataExpr), _*]

remove [ProcExpr(DataExprUnit, "->", ProcExpr), ".", ProcExpr]

remove [ProcExpr(DataExprUnit, "->", ProcExpr), "@", DataExprUnit]

remove [ProcExpr(ProcExpr, "+", ProcExpr), "@", DataExprUnit]

remove [ProcExpr(ProcExpr, ".", ProcExpr), "@", DataExprUnit]

remove [ProcExpr(DataExprUnit, IfThen, ProcExpr), ".", ProcExpr]

remove [DataExprUnit, "->", ProcExpr(ProcExpr, "+", ProcExpr)]

remove [DataExprUnit, IfThen, ProcExpr(ProcExpr, "+", ProcExpr)]

remove [DataExpr("!", DataExpr), "(", DataExprList, ")"]

remove [DataExpr("-", DataExpr), "(", DataExprList, ")"]

remove [DataExpr("#", DataExpr), "(", DataExprList, ")"]

remove [ProcExprNoIf("sum", VarsDeclList, ".", ProcExprNoIf), ".", ProcExprNoIf]

remove [DataExpr(DataExpr, _, _, DataExpr), "whr", AssignmentList, "end"]

remove ["exists", VarsDeclList, ".", DataExpr(DataExpr, "(", DataExprList ,")")]

remove ["exists", VarsDeclList, ".", DataExpr(DataExpr, ">", DataExpr)]

remove ["exists", VarsDeclList, ".", DataExpr(DataExpr, "+", DataExpr)]

remove ["forall", VarsDeclList, ".", DataExpr(DataExpr, "+", DataExpr)]

remove ["forall", VarsDeclList, ".", DataExpr(DataExpr, ">", DataExpr)]

remove [DataExprUnit("!", DataExprUnit), "(", DataExprList, ")"]

associativity and priority

```
{
        DataExpr ::= "!" DataExpr,
        DataExpr ::= "-" DataExpr,
        DataExpr ::= "#" DataExpr    >

left:   DataExpr ::= DataExpr "*"  DataExpr,
        DataExpr ::= DataExpr "."  DataExpr >

left:   DataExpr ::= DataExpr "/"   DataExpr,
        DataExpr ::=  DataExpr "div" DataExpr,
        DataExpr ::= DataExpr "mod" DataExpr    >

left:   DataExpr ::= DataExpr "+"   DataExpr,
        DataExpr ::= DataExpr "-"   DataExpr    >

left:   DataExpr ::= DataExpr "++"  DataExpr    >

left:   DataExpr ::=  DataExpr "<|"  DataExpr   >

right:  DataExpr ::= DataExpr "|>"  DataExpr    >

left:   DataExpr ::= DataExpr "<"   DataExpr,
        DataExpr ::= DataExpr "<="  DataExpr,
        DataExpr ::= DataExpr ">="  DataExpr,
        DataExpr ::= DataExpr ">"   DataExpr,
        DataExpr ::= DataExpr "in"  DataExpr    >

left: DataExpr ::=  DataExpr "=="  DataExpr,
      DataExpr ::= DataExpr "!="  DataExpr   >

right: DataExpr ::= DataExpr "&&"  DataExpr >

right: DataExpr ::= DataExpr "||"  DataExpr >

right: DataExpr ::= DataExpr "=>"  DataExpr
}

{
DataExprUnit ::= "!" DataExprUnit >
DataExprUnit ::= "-" DataExprUnit >
DataExprUnit ::= "#" DataExprUnit
}

{
left: ProcExpr ::= ProcExpr "|"   ProcExpr >

right: ProcExpr ::= ProcExpr "."   ProcExpr >
```

100

```
left: ProcExpr ::= ProcExpr "<<"  ProcExpr  >

right:  ProcExpr ::= ProcExpr "||"  ProcExpr,
        ProcExpr ::= ProcExpr "||_" ProcExpr >

left: ProcExpr ::= ProcExpr "+"    ProcExpr
}

{
left: ProcExprNoIf ::= ProcExprNoIf "|"    ProcExprNoIf >

right: ProcExprNoIf ::= ProcExprNoIf "."    ProcExprNoIf >

left: ProcExprNoIf ::= ProcExprNoIf "<<"  ProcExprNoIf >

right:  ProcExprNoIf ::= ProcExprNoIf "||"  ProcExprNoIf,
        ProcExprNoIf ::= ProcExprNoIf "||_" ProcExprNoIf >

left: ProcExprNoIf ::=  ProcExprNoIf "+"    ProcExprNoIf
}

{
        BesExpr ::= "!" BesExpr     >

right:  BesExpr ::= BesExpr "&&" BesExpr   >

right:  BesExpr ::= BesExpr "||" BesExpr    >

right:  BesExpr ::= BesExpr "=>" BesExpr
}

{
        PbesExpr ::= "!" PbesExpr >

right:  PbesExpr ::= PbesExpr "||" PbesExpr >

right:  PbesExpr ::= PbesExpr "&&" PbesExpr >

right:  PbesExpr ::= PbesExpr "=>" PbesExpr
}

{
        ActFrm ::= "!" ActFrm >

right: ActFrm ::= ActFrm "||" ActFrm >

right: ActFrm ::= ActFrm "&&" ActFrm >

right: ActFrm ::= ActFrm "=>" ActFrm
}

{
        RegFrm ::= RegFrm "+",
        RegFrm ::= RegFrm "*"    >

left:   RegFrm ::= RegFrm "+" RegFrm     >

right:  RegFrm ::= RegFrm "." RegFrm
}

{
```

```
           StateFrm ::= "!" StateFrm >

right:  StateFrm ::= StateFrm "&&" StateFrm >

right:  StateFrm ::= StateFrm "||" StateFrm >

right:  StateFrm ::= StateFrm "=>" StateFrm
}
```

Appendix F

An MEBNF grammar for Tom

---

module Tom

start symbol Program

context-free syntax

Program ::= Chunk*

Chunk ::= Water | Island

Water ::= "{" Chunk* "}" | Identifier | Integer |
          String | Character | SpecialChar | SpecialChar2 |
          CamlChar | "," | "." | "(" | ")"

Island ::= TomConstruct | BackQuoteTerm

TomConstruct ::= IncludeConstruct
              | MatchConstruct
              | GomConstruct
              | StrategyConstruct
              | OperatorConstruct
              | OperatorListConstruct
              | OperatorArrayConstruct
              | TypeTermConstruct
              | MetaQuoteConstruct

IncludeConstruct ::= "%include" "{" Water* "}"

BackQuoteTerm ::= "`" CompositeTerm | "`" "(" CompositeTerm+ ")"

CompositeTerm ::= VariableStar
              | Variable
              | Identifier "(" (CompositeTerm+ ("," CompositeTerm+)*)? ")"
              | "." Identifier "(" CompositeTerm* ")"
              | BackQuoteWater

BackQuoteWater ::= Identifier | Integer | String |
                  Character | SpecialChar | SpecialChar2 | CamlChar

```
MatchConstruct ::= "%match" ( "(" MatchArguments ")" )? "{" PatternAction* "}"

MatchArguments ::= Type? Term ("," Type? Term)*

PatternAction ::= PatternList "->" "{" Chunk* "}"

ConstraintAction ::= Constraint "->" "{" Chunk* "}"

Term ::= Identifier
     | VariableStar
     | Identifier "(" ( Term ("," Term)* )? ")"


GomConstruct ::= "%gom" ( "(" Water* ")" )? "{" GomGrammar "}"

GomGrammar ::= Module

Module ::= "module" ModuleName Imports? Grammar

Imports ::= "imports" ModuleName+

Grammar ::= "abstract" "syntax" (TypeDefinition | HookDefinition)*

TypeDefinition ::= SortName "=" OperatorDefinition ("|" OperatorDefinition)*

OperatorDefinition ::= OperatorName "(" ( SlotDefinition ("|" SlotDefinition)* )? ")"
                    | OperatorName "(" Variable "*" ")"

SlotDefinition ::= SlotName ":" SlotName

HookDefinition ::= HookType ":" HookOperation

HookType ::= OperatorName
          | "module" ModuleName
          | "sort" SortName

HookOperation ::= "make" "("  ( Identifier ("," Identifier)* )? ")" "{" Chunk* "}"
               | "make_insert" "(" Identifier "," Identifier ")" "{" Chunk* "}"
               | "make_empty" "(" ")" "{" Chunk* "}"
               | "Free" "(" ")" "{" "}"
               | "FL" "(" ")" "{" "}"
               | ("AU" | "ACU") "(" ")" "{" ("`" Term)? "}"
               | "interface" "(" ")" "{" Identifier ("," Identifier)* "}"
               | "import" "(" ")" "{" Water* "}"
               | "block" "(" ")" "{" Chunk* "}"
               | "rules" "(" ")" "{" RulesCode "}"
               | "graphrules" "(" Identifier "," ("Identity" | "Fail") ")" "{" GraphRulesCode "}"

RulesCode ::= Rule*

Rule ::= Pattern "->" Term ("if" Condition)?

Pattern ::= (AnnotatedName "@")? PlainPattern

PlainPattern ::= "!"? (Variable | VariableStar | ConstantValue)
              | "!"? HeadSymbolList "(" ( Pattern ( "," Pattern )* )? ")"

GraphRulesCode ::= GraphRule*

GraphRule ::= TermGraph "->" TermGraph ("if" Condition)?
```

```
TermGraph ::= (Label ":")? TermGraph
           | "&" Label
           | Variable "*"
           | HeadSymbolList "(" (TermGraph ("," TermGraph)* )? ")"


Condition ::= "(" Condition ")"
           | Condition "||" Condition
           | Condition "&&" Condition
           | Pattern "<<" Term
           | Term Operator Term

OperatorConstruct ::= "%op" Type Name "(" ( SlotName ":" Type ( "," SlotName ":" Type )*)?  ")"
                                      "{" ( KeywordIsFsym |
                                            KeywordMake |
                                            KeywordGetSlot |
                                            KeywordGetDefault)* "}"

OperatorListConstruct ::= "%oplist" Type Name "(" Type "*" ")"
                                      "{" KeywordIsFsym ( KeywordMakeEmptyList |
                                                          KeywordMakeInsert |
                                                          KeywordGetHead |
                                                          KeywordGetTail |
                                                          KeywordIsEmpty )* "}"

OperatorArrayConstruct ::= "%oparray" Type Name "(" Type "*" ")"
                                   "{" KeywordIsFsym (KeywordMakeEmptyArray |
                                                      KeywordMakeAppend |
                                                      KeywordGetElement |
                                                      KeywordGetSize)* "}"


KeywordIsFsym ::= "is_fsym" "(" Name ")" "{" Water+ "}"

KeywordGetSlot ::= "get_slot" "(" Name "," Name ")" "{" Water+ "}"

KeywordGetDefault ::= "get_default" "(" Name ")" "{" Water+ "}"

KeywordMake ::= "make" ( "(" Name ("," Name)* ")" )? "{" Water+ "}"

KeywordGetHead ::= "get_head" "(" Name ")" "{" Water+ "}"

KeywordGetTail ::= "get_tail" "(" Name ")" "{" Water+ "}"

KeywordMakeEmptyList ::= "make_empty" ( "(" ")" )? "{" Water+ "}"

KeywordMakeInsert ::= "make_insert" "(" Name "," Name ")" "{" Water+ "}"

KeywordGetElement ::= "get_element" "(" Name "," Name ")" "{" Water+ "}"

KeywordGetSize ::= "get_size" "(" Name ")" "{" Water+ "}"

KeywordIsEmpty ::= "is_empty" "(" Identifier ")" "{" Water+ "}"

KeywordMakeEmptyArray ::= "make_empty" "(" Name ")" "{" Water+ "}"

KeywordMakeAppend ::= "make_append" "(" Name "," Name ")" "{" Water+ "}"

TypeTermConstruct ::= "%typeterm" Type ("extends" Type)?
                        "{" KeywordImplement KeywordIsSort? KeywordEquals? "}"

KeywordImplement ::= "implement" "{" Water+ "}"
```

```
KeywordIsSort ::= "is_sort" "(" Name ")" "{" Water+ "}"

KeywordEquals ::= "equals" "(" Name "," Name ")" "{" Water+ "}"

StrategyConstruct ::= "%strategy" StrategyName "(" StrategyArguments? ")"
                      "extends" "`"? Term "{" StrategyVisit* "}"

StrategyArguments ::= SubjectName ":" AlgebraicType ("," SubjectName ":" AlgebraicType)*
                    | AlgebraicType SubjectName ("," AlgebraicType SubjectName )*

StrategyVisit ::= "visit" AlgebraicType "{" VisitAction* "}"

VisitAction ::= (Label ":")? PatternList "->" "{" Chunk* "}"

PatternList ::= Pattern ("," Pattern)* (("&&" | "||") Constraint)?

Constraint ::= Pattern "<<" Type Term
             | Constraint "&&" Constraint
             | Constraint "||" Constraint
             | "(" Constraint ")"
             | Term Operator Term

HeadSymbolList ::= HeadSymbol "?"?
                 | "(" HeadSymbol ("|" HeadSymbol)+ ")"

ExplicitTermList ::= "(" (Pattern ("," Pattern)* )? ")"

ImplicitPairList ::= "[" (PairPattern ("," PairPattern)* )? "]"

PairPattern ::= Identifier "=" Pattern

ConstantValue ::= String
                | Integer
                | Character
                | Float

MetaQuoteConstruct ::= "%[" Chunk* "]%"
```

lexical syntax

```
StrategyName ::= Identifier
AlgebraicType ::= Identifier
Type ::= Identifier
Variable ::= Identifier
ModuleName ::= Identifier
SortName ::= Identifier
AnnotatedName ::= Identifier
SubjectName ::= Identifier
Label ::= Identifier
OperatorName ::= Identifier
Name ::= Identifier
SlotName ::= Identifier

VariableStar ::= Identifier "*"
HeadSymbol ::= Identifier | Integer | Float | String | AnyChar
SpecialChar ::= [;:+\-=&<>!%:?!&@\[\]\^#$]
CamlChar ::= Identifier "'"
SpecialChar2 ::= [*/]+ -/- { "//".*, "/*".*}
Operator ::= ">" | ">=" | "<" | "<=" | "==" | "!="
Identifier ::= [a-zA-Z_]+[a-zA-Z0-9\-_]*
String ::= ["]([^"\\]|[\\]["trnu\\])*["]
```

```
Character ::= ['].['] | ['][\\] EscapeChar [']
EscapeChar ::= [btnfr'\\]
AnyChar ::= .
Integer ::= [0-9]+
Float ::= [1-9]+[\.][0-9]+
```

layout syntax

```
WhiteSpace ::= [\ \r \n \t]+
MultilineCommentCStyle ::= [/][*]([^*]|[*]+[^/*])*[*]+[/]
MultilineCommentCamlStyle ::= [(][*]([^*]|[*]+[^)*])*[*]+[)]
SingleLineComment ::= [/][/][^\r\n]*
```

disambiguation rules

prefer [Chunk(Island)], [Chunk(Water), _*]

prefer [Variable], [BackQuoteWater]

prefer [Chunk(Island(BackQuoteTerm))],
        [Chunk(Island(BackQuoteTerm)), Chunk(Water("*")))]

prefer [Chunk(Island(BackQuoteTerm))],
        [Chunk(Island(BackQuoteTerm)), Chunk(Water("(")))]
```