Master Thesis

Automated ANTLR Tree walker Generation

UNIVERSITY OF TWENTE.

Author:	A.J. Admiraal
Email:	alex@admiraal.dds.nl
Institute:	University of Twente
Chair:	Formal Methods and Tools
Date:	January 25, 2010

Automated ANTLR Tree walker Generation

Supervisors

dr. ir. Theo Ruys
prof. dr. ir. Joost-Pieter Katoen
dr. ir. Arend Rensink

Abstract

ANTLR is a parser generator that allows the user to write flexible multi-pass language parsers. It can generate a parser that builds an abstract syntax tree from the parsed text, and it can generate tree walkers that can parse this tree. A typical ANTLR multi-pass language parser consists of one or more tree walkers, which can be used as checker, optimiser and/or code generator.

Ensuring a tree walker correctly matches the abstract syntax trees produced by the parser is a manual task. When a change in the parser specification affects the abstract syntax tree, all tree walkers need to be inspected and changed accordingly. Mistakes can lead to nontrivial errors and may therefore be hard to find and solve. This can be a significant problem, both for experienced and inexperienced users, both during development and maintenance of a language parser.

The root-cause of this problem lies in the redundancy between the ANTLR parser and tree walker specifications. Both specify the syntax of the abstract syntax tree; the parser specifies how it is to be created and the tree walker how it is to be parsed.

This thesis introduces ANTLR$_{TG}$, an extension for ANTLR that allows the user to automatically generate a matching tree walker based on the specification of an ANTLR parser. An algorithm has been created, using a tree pattern algebra, that can determine a tree walker that is able to parse all possible tree walkers that can be generated by a given parser. ANTLR$_{TG}$ has been fully implemented and is demonstrated through a case-study that implements a compiler for the Triangle language.

Preface

The past few years, I have used ANTLR to build several parsers and compilers. Several times I ran into the problem that I have identified in this thesis, but it was only during the first months of my master project that I found a potential solution for it. After discussing this with Theo Ruys, my primary supervisor, we decided to change the scope of the project and I started implementing and formalising the algorithm.

This is when I learned that writing an algorithm in code is one thing, but writing an algorithm on paper is something completely different. It took me quite some time to get the algorithm from the idea and the proof-of-concept in code to what it is today on paper, and I learned a lot of new skills along the way.

Halfway through my project, I got a job offer at Philips Healthcare. As part of my Bachelor degree, I have done several projects at the Academic Medical Center in Amsterdam, which built up my affinity with Healthcare. Therefore, Philips Healthcare was, and had been for a while, on the top of my list of companies where I would like to work, I therefore decided I should accept this offer and continue my project part-time.

I would like to thank my supervisors Theo Ruys, Joost-Pieter Katoen and Arend Rensink for their support. A special thanks to Theo Ruys for reading and correcting this thesis and for the help and ideas that made it possible for me to formalise the algorithm. Finally, I wish to thank my colleagues at Philips Healthcare for pushing me to keep working on this thesis.

Contents

List of Figures

Chapter 1

Introduction

Ever since the first compiler was created by Grace Hopper in 1952 for the A-0 programming language [1], compilers have become more and more prominent tools for writing computer software. Over the years the responsibilities of a compiler have been extended significantly. The early compilers simply translated a piece of human readable source code into a machine language, where the higher level programming languages were relatively close to the machine languages. Modern compilers have to parse significantly more complex languages, do contextual analysis to provide the programmer with helpful error and warning messages and do complex optimisations.

Today, the divide and conquer paradigm [2] is used on many large-scale compilers; the compilation task is split up into a series of simple tasks, or phases, which can be executed separately. This design strategy was adopted by the Production Quality Compiler-Compiler Project (PQCC) at Carnegie Mellon University [3]. Due to the separation of the different responsibilities of the compiler, it becomes possible to develop complex compilers for complex languages such as C++, Java and C#. Today there are many compiler-compilers, or parser generators, available.

This thesis focuses on one specific parser generator: ANTLR [4]. One of the key features of ANTLR is that the format it uses to specify the syntax of a language is very similar to EBNF [5]. This chapter will briefly introduce ANTLR and then states the problem that we are going to solve.

1.1 ANTLR

ANTLR [6] [4] is a parser generator that can parse language specifications in an EBNF-like syntax and generate C, C++, Java, Python, C# or Objective-C source code for a parser that parses the specified language. The EBNF language specification can be annotated with source code during the parsing process. ANTLR distinguishes three compilation phases; the lexical analysis phase, the parsing phase and tree walking phases. An ANTLR generated compiler usually at least implements the lexical analysis and parsing phases, and can optionally add one or more tree walking phases. A few examples for tree walking phases are checking the logical correctness, optimisation and generating assembly code.

1.1.1 Compilation phases

A typical compiler first uses the lexical analyser, or "Lexer", to analyse the input file. The lexical analyser is responsible for grouping the characters from the input file, or character stream, into a stream of words or tokens. Using the token stream as an input; the parser then parses this token stream and creates a tree of tokens, called an "Abstract Syntax Tree" or AST. After this phase, one or more tree walking phases can use and modify this AST. Figure 1.1 schematically demonstrates this concept. Modification of the AST is an important part of the tree walking phases, for example the phase that checks the logical correctness, or "Checker", might annotate all variables with a reference to their declaration. And the optimisation phase might change a construction that models "`a = 1 + 2`" into "`a = 3`".

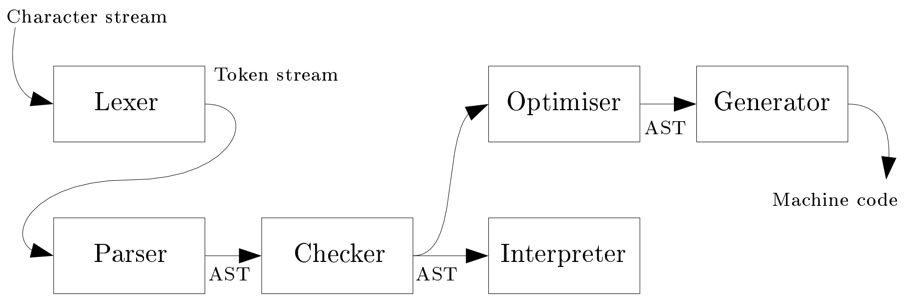


Figure 1.1: Several phases of a compiler

The lexical analyser

The lexical analyser is responsible for converting the character stream into a token stream. For ANTLR, the EBNF specification of the lexical analyser specifies exactly how tokens can be recognised. Lexical analysis is a well understood process, more information on this can be found in many places, for example in [7]. This thesis will focus mainly on the parsing and tree walking phases and the AST data structures in between. For more information on the lexical analyser as provided by ANTLR, please refer to [8].

The parser

The parser is responsible for creating the initial AST from the token stream that the lexical analyser has produced. The EBNF specification of the language is used to determine the structure of the tree, it can be annotated with special tree shaping operators to control the shape of the AST. An example parser specification is demonstrated in figure 1.3.

The tree walkers

A tree walker is used to process the AST that has been produced by the parser. Basically a tree walker is a parser that parses an AST instead of a token stream. Checking phases or optimisation phases are usually implemented using tree walkers. To let ANTLR generate a tree walker, one has to write an EBNF tree-walker specification for the AST that has been produced by the parser. Because the shape of the AST depends on the input sentences and the parser specification, writing a tree walker specification requires specific knowledge on both. An example tree walker specification is demonstrated in figure 1.5.

1.1.2 Abstract syntax trees

An Abstract Syntax Tree, or "AST", is a finite, directed, labelled tree, where each node represents a programming language construct, the children of each node represent components of the construct. Figure 1.2 demonstrates a simple AST for the assignment statement "`a = 1 + 2`".

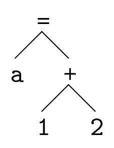


Figure 1.2: An AST for the assignment statement "`a = 1 + 2`"

When the compiler tasks are separated by multiple phases, an AST is often used as an intermediate structure between the parsing phase and the other phases. The AST should only contain relevant information, where relevant is determined by the language that is being parsed. For example, for the programming language C, semicolons and commas are discarded, where parentheses can usually only be recognised by the shape of the tree. For example "`a = 1 + 2;`" and "`a = (1 + 2);`" would most likely be parsed to the same AST as depicted in figure 1.2.

1.1.3 EBNF

Extended Backus-Naur form, or "EBNF", is a commonly used notation for context-free grammars and is an ISO/IEC standard [5]. EBNF was originally developed by Niklaus Wirth [9] based on the Backus-Naur form, or "BNF", notation that was developed by John Backus and Peter Naur. EBNF is intended to be extensible, this means that many variants of EBNF currently exist.

EBNF syntax for ANTLR parsers

ANTLR uses a specialised variant of EBNF, where the operators for repetition, concatenation, option, etc. have been replaced by operators that are more commonly used in regular expressions. This variant of EBNF is very similar to the variant of EBNF that the W3C uses to specify the XML syntax [10]. Table 1.1 explains the operators that ANTLR uses in its EBNF syntax.

All references to EBNF hereafter refer to the ANTLR specific version of EBNF, unless otherwise noted. Please refer to [8] for all details on this version of EBNF. Figure 1.3 shows a fragment of an EBNF specification for a simple language. Assuming that `value` matches decimal numbers and `identifier` literal identifiers, the grammar fragment matches sentences like: "`a + b + 3 + -c`" and "`1 + -(a + b)`".

Besides the operators demonstrated above, ANTLR provides tree shaping operators in its EBNF extensions. Table 1.2 briefly explains these operators. When it is the intention to use the AST,

ANTLR	EBNF	Explanation
`A B`	`A, B`	Matches `A` followed by `B`.
`A | B`	`A | B`	Matches `A` or `B`.
`A?`	`[A]`	Matches zero or one occurrences of `A`.
`A+`	`A, {A}`	Matches one or more occurrences of `A`.
`A*`	`{A}`	Matches zero or more occurrences of `A`.

Table 1.1: ANTLR EBNF operators versus the original EBNF operators

```
expression
  : operand ("+" operand)*
  ;

operand
  : ("-")? operand
  | "(" expression ")"
  | value
  | identifier
  ;
```

Figure 1.3: An example ANTLR parser specification

that is generated by the parser, as input for a tree walker, it may be desirable to use these tree shaping operators provided by ANTLR. Figure 1.4 demonstrates the grammar from figure 1.3 annotated with the tree shaping operators provided by ANTLR. In this grammar the add (+) and invert (-) operators have been annotated with a double-caret (`^^`) denoting that they should form a node in the AST, the other elements shall form leafs. The parentheses have been annotated with an exclamation mark (`!`) to indicate that they are not relevant and should not be taken into the AST at all.

Operator	Explanation
`A^^`	`A` becomes the parent node for this EBNF rule.
`A^`	`A` becomes the parent node for this section.
`A!`	`A` should not be taken into the AST.

Table 1.2: ANTLR tree shaping operators

If the sentence "`1 + -(a + b)`" were to be parsed by a parser generated from the EBNF grammar from figure 1.4, the AST from figure 1.6 would be generated.

EBNF syntax for ANTLR tree walkers

The EBNF syntax that ANTLR uses for tree-walkers is slightly different from the one used for parsers. The tree shaping operators are not used for tree walkers, a tree operator is introduced instead: "`^()`". The first terminal or non-terminal between the parentheses of the tree operator is considered to be a parent in the AST subtree currently being parsed. The other terminals

```
expression
  : operand ("+"^^ operand)*
  ;

operand
  : ("-"^^)? operand
  | "("! expression ")"!
  | value
  | identifier
  ;
```

Figure 1.4: An EBNF grammar fragment with tree shaping operators

or non-terminals are considered to be the children of this parent. Figure 1.5 shows the EBNF specification for the tree walker that is able to parse all ASTs that can be generated by the parser specification of figure 1.4.

```
expression
  : ^("+" expression expression)
  | ^("-" expression)
  | value
  | identifier
  ;
```

Figure 1.5: A tree walker specification

For example, consider the AST of figure 1.6 with the tree walker specification of figure 1.5. The root of the AST fires the first alternative of the tree walker specification. This rule demands a node "+" with two children conforming to the alternative `expression`, in this case the left child is a value and the right child is a node "`-`" with one child. This child is again a node "+" with two children, both conforming to the alternative `identifier`.

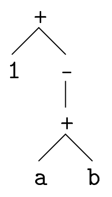


Figure 1.6: An AST for the sentence "`1 + -(a + b)`"

1.1.4 Concrete Syntax Trees

The Concrete Syntax Tree, or "CST", is used to model the parsing of a grammar. In a non-empty Concrete Syntax Tree, the root node is always the start symbol of the grammar that was used to parse a sentence. The leafs consists of all terminals that were encountered while parsing a sentence. The paths from the leafs to the root in the CST contain all encountered non-terminals that were found while parsing the sentence. The CST is also known as a “parse tree”.

A CST is useful for demonstrating how a sentence is related to a grammar. For example, figure 1.7 demonstrates a CST for the sentence “`1 + -(a + b)`” as parsed by the grammar of figure 1.3. For every terminal one can trace back all non-terminals to the start symbol. More information on CSTs can be found in any compiler construction textbook, such as [7].

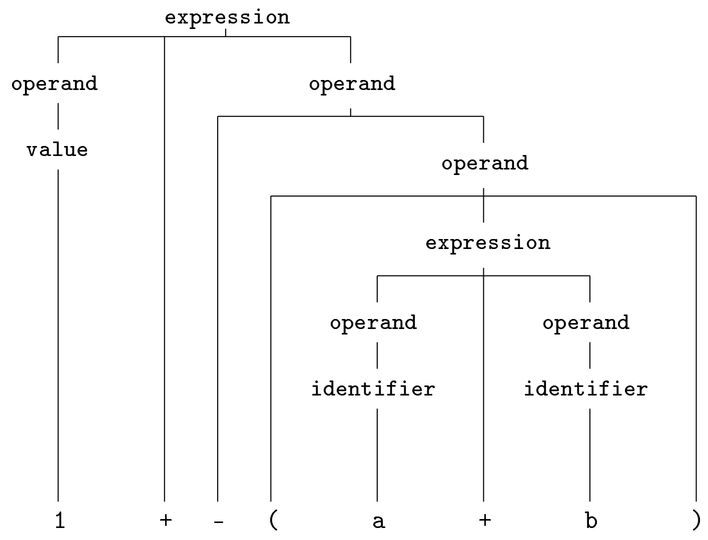


Figure 1.7: A CST for the sentence “`1 + -(a + b)`” as parsed by the grammar of figure 1.3

1.2 Problem description

Language parsers and translators are widely used software components; from relatively simple parsers for configuration files to extremely complex parsers such as C++ compilers. For many languages, one or two phases, or passes, are sufficient. ANTLR excels in ease of use for these kinds of languages. There is a group of languages that need many different passes over the AST. For example, highly optimised C or C++ compilers. For this approach, using ANTLR is not ideal.

As explained in section 1.1.1, the tree walker can be seen as a parser for the AST that was produced by the previous phase. While writing the tree walker specification, the developer needs to know exactly how the AST is shaped with respect to the language specification. Which requires detailed knowledge on the parser specification and the way tree shaping operators behave. This leads to several research questions:

- With each change of the parser specification, all tree walker specifications that rely on this parser specification have to be changed accordingly. Is it possible to automatically update tree walker specifications after changes on the parser specification?

- A parser specification holds all information on the Abstract Syntax Trees that can be generated. Is it possible to automatically generate the ANTLR tree walker specification, given the ANTLR parser specification?
- A parser specification should precisely define the sentences of the source language. A tree walker, on the other hand, does not have to follow the source language precisely. Consequently, it is common practise to create tree walker specifications that are much more generic and concise than the corresponding parser specification. Is it possible to automate this simplification process?

All information that is needed to build a tree walker specification is already present in the parser specification. Therefore it should be possible to automatically generate a bare tree walker specification based on the parser specification. This thesis will try to create an algorithm to generate a skeleton tree walker specification from a parser specification.

1.3 Overview of the thesis

Chapter 2 will introduce some necessary background information and introduce some related work. Chapter 3 will provide an algorithm that is able to extract one or more tree walkers from an ANTLR parser specification. This chapter will provide a detailed overview of the issues that arise when extracting the required information from the parser specification and formalises the algorithm using a tree pattern algebra. Chapter 4 will introduce several methods that can be used to generalise the generated tree walker specifications to improve their practical usability.

Chapter 5 will introduce an implementation for the solution mentioned in chapters 3 and 4. This solution will then be evaluated for its practical usability in the case study described in chapter 6. This case study involves the implementation of a simple translator which consists of a parser and two tree walkers.

Finally, chapter 7 provides a small summary of the results, conclusions and an overview of any future work.

Chapter 2

Preliminaries and related work

2.1 Language parsing

2.1.1 Grammars

Ever since the Algol project [11], the formal specification of a language almost always involves some form of a context-free grammar [12]. The context-free grammar is usually used to specify the syntax of a language, constructions that can not be specified using a context-free grammar are usually considered the semantics of a language. For example the sentence "a = 1;" is for many languages a syntactical correct sentence. However, if the variable "a" is not defined within the scope of the statement; the sentence is semantically incorrect for many languages, because "a" was not defined.

Definition 2.1.1 presents the context-free grammar as defined by [12] with some minor adaptations to suit the context-free grammars defined in ANTLR. Figure 2.1 demonstrates an example context-free grammar.

Definition 2.1.1 *A context-free grammar G is denoted by a 4-tuple (N, T, R, S), where N is a finite nonempty set of non-terminals, T a finite, nonempty set of terminals and $S \in N$ the start symbol. The alphabet of G is denoted by $N \cup T$. R is a finite set of production rules: for each $r \in R : n \to \alpha$, where $n \in N$. Here α can be defined using EBNF:*
α `=` a `|` A `|` `(`α`)op` `|` $\alpha_1\alpha_2$ `|` $\alpha_1|\alpha_2$.
`op = * | + | ? | .`
with $a \in N$ and $A \in T$.

To accommodate the tree shaping operators, as explained in section 1.1.3, the definition of the context-free grammar will be extended with the annotations in definition 2.1.2. Any reference hereafter to a parser specification will refer to an annotated context-free grammar. Figure 2.2 annotates the context-free grammar of figure 2.1 with tree shaping operators.

Definition 2.1.2 *An annotated context-free grammar G' is denoted by a 4-tuple (N, T', R, S), where N is a finite nonempty set of non-terminals, T' a finite nonempty set of annotated terminals and $S \in N$ the start symbol. The alphabet of G' is denoted by $N \cup T'$. R is a finite set of*

```
start
  : ((A B C)* D rule)* (K L M)?
  ;

rule
  : (E F G (rulesub)*)+
  | K L M
  ;

rulesub
  : H I J
  ;
```

Figure 2.1: A context-free grammar

production rules: for each $r \in R : n \rightarrow \alpha$, where $n \in N$. Each $t \in T'$ can optionally be annotated with either `!` *or* `^^`*. α can be defined using EBNF:*

α = a | A | (α)op | $\alpha_1\alpha_2$ | $\alpha_1|\alpha_2$.

op = * | + | ? | .

with $a \in N$ and $A \in T'$. The annotations `^^` *and* `!` *on terminals declare the following behaviour:*

`!` *Each terminal that is annotated with an exclamation mark will not be present in the AST.*

`^^` *When parsing from left to right, any terminal that is annotated by a double-caret (*`^^`*) becomes the root node of the section that has formed until there. Any subsequent terminals and non-terminals become children of the last terminal annotated with a double-caret.*

```
start
  : ((A B^^ C)* D rule)* (K^^ L M)?
  ;

rule
  : (E! F^^ G (rulesub)*)+
  | K L^^ M
  ;

rulesub
  : H I^^ J
  ;
```

Figure 2.2: An annotated context-free grammar, based on figure 2.1

2.1.2 Abstract Syntax Trees

The AST plays an important role in the algorithms described in this thesis. A brief introduction to the AST was already given in section 1.1.2, this section will define the AST and explains how an AST is constructed by a parser. A large amount of information is available on Abstract

Syntax Trees, for example in [7]. Definition 2.1.3 is used as the definition of an AST in this thesis.

Definition 2.1.3 *An Abstract Syntax Tree, or "AST" is a finite, directed, labelled tree, where each node represents a programming language construct, the children of each node represent components of the construct. An "AST section" is a part of the AST that was formed by exactly one rule of a grammar.*

An AST that is produced by a grammar consists of several sections, or sub-trees. Every rule in a grammar produces one or more sections, they are interconnected by the section of the rule associated with the start symbol. A basic AST is build up by processing a grammar specification G with an input sentence S, denoted by $G \uparrow S$. Every terminal parsed by a rule of G results in a node in the section for that rule. Every non-terminal parsed by a rule of G results in a new section, which is connected to the section of the rule containing the non-terminal by its root. When a grammar is unambiguous, every sentence parsed by the grammar results in a unique AST.

Consider the annotated context-free grammar G' from figure 2.2 and sentence S_1 = "`A B C A B C D E F G H I J K L M`". While parsing S_1, the `start` rule accepts the terminals `A`, `B^^`, `C`, ..., `D`, `rule`, `K`, `L^^`, `M`. The `rule` rule accepts the terminals `E`, `F^^`, `G` and the `rulesub` rule accepts the terminals `H`, `I^^`, `J`. Figure 2.3 demonstrates the parse tree for this.

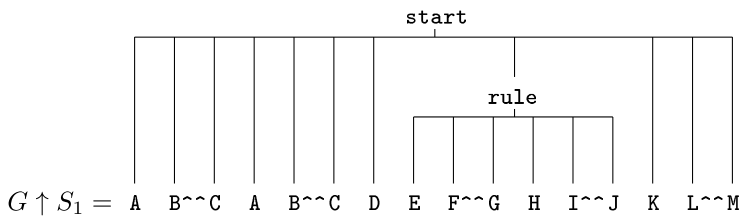


Figure 2.3: The sentence S_1 parsed by the grammar G from figure 2.2

2.2 Tree representation

This thesis will present algorithms to process an AST, several of these algorithms rely on breaking down the AST into several small "subtrees". These "subtrees" are sections of trees consisting of nodes with a common depth, or distance from the root. For example; in figure 2.4 all nodes with "$1 \leq depth \leq 2$" are selected in a subtree.

2.2.1 Classical notation

Usually, in computer science, trees are represented using edges between the parent and its children. The root node is usually at the top of the tree, with its children directly below it and edges between the parent and the children. Such as represented in figure 1.6. The downside of this notation is that the relationship of children at the same level in the tree is not modelled. For an AST however, the relationship of the children can be very important; "`a - b`" is usually not the same as "`b - a`".

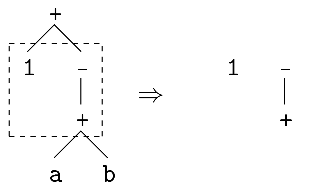


Figure 2.4: A subsection of $1 \leq depth \leq 2$

2.2.2 First-child, next-sibling notation

Because of the ambiguities with the classical tree notation, this thesis will also use the Filial-Heir notation [13]. This is also known as the "First-child, next-sibling" notation, or the child-sibling tree implementation. In this notation, the root is at the top of the tree and has an edge pointing downwards to its first child. The first child has an edge pointing right to its next sibling, which recurses until the last sibling. Each of the siblings can have an edge pointing downwards to their first child. Definition 2.2.1 defines the "FCNS-tree", for an example see figure 2.5. Note that in this notation it is also legal for the root node to have a sibling, this represents a forest where the roots have a sequential relationship.

Definition 2.2.1 *A "first-child, next-sibling tree", or "FCNS-tree", $\mathcal{F}$ is a finite acyclic directed connected graph denoted by a tuple (V, E). Where $V(\mathcal{F})$ is a finite nonempty set of labelled vertexes and $E(\mathcal{F})$ is a set of ordered pairs $\{u, v, l\}$ with $u, v \in V(\mathcal{F})$ where $\{u, v, l\}$ is a labelled edge in the graph from vertex u to vertex v with label $l \in \{$"fc","ns"$\}$. In this graph every vertex has a maximum of two outgoing edges: an edge labelled: "fc" (first-child) and an edge labelled: "ns" (next-sibling), every vertex has exactly one incoming edge except for the root vertex.*

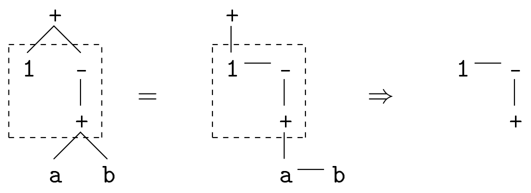


Figure 2.5: A subsection of $1 \leq depth \leq 2$, using the Filial-Heir notation

This notation has several advantages when representing ASTs. Primarily, the relationship between child nodes is explicit, which is important for evaluating the order of operands. A second advantage is that the Filial-Heir tree itself always is a binary tree, which simplifies iteration trough it somewhat.

2.2.3 Linear notation

Next to the graphical notations demonstrated in the previous sections, a linear notation is also used in this thesis. The benefit of a linear notation is that it is very compact and can therefore easily be used in formulae and code. The linear tree notation used is deliberately chosen to be an abstracted form of the linear notation used in ANTLR [8]. This notation is based on the tree operator "`^()`" introduced in section 1.1.3. The first element between the parentheses of the tree operator is considered to be a parent node in the tree, the other elements are considered to be the children of this parent. Definition 2.2.2 defines the linear notation, figure 2.6 presents an example.

Definition 2.2.2 *A vertex $v \in V(\mathcal{F})$ is denoted as: " $\hat{}(v\ C_1 ... C_n)$ " where $C_1...C_n$ represents the ordered set of children of v , which are recursively denoted in the same way. When the set of children of the vertex v is empty, the parentheses and $\hat{}$ sign may be discarded.*

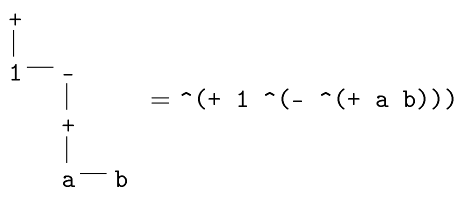


Figure 2.6: The linear notation of a tree

2.2.4 Properties of FCNS trees

A number of properties can be defined for FCNS-trees. These properties are used in the algorithms defined further on in this thesis. These properties are the root, the stem, levels and the depth.

Definition 2.2.3 *Any non-empty FCNS-tree always has exactly one root vertex. This is the vertex with an in-degree of 0, i.e. the vertex without incoming edges.*

This basically follows immediately from definition 2.2.1. Please note that the root node does not always follow the standard intuition for a root node. For example, it can have siblings. In figure 2.6, the topmost + is the root node.

Definition 2.2.4 *A set of stem vertexes can be determined for any non-empty FCNS-tree: the root vertex is a stem vertex. Any vertex that can be reached via an edge labelled "fc" from another stem vertex is also a stem vertex. All other vertexes are not stem vertexes.*

The stem vertexes form a path that starts at the root vertex, the edges along this path are drawn with double lines to clarify this. Vertexes along this path have a certain distance from the root vertex, this distance determines the level of the stem-subgraph. In figure 2.6, the topmost + and 1 are the stem vertexes. The set of stem vertices is also referred to as the stem of the FCNS tree. In the graphical representation of an FCNS tree, the edges between stem vertexes can be drawn with double lines.

Definition 2.2.5 *A non-empty FCNS-tree can be subdivided into one or more stem-subgraphs. These stem-subgraphs consist of one stem vertex and the entire subgraph connected to the "next-sibling" edge of that stem vertex.*

Definition 2.2.6 *The depth of an FCNS-tree is equal to the number of nodes in the stem.*

In figure 2.6, the depth of the FCNS-tree is 2. Please note that the definition of the depth of an FCNS tree is not equal to the definition of the depth of an ordinary tree. For example; in the ordinary tree representation in figure 2.5, the depth is 4.

Any ordinary forest of trees can be rewritten as an FCNS-tree and vice versa [13], as demonstrated in figure 2.7. The algorithms defined in this chapter strongly rely on the ordering of the child nodes inside a tree, therefore these FCNS-trees are used to represent them as they provide a more intuitive explanation of the algorithms.

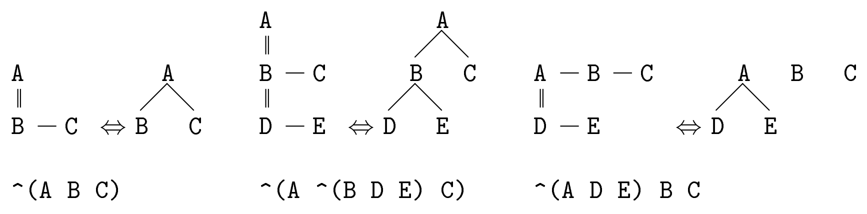


Figure 2.7: The differences between ordinary trees and FCNS-trees

Functions on FCNS trees

A number of functions can be defined for FCNS-trees. These functions are used in the algorithms defined further on in this document.

The root node of any FCNS-tree u can be selected by the function: $Root : \mathcal{F} \rightarrow \mathcal{V}$. The next sibling vertex of any vertex from any FCNS-tree can be selected by the function: $NextSibling : \mathcal{V} \rightarrow \mathcal{V} \cup \{\bot\}$. Similarly, the first child vertex can be selected by the function: $FirstChild : \mathcal{V} \rightarrow \mathcal{V} \cup \{\bot\}$. The previous sibling vertex of any vertex can be selected by the function: $PreviousSibling : \mathcal{V} \rightarrow \mathcal{V} \cup \{\bot\}$. Similarly, the parent vertex of any vertex can be selected by the function $Parent : \mathcal{V} \rightarrow \mathcal{V} \cup \{\bot\}$. The depth of any FCNS-tree can be retrieved with the function $Depth : \mathcal{F} \rightarrow \mathbb{N}$.

For some operations the last sibling is required, the last sibling is the vertex that can be found by following the edges labelled as next-sibling recursively from the current vertex until the last vertex has been found (i.e. without a next-sibling edge). This process is represented by the function $LastSibling : \mathcal{V} \rightarrow \mathcal{V}$.

Definition 2.2.7 *For any vertex v in a FCNS-tree, the set of all siblings can be determined: v is in the set of all siblings. Any vertex that is connected to a vertex that is in the set of all siblings via an edge labelled "ns", traversed either forwards or backwards, is also in the set of all siblings. All other vertexes are not in the set of all siblings. This process is represented by the function AllSiblings : $\mathcal{V} \rightarrow 2^{\mathcal{V}}$.*

2.3 Related work

Not many solutions for the problems identified in section 1.2 exist, most of them try to circumvent the problem by not using the tree walkers or taking a slightly different approach to the tree walking. Terence Parr explains in an article on his website [14] why he thinks tree walkers should be used by translators. In a response to this article, Andy Tripp wrote a white-paper that advocates hand-written code for walking an AST using the visitor pattern [15] rather than using ANTLR Tree Grammars [16]. This white-paper mainly focuses on the mixing of different languages and not so much on the maintenance issues, nevertheless a hand-written tree walker can potentially be easier to debug since hand-written code is usually better readable compared to generated code.

In 2008, Terence Parr introduced rewrite rules for ANTLR version 3 [17]. These rewrite rules allow the user to directly specify the desired shape of the AST, instead using the `!` and `^` operators, this makes the shape of the AST that is generated by the parser more transparent.

A more complete solution is ANTLRWorks [18] by Jean Bovet. ANTLRWorks is a development environment for ANTLR grammars which is able to help the user build and debug a grammar. It provides a grammar editor, various visualisation methods for grammars and an integrated debugger.

The solutions mentioned above do not resolve the underlying redundancy between the parser specification and the tree walker specification. Although, especially ANTLRWorks, they provide a method to deal with the consequences of this redundancy. The intention of this thesis is to directly target this redundancy.

Chapter 3

Generating the tree grammar

This thesis will present an algorithm for extracting one or more tree walkers from a parser specification. The input for this algorithm is the EBNF parser specification, similar to figure 1.4. The output will be a skeleton EBNF tree walker specification, similar to figure 1.5. The goal is to create a practically useful human-readable tree walker. This chapter will present the algorithm that handles the conversion from a parser specification to a tree walker specification. The algorithm is designed around grammars, ASTs and trees as introduced in chapter 2. For each of them a formal definition is given and examples are provided to understand them.

3.1 Rationale

The research questions in section 1.2 follow from a redundancy issue within the ANTLR parser and tree walker specifications. Usually, within models and specifications, redundancy is regarded as a bad thing as it implies more effort to implement and maintain. Furthermore, the behaviour of the tree building operators provided by ANTLR can become rather involved, especially when multiple of them are used in the same rule.

To address the redundancy issues while leaving the ANTLR paradigm intact, the solution is either to generate a tree walker specification or to validate a manually written tree walker specification against the parser specification. Although validation has some benefits over generation, in the sense that it gives the programmer more freedom in defining the tree walker; this would imply that an algorithm needs to be created that validates whether two algorithms generate the same language, which is impossible according to Rice's theorem [19]. When leaving the ANTLR paradigm, one can choose to use the visitor pattern [15] to accommodate the tree walkers. The downside of this is that the parser and the tree walkers will then use very different mechanisms.

Because of the considerations mentioned above, the choice has been made to automatically generate a tree walker specification from the parser specification.

3.2 Tree pattern algebra

As demonstrated in section 2.1.2, the resulting structure of the AST can be complex. An annotation anywhere in the rule of the context-free grammar can influence the entire section of the

AST generated by that rule. This section presents an algebra to help solve this problem. The basic idea behind this algebra is to use atomic building blocks to build up an AST. These building blocks represent atomic operations on the AST construction and are called “Tree Patterns”. Similar methods to use an algebra to have been presented in [20] and [21], the difference here is that the algebra is used to build a tree and not to match a tree.

3.2.1 Tree patterns

The FCNS-tree (as defined in section 2.2.2) forms the basis for tree patterns. Tree patterns are FCNS-trees labelled with terminals and/or non-terminals from the input context-free grammar and contain two connection points: a root connector and a child connector.

Definition 3.2.1 *A tree pattern $\mathcal{F}^+$ is an FCNS-tree such that: the last two stem vertexes are labelled with a root connector ($\bullet$) and a child connector ($\circ$) in that order. The other vertexes, if any, are labelled with terminals or non-terminals.*

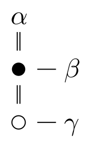
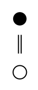


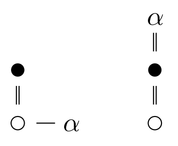
Figure 3.1: The tree pattern template

Figure 3.1 demonstrates the tree pattern template according to definition 3.2.1. Note that α , β and γ represent FCNS trees with zero or more vertexes labelled with terminals and/or non-terminals. The root connector of any tree pattern p can be selected by the function: *Rootconnector* : $\mathcal{F}^+ \rightarrow \mathcal{V}$, the child connector of any tree pattern p can be selected by the function: *Childconnector* : $\mathcal{F}^+ \rightarrow \mathcal{V}$.

Definition 3.2.2 *A minimal tree pattern is defined as the null pattern, also denoted by $\perp$. This is the smallest tree pattern conforming to definition 3.2.1:*



Definition 3.2.3 *The two elementary tree patterns are shown below:*



where α can be either one terminal or one non-terminal.

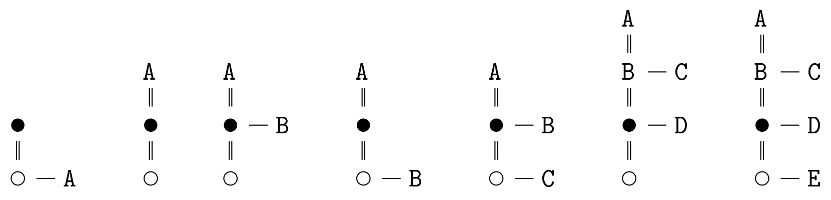


Figure 3.2: Some example tree patterns

The elements of the tree pattern algebra are defined as the infinite set of all possible tree patterns.

Figure 3.2 presents some example tree patterns, the linear representation of these patterns is (from left to right):

```
^(● ○ A)
^(A ^(● ○))
^(A ^(● ○) B)
^(A ^(● ○ B))
^(A ^(● ○ C) B)
^(A ^(B ^(● ○) D) C)
^(A ^(B ^(● ○ E) D) C)
```

3.2.2 Operations on tree patterns

The tree pattern algebra defines a limited set of operations. The most significant one is a binary operation called "Stacking", which combines two tree patterns. Furthermore, we define three unary operations called "Top", "Bottom" and "Collapse".

Stacking

Tree patterns are designed to be building blocks for an AST, these building blocks can be "stacked" on top of each other to form an AST. Tree pattern stacking is explicitly denoted by $p \Lsh q$, the root connector and child connector of the second pattern are used as connection points between the two patterns. The root of p becomes the first child of the parent of the root connector of q and the siblings of the root connector of q become the siblings of the root of p, the root connector of q itself is discarded. Furthermore, the siblings of the child connector of q will be appended to the set of children of the root of p, also the child connector of q is discarded. Definition 3.2.4 defines how stacking works.

Definition 3.2.4 *Stacking of tree patterns is denoted by* $\Lsh$: $\mathcal{F}^+ \times \mathcal{F}^+ \to \mathcal{F}^+$. *This operator stacks two patterns and results in a new tree pattern.*
$r = p \Lsh q$, *where:*

$$\begin{aligned}
V(r) = \;& V(p) \cup (V(q) - \{Rootconnector(q), Childconnector(q)\}) \\
E(r) = \;& E(p) \cup E(q) \cup \{ \\
& (LastSibling(Root(p)) \overset{ns}{\to} NextSibling(Rootconnector(q))), \\
& (LastSibling(FirstChild(Root(p))) \overset{ns}{\to} \\
& NextSibling(Childconnector(q))), \\
& (Parent(Rootconnector(q)) \overset{fc}{\to} Root(p))\}
\end{aligned}$$

Any dangling edges will be discarded. Note that the root connector and child connector of the second pattern are not part of the new pattern.

The stacking operator $\Lsh$ is also implicit; writing down two tree patterns next to each other, without parentheses separating them, implies stacking. The stacking of two patterns forms a new pattern, meaning $p \Lsh q$ results in a new pattern r. Furthermore, stacking is left-associative: $(p \Lsh q \Lsh r) \Leftrightarrow ((p \Lsh q) \Lsh r)$. From definition 3.2.4 it can be derived that stacking is not commutative: $(p \Lsh q) \neq (q \Lsh p)$. Figure 3.3 demonstrates three examples for tree pattern stacking.

The name of the root connector refers to the fact that in $p \Lsh q$, the root connector of q is connected to the root of p . The child connector of q connects to the last child of the root of p .

Top and Bottom

Patterns can be subdivided into a top and a bottom part. The root connector and child connector are the separators in this process; everything above and to the right of the root connector is considered to be the top part, everything below and to the right of the child connector is considered to be the bottom part. To ensure that the top and bottom parts are both valid tree patterns, the root connector and child connector are present in both parts. The functions $Top : \mathcal{F}^+ \rightarrow \mathcal{F}^+$ and $Bottom : \mathcal{F}^+ \rightarrow \mathcal{F}^+$, as defined in definitions 3.2.5 and 3.2.6, are used to split a pattern this way. Figure 3.6 demonstrates these functions.

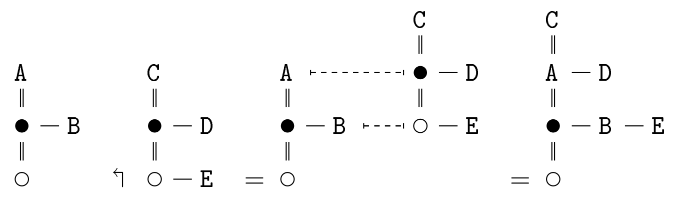
Definition 3.2.5 *The function* $Top : \mathcal{F}^+ \rightarrow \mathcal{F}^+$ *selects the top part of a tree pattern.*
$q = Top(p)$*, where:*
$V(q) = \quad V(p) - (AllSiblings(Childconnector(p)) - Childconnector(p))$
$E(q) = \quad E(p)$
Any dangling edges will be discarded.

Definition 3.2.6 *The function* $Bottom : \mathcal{F}^+ \rightarrow \mathcal{F}^+$ *selects the bottom part of a tree pattern.*
$q = Bottom(p)$*, where:*
$V(q) = \quad V(p) - ((V(Top(p)) - Rootconnector(p)) - Childconnector(p))$
$E(q) = \quad E(p)$
Any dangling edges will be discarded.

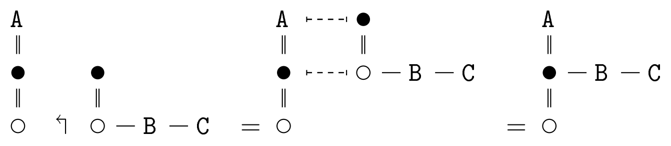
Collapsing

The root connector and child connector can be removed from the tree pattern by collapsing it. Collapsing appends all siblings of the root connector to the set of siblings of the child connector and makes the next sibling of the child connector the first child of the parent of the root connector. The function $Collapse : \mathcal{F}^+ \rightarrow \mathcal{F}$, as defined in definition 3.2.7, is designed to do this, figure 3.7 demonstrates this.

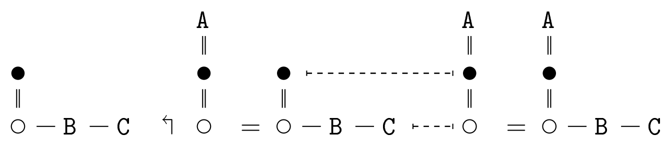
Definition 3.2.7 *The function* $Collapse : \mathcal{F}^+ \rightarrow \mathcal{F}$ *collapses the root connector and child connector from the tree pattern.*
$v = Collapse(p)$*, where:*



Or linear: ^(A ^(● ○) B) ↰ ^(C ^(● ○ E) D) = ^(C ^(A ^(● ○) B E) D)



Or linear: ^(A ^(● ○)) ↰ ^(● ○ B C) = ^(A ^(● ○) B C)



Or linear: ^(● ○ B C) ↰ ^(A ^(● ○)) = ^(A ^(● ○ B C))

Figure 3.3: Examples for the stacking operator ↰: $\mathcal{F}^+ \to \mathcal{F}^+$

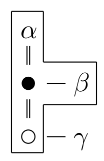


Figure 3.4: Template for the function *Top* : $\mathcal{F}^+ \to \mathcal{F}^+$

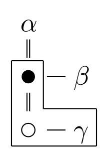


Figure 3.5: Template for the function *Bottom* : $\mathcal{F}^+ \to \mathcal{F}^+$

$$Top\left(\begin{array}{l} \texttt{G} \\ \| \\ \texttt{H} - \texttt{E} \\ \| \\ \bullet - \texttt{F} \\ \| \\ \circ - \texttt{D} \end{array}\right) = \begin{array}{l} \texttt{G} \\ \| \\ \texttt{H} - \texttt{E} \\ \| \\ \bullet - \texttt{F} \\ \| \\ \circ \end{array}$$

Or linear: *Top*(^(G ^(H ^(● ○ D) F) E)) = ^(G ^(H ^(● ○) F) E)

$$Bottom\left(\begin{array}{l} \texttt{B} \\ \| \\ \bullet - \texttt{C} \\ \| \\ \circ - \texttt{A} \end{array}\right) = \begin{array}{l} \bullet \\ \| \\ \circ - \texttt{A} \end{array}$$

Or linear: *Bottom*(^(B ^(● ○ A) C)) = ^(● ○ A)

Figure 3.6: Examples for the functions $Top : \mathcal{F}^+ \rightarrow \mathcal{F}^+$ and $Bottom : \mathcal{F}^+ \rightarrow \mathcal{F}^+$

$$\begin{array}{ll} V(v) = & V(p) - Rootconnector(p) - Childconnector(p) \\ E(v) = & E(p) \cup \{ \\ & (Parent(Rootconnector(p)) \overset{fc}{\rightarrow} NextSibling(Childconnector(p))), \\ & (LastSibling(Childconnector(p)) \overset{ns}{\rightarrow} NextSibling(Rootconnector(p)))\} \end{array}$$

Any dangling edges will be discarded.

$$Collapse\left(\begin{array}{l} \texttt{G} \\ \| \\ \texttt{H} - \texttt{E} \\ \| \\ \bullet - \texttt{F} \\ \| \\ \circ - \texttt{D} - \texttt{I} \end{array}\right) = \begin{array}{l} \texttt{G} \\ \| \\ \texttt{H} - \texttt{E} \\ \| \\ \texttt{D} - \texttt{I} - \texttt{F} \end{array}$$

Or linear: *Collapse*(^(G ^(H ^(● ○ D I) F) E)) = ^(G ^(H D I F) E)

Figure 3.7: Example for the function $Collapse : \mathcal{F}^+ \rightarrow \mathcal{F}$

3.2.3 Master and slave patterns

When taking a closer look at the stacking operator, two kinds of patterns can be distinguished. Considering the stacking $p \Lsh q = r$; if q has a root other than the root connector, q will provide the root of r and is designated a master pattern. One exception is made for the null pattern, as it is a master pattern even though the root connector is the root of it. Definition 3.2.8 defines the distinction between master and slave patterns.

Definition 3.2.8 *A slave tree pattern is a tree pattern where (i) the root connector is the root and (ii) the root connector and/or the child connector has siblings. Otherwise, it is a master tree pattern.*

Consider the examples in figure 3.3; in the first and third examples, q is a master pattern and provides the new root in the stacking. For the second example, q is a slave and does not provide the root. Figure 3.8 demonstrates the elementary tree patterns.

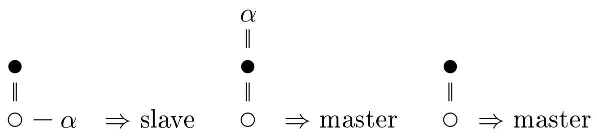


Figure 3.8: The elementary tree patterns

Parent-child relations in stacking

When stacking two patterns, one of the patterns usually dominates the other pattern in the sense that it provides the topmost or leftmost nodes in the resulting pattern. Which pattern will be the dominant pattern depends on the order in which they are stacked and whether the patterns are master or slave . The dominant pattern is referred to as the "parent" pattern, the other pattern is referred to as the "child" pattern.

Definition 3.2.9 *For $p \Lsh q$, The child pattern can be determined as follows: If q is master; p will become the child of q, otherwise q will become the child of p.*

For example, consider figure 3.3. The patterns `^(A ^(● ○) B)` , `^(● ○ B C)` and `^(● ○ B C)` become the child patterns in these operations.

3.3 The context-free tree grammar

The output for the algorithm is a context-free tree grammar. Instead of parsing sentences, a context-free tree grammar parses trees generated by an annotated context-free grammar. Definition 3.3.1 defines a context-free tree grammar.

Definition 3.3.1 *A context-free tree grammar T is denoted by a 4-tuple (N, F, R, S), where N is a finite nonempty set of non-terminals, F a finite nonempty set of FCNS trees and $S \in N$ the start symbol. The alphabet of T is denoted by $N \cup F$. R is a finite set of production rules: for each $r \in R : n \rightarrow \alpha$, where $n \in N$. Here α can be defined using EBNF:*
  α `= A | (`α`)op |` $\alpha_1 + \alpha_2$
  `op = * | + | ? |`
with $A \in F$.

For example, consider the two FCNS trees from figure 3.9. The context-free tree grammar from figure 3.10 is able to parse both trees. When this grammar is written using the linear notation described in definition 2.2.2, it becomes much more compact as demonstrated in figure 3.11. The context-free tree grammar in linear notation is very similar to a tree walker specification of ANTLR.

3.4 Building the AST

When parsing a sentence with an annotated context-free grammar, an AST is built up. The exact shape of the AST depends on the placement of the double-caret annotations within the production rules. To build the corresponding AST, one has to parse the sentence from left to right and obey the following rules:

- Every unannotated terminal becomes either a child of the last terminal annotated with a double-caret or, if there is no such terminal, a sibling of the previous unannotated terminal.
- Every terminal annotated with a double-caret becomes the root of the tree as parsed until there.
- Every terminal annotated with a exclamation mark is not taken into the tree.

For example, consider the following annotated context-free grammar rule: "`start: A B C^^ D E^^ F G!;`". Parsing a sentence using this rule, from left to right; (i) the first terminal is `A`, which becomes the first node in the tree. (ii) `B` becomes a sibling of `A`. (iii) `C` becomes the parent of both `A` and `B`. (iv) `D` becomes the child of `C` and thus siblings of `A` and `B`. (v) `E` becomes the parent of `C`. (vi) `F` becomes the child of `E` and thus a sibling of `C`. Finally, (vii) `G` is not taken into the tree. Resulting in the tree depicted in figure 3.12.

3.4.1 Rewriting production rules

Now assume that instead of the rule "`A B C^^ D E^^ F G!`" the rule of figure 3.13 is used, meaning that the resulting tree consists of $n \cdot$`(A B C^^)`$+m\cdot$`(D E^^)`$+$`(F G!)`, with $n, m \in \mathbb{N}$. This section will demonstrate how to use tree patterns to determine the resulting AST for any finite value of n and m. First, definition 3.4.1 will show how to write an annotated context-free grammar using tree patterns instead of annotations.

Definition 3.4.1 *Any annotated context-free grammar, as defined in definition 2.1.2, can be translated using elementary tree patterns. This can be done by translating each of the terminals or non-terminals on the right-hand side of each production rule as tree patterns based on their annotation:*

```
A
|
B — C       A
    |       |
    D — E   B — D — E
```

Figure 3.9: Two example trees.

```
start
      A
      |
   :  B — tail
   ;

tail
      C
      |
   :  D — E
   |  D — E
   ;
```

Figure 3.10: A context-free tree grammar that is able to parse the trees from figure 3.9.

```
start
  : ^(A B tail)
  ;

tail
  : ^(C D E)
  | D E
  ;
```

Figure 3.11: The context-free tree grammar from figure 3.10 using the linear notation

```
E
‖
C — F
‖
A — B — D
```

Figure 3.12: The AST for the rule "`A B C^^ D E^^ F G!`", given the sentence "`A B C D E F G`"

```
rule
  : (A B C^^)* (D E^^)* (F G!)
  ;
```

Figure 3.13: A production rule.

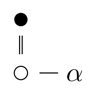
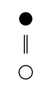
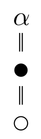
Terminal/Non-Terminal	*Pattern*	*Linear*
α		^(● ○ α)
α!		^(● ○)
α^^		^(α ^(● ○))

Figure 3.14 shows the application of definition 3.4.1 on the production rule from figure 3.13. Considering figure 3.14 and the stacking operator from section 3.2.2; one can immediately recognise the rules for building an AST presented earlier in this section. For example, `B` becomes a sibling of `A` and `C` becomes the parent of `A` and `B`.

```
rule
                      C               E
                      ‖               ‖
      ●       ●       ●       ●       ●       ●       ●
      ‖       ‖       ‖       ‖       ‖       ‖       ‖
  : ( ○ — A   ○ — B   ○  )* ( ○ — D   ○  )* ( ○ — F   ○ )
  ;
```

Figure 3.14: The production rule from figure 3.13 written down using tree patterns

3.4.2 Single carets

The algorithm only considers double caret operators in a grammar. The reason for this is that the behaviour of single caret operators can be modelled using double caret operators. The single caret operator behaves exactly like the double caret operator, with the exception that its effect stays limited to the nearest set of parentheses. Furthermore, single caret and double caret operators are not allowed to be used together in one rule. Therefore, moving all terminals and non-terminals between these parentheses to a new rule and replacing the single carets by double carets will generate the same AST.

3.4.3 Normal-form

Writing down an annotated context-free grammar using definition 3.4.1 can cause two or more patterns to follow each other without being separated by parentheses. As explained in section 3.2.2; this means the stacking operator is implicitly present and can be applied to them. For any further operations on the annotated context-free grammar it may be desirable to be as compact as possible, therefore we introduce the annotated context-free grammar written down using tree patterns in normal-form. This means that all stacking operators between consecutive tree patterns have been applied and the stacking operator only remains between EBNF sections. Definition 3.4.2 defines this normal-form, figure 3.15 shows the production rule from figure 3.14 in normal-form.

Definition 3.4.2 *A production rule in an annotated context-free grammar written down using tree patterns is in normal-form iff the right-hand side of the rule only contains a stacking operator between EBNF sections. If all rules of an annotated context-free grammar are in normal-form, the grammar itself is said to be in normal-form.*

```
rule
       C                E
       ‖                ‖
       ●                ●             ●
       ‖                ‖             ‖
   : ( ○ — A — B  )* (  ○ — D  )* (   ○ — F  )
   ;
```

Figure 3.15: The production rule from figure 3.14 in normal-form

Figure 3.16, depicts the production rule from figure 3.15 as a formula. In this formula the operator $\cdot : \mathbb{N}_0 \times \mathcal{F}^+ \rightarrow \mathcal{F}^+$ is defined as stacking the pattern repetitively on top of itself, where the number of repetitions is equal to the natural number minus one and $\perp$ is the result if the natural number equals zero. When using the value 1 for n and m in this formula; the resulting tree is equal to the tree from figure 3.12. Please note that the collapse is required to remove the root connector and child connector from the final tree pattern. Figure 3.17 shows several trees for different values of n and m.

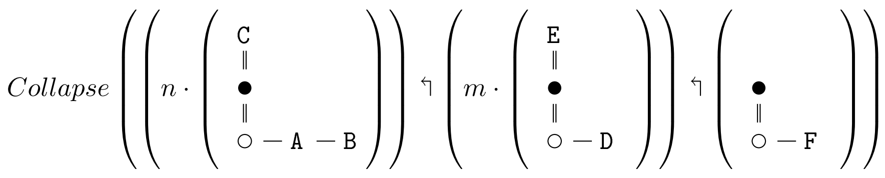


Figure 3.16: A formula for all possible ASTs of figure 3.15.

3.5 Modelling an annotated context-free grammar

Section 3.4 defined how to use tree patterns to formalise the generation of an AST. The tree patterns can be seen as atomic building blocks that can be connected in any order allowed by the context free grammar. This section will present an algorithm that extracts the order in which the tree patterns can potentially be connected to each other. First, the potential tree patterns that start the sequence are modelled, this is later used for the start rule in the context-free tree grammar. Next, for each pattern it is determined which patterns can potentially follow it, which is used to generate the other rules in the context-free tree grammar. The starting point of this procedure will be the annotated context-free grammar written down using tree patterns in normal-form.

```
values for n and m    Produced tree
  n = 0, m = 0              F
                         C
                         ‖
  n = 1, m = 0           A — B — F
                         E
                         ‖
                         C — F
                         ‖
  n = 1, m = 1           A — B — D
                       E
                       ‖
                       C — F
                       ‖
                       C — D
                       ‖
  n = 2, m = 1         A — B — A — B
```

Figure 3.17: The different trees generated by the formula of figure 3.16.

3.5.1 Overview

This part of the algorithm extracts the sequential order in which the tree patterns of a production rule can be used to form an AST. For example, consider the production rule from figures 3.15 and 3.16, where the patterns are labelled p, q and r from left to right. Figure 3.17 shows that patterns p, q and r can all potentially provide the root of the AST. Furthermore, pattern p can have the patterns p and r as a child and pattern q can have the patterns p, q and r as a child.

These possible constructions have to be allowed by the corresponding context-free tree grammar. For example, an abstract representation of the rule representing the start symbol would be "$start$: $p|q|r$;". For the rules representing p, q and r they would be "p : $p|r$;", "q : $p|q|r$;" and "r :;".

The other paragraphs of this section explain in detail how this structure and information is extracted, section 3.6 explains how this information is combined to generate the actual context-free tree grammar.

3.5.2 Patterns that provide the root

A context-free tree grammar, as defined in section 3.3, parses a tree from top to bottom. However, section 3.4 demonstrated that, if it is annotated with a double-caret, the root node may be the last node that has been parsed by the parser. To build the context-free tree grammar from tree patterns, one has to start with the tree pattern that provided the root of the AST section produced by the production rule. To determine which patterns will provide the root of the AST section, a minimal deterministic automaton is used. Figure 3.18 shows an automaton for the production rule from figure 3.15; p, q and r represent respectively the leftmost, centre and rightmost patterns from figure 3.15.

The accepting state in the automaton from figure 3.18 only has incoming edges labelled with r, meaning that the last pattern to be parsed is always the rightmost pattern from figure 3.15. However, this does not mean that pattern r will always provide the root. Pattern r is a slave

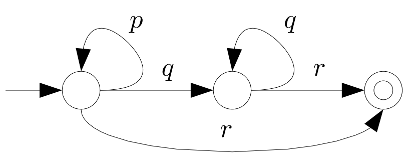


Figure 3.18: An automaton for the grammar from figure 3.15.

pattern; if it is stacked on another pattern, that pattern will provide the root of the resulting pattern. This means that the pattern that is parsed before r is relevant. The automaton has a path where r is the only pattern to be parsed and hence provides the root, there are also paths where either pattern p or pattern q precedes r; as both p and q are master patterns, they will provide the root of the AST section. For this grammar, patterns p, q and r can provide the root node of the AST section, hence they are in the set of root patterns RT. Definition 3.5.1 defines this set.

Definition 3.5.1 *For each production rule $\rho \in R(G)$ of an annotated context-free grammar G we define the set of root patterns RT. A pattern p is member of RT if an AST section P can be produced by ρ where the root node of P is provided by p, as defined by definition 2.2.3. The null-pattern $\perp$ is included in the set RT if an AST section P can be produced by ρ where P is empty.*

To obtain the complete set of root patterns RT one has to consider every path through the automaton for ρ. For each path with master patterns, the last master pattern in the path is member of this set. For each path without master patterns, the first pattern is member of this set. Finally, if there exists a path without patterns, the null-pattern $\perp$ is included in the set (i.e. the initial state is also an accepting state).

Figure 3.19 presents pseudo-code for obtaining the set RT given a minimal deterministic automaton for the rule. The algorithm traverses the automaton backwards, starting at the accepting states and working towards the initial state, using the recursive function `obtainIn` at line 11. To prevent infinite loops, the states are marked once visited so they are not visited again using the code at lines 2, 13 and 27. The loop on line 3 calls the function `obtainIn` for each accepting state in the specified automaton A, it also adds $\perp$ to the set RT if an accepting state is also an initial state. The function `obtainIn` analyses all incoming edges of a state. If one of those edges is labelled with a master pattern; it is added to the set RT. If one of those edges is labelled with a slave pattern; the function `obtainIn` is called recursively on the source state of the edge, the slave pattern is added to the set if the source state of the edge is an initial state.

3.5.3 Iterating through the rule

As the context-free tree grammar parses from top to bottom, it is important to know which patterns can exist below each other. The nodes below a pattern are provided by patterns that have become children of this pattern, as defined in definition 3.2.9. Again, the minimal deterministic automaton can be used to derive this information. For example, consider the automaton from figure 3.18. The left state in this automaton has an outgoing edge labelled with p and an

```
A  : The automaton to obtain the set from
RT: The resulting set $RT$
$RT := \emptyset$;
Set all states $\in A$ to not-visited;
foreach $state \in A$ do
    if state is accepting state then
        $RT := RT \cup$ obtainIn($state$);
        if state is an initial state then
            $RT := RT \cup \{\perp\}$;
        end
    end
end
function obtainIn($S$) : $RT$  begin
    $RT := \emptyset$;
    if $S$ has not been visited then
        foreach $edge \in$ incoming edges of $S$ do
            $pattern :=$ label of $edge$;
            if pattern is a master pattern then
                $RT := RT \cup \{pattern\}$;
            end
            else
                $state :=$ source state of $edge$;
                $RT := RT \cup$ obtainIn($state$);
                if state is an initial state then
                    $RT := RT \cup \{pattern\}$;
                end
            end
        end
        Set $S$ to visited;
    end
end
```

Figure 3.19: Pseudo-code for obtaining the set RT from an automaton.

```
A     : The automaton to obtain the set from
P     : The pattern to obtain the set for
$CH^+$: The resulting set $CH^+$
$CH^+ := \emptyset$;
Set all states $\in A$ to not-visited;
if $P$ is a master pattern then
    foreach edge $\in$ edges of $A$ with label $P$ do
        $CH^+ := CH^+ \cup$ obtainIn(source state of edge);
    end
end
function obtainIn($S$) : $CH^+$ begin
    $CH^+ := \emptyset$;
    if $S$ has not been visited then
        foreach edge $\in$ incoming edges of $S$ do
            pattern := label of edge;
            if pattern is a master pattern then
                $CH^+ := CH^+ \cup \{pattern\}$;
            end
            else
                $CH^+ := CH^+ \cup$ obtainIn(source state of edge);
            end
        end
        if $S$ is an initial state then
            $CH^+ := CH^+ \cup \{\perp\}$;
        end
        Set $S$ to visited;
    end
end
```

Figure 3.20: Pseudo-code for obtaining the set $CH^+(p)$ from an automaton for a given pattern.

```
**A**      : The automaton to obtain the set from
**P**      : The pattern to obtain the set for
$CH^-$: The resulting set $CH^-$

$CH^- := \emptyset$;
foreach edge ∈ edges of A with label P do
    state := destination state of edge;
    foreach edge ∈ outgoing edges of state do
        pattern := label of edge;
        if pattern is a slave pattern then
            $CH^- := CH^- \cup \{pattern\}$;
        end
    end
    if state is an accepting state then
        $CH^- := CH^- \cup \{\perp\}$;
    end
end
if P is a master pattern then
    Set all states ∈ A to not-visited;
    foreach edge ∈ edges of A with label P do
        $CH^- := CH^- \cup$ obtainIn(source state of edge, P);
    end
end

function obtainIn(S, P) : $CH^-$ begin
    $CH^- := \emptyset$;
    if S has not been visited then
        foreach edge ∈ incoming edges of S do
            pattern := label of edge;
            if pattern is a slave pattern then
                $CH^- := CH^- \cup$ obtainIn(source state of edge, pattern);
            end
        end
        if S is an initial state then
            $CH^- := P$;
        end
        Set S to visited;
    end
end
```

Figure 3.21: Pseudo-code for obtaining the set $CH^-(p)$ from an automaton for a given pattern.

incoming edge labelled with p, meaning that p can be stacked on itself and p can become a child of p. Furthermore, the left state in this automaton has an outgoing edge labelled with r, meaning that r can be stacked on p and, as r is a slave pattern, r can become a child of p.

Now, for each pattern two sets can be defined; the set of master patterns that can become children of the pattern and the set of slave patterns that can become children of the pattern, respectively CH^+ and CH^-, defined in definitions 3.5.2 and 3.5.3.

Definition 3.5.2 *For each master pattern p within a rule $\rho \in R(G)$ of an annotated context-free grammar G we define the set of master child patterns $CH^+(p)$. A master pattern p' is member of this set if an AST section can be produced by ρ where p' is a child of p, as defined in definition 3.2.9. If an AST section can be produced where there is no pattern p', the null-pattern $\perp$ is included in the set $CH^+(p)$.*

For the production rule from figure 3.15, the sets are $CH^+(p) = \{p, \perp\}$ and $CH^+(q) = \{p, q, \perp\}$. Figure 3.20 presents pseudo-code for obtaining the set CH^+ for an active pattern given a minimal deterministic automaton for the rule. The algorithm traverses the automaton backwards, starting at the source states of edges labelled with the master pattern, using the recursive function `obtainIn` at line 8. To prevent infinite loops, the states are marked once visited so they are not visited again using the code at lines 2, 10 and 23. The loop on line 3 calls the function `obtainIn` for each source state of edges labelled with the master pattern in the specified automaton A. The function `obtainIn` analyses all incoming edges of a state. If one of those edges is labelled with a master pattern; it is added to the set CH^+. If one of those edges is labelled with a slave pattern; the function `obtainIn` is called recursively on the source state of the edge. On line 20 $\perp$ is added to the set CH^+ if the state provided to the function `obtainIn` is the initial state.

Definition 3.5.3 *For each pattern p within a rule $\rho \in R(G)$ of an annotated context-free grammar G we define the set of slave child patterns $CH^-(p)$. A slave pattern p' is member of this set if an AST section can be produced by ρ where p' is a child of p, as defined in definition 3.2.9. If an AST section can be produced where there is no pattern p', the null-pattern $\perp$ is included in the set $CH^-(p)$.*

For the production rule from figure 3.15, the sets are $CH^-(p) = \{r, \perp\}$, $CH^-(q) = \{r, \perp\}$ and $CH^-(r) = \{\perp\}$. Figure 3.21 presents pseudo-code for obtaining the set CH^- for a pattern given a minimal deterministic automaton for the rule. In the loop at line 2, the algorithm looks one step forward in the automaton for slave child patterns. In case the pattern under investigation is a master pattern; the algorithm also traverses the automaton backwards, starting at the source states of edges labelled with the master pattern, using the recursive function `obtainIn` at line 20. To prevent infinite loops, the states are marked once visited so they are not visited again using the code at lines 15, 22 and 32.

3.6 Constructing the context-free tree grammar

Section 3.5 defined how to extract the potential roots of an AST section into the set RT and to extract the potential paths through the automaton in the sets CH^+ and CH^-. This information can now be used to generate the context-free tree grammar.

3.6.1 Overview

This section will generate the context-free tree grammar using the information modelled in the sets RT, CH^+ and CH^-. This information has been summarised in figure 3.23 for the examples of the previous section.

```
rule
     C              E
     ‖              ‖
     ●              ●          ●
     ‖              ‖          ‖
  : ( ○ — A — B  )* ( ○ — D  )* ( ○ — F  )
  ;
```

Figure 3.22: An example production rule.

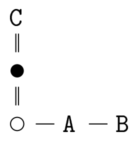
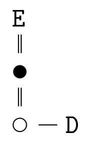
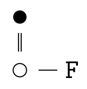
Designation	Pattern	Type	RT	CH^+	CH^-
p		master	$\in RT$	$\{p, \perp\}$	$\{r, \perp\}$
q		master	$\in RT$	$\{p, q, \perp\}$	$\{r, \perp\}$
r		slave	$\in RT$	*Not Applicable*	$\{\perp\}$

Figure 3.23: The sets RT, CH^+ and CH^- for the production rule from figure 3.22.

Figures 3.10 and 3.11 illustrated that a context-free tree grammar traverses the stem of the AST from top to bottom. For example, consider figure 3.24; in this figure all nodes are marked with subscripts 1 - 4, grouping them by the original instance of the pattern that provided these nodes. Figure 3.25 presents a context-free tree grammar that is able to parse this AST.

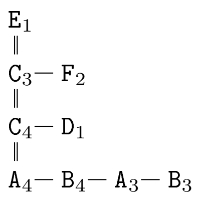


Figure 3.24: An AST generated by the production rule of figure 3.22.

Note that master tree patterns can influence at most three lines in the AST; the line where the root of the tree pattern is placed, the line where the siblings of the root connector are placed

```
start
  : ^(E ch+_q F)
  ;

ch+_q
  : ^(C ch+_p D)
  ;

ch+_p
  : ^(C ch+_p A B)
  | A B
  ;
```

Figure 3.25: A tree grammar that can parse the tree from figure 3.24.

and the line where the siblings of the child connector are placed. That means that, ignoring slave tree patterns for now, every horizontal line except for the bottom line in an AST takes the following form: A root node of a tree pattern, followed by zero or more siblings of the root connector of the pattern above it, followed by zero or more siblings of the child connector of the pattern above that pattern.

The context-free tree grammar of figure 3.25 is designed such that every rule parses a root node of a tree pattern and all nodes that are a direct child of it. The first child of the root node, i.e. next stem node, is always referenced with a non-terminal to a rule that models the next line in the AST. This non-terminal is followed by any siblings of the root connector of the tree pattern. The siblings of the child connector of the pattern are placed on the next line in the AST and are therefore taken to the rule specified by the non-terminal that models the first child of the root node. Therefore the rule `ch+`$_q$ also contains the siblings of the child connector of q. The rule `ch+`$_p$ has one alternative `A B` which is the result of the very first instance of pattern p, modelled as `A`$_4$, `B`$_4$ and `C`$_4$ in figure 3.24, has no master child patterns and therefore the siblings of the child connector of the pattern are not appended to the siblings of the root connector of another parent, but remain as direct children of their own parent.

Extending the context-free tree grammar of figure 3.25 to a context-free tree grammar that can parse any AST produced by the production rule from figure 3.22 can be realised by extending the rules such that they model all entries in the sets RT, CH^+ and CH^-. Figure 3.26 presents this grammar; the start rule has been extended with the patterns p and r and pattern q has been added as a potential child of q in the rule `ch+`$_q$.

The rest of this section will formalise the process described in this overview using the tree pattern algebra presented in section 3.2. Also the process will be extended to include of slave tree patterns.

3.6.2 Start symbol

The context-free tree grammar traverses the AST section from top to bottom, the rule representing the start symbol is therefore generated from the set RT. The set RT contains all patterns that can potentially become the root of the AST section, each pattern in this set will provide an alternative in this rule. The tree pattern formulæ from definition 3.6.1 and 3.6.2 abstract the alternatives for master patterns and slave patterns respectively. Figure 3.27 demonstrates these

```
start
  : ^(C ch+_p F)
  | ^(E ch+_q F)
  | F
  ;

ch+_q
  : ^(C ch+_p D)
  | ^(E ch+_q D)
  | D
  ;

ch+_p
  : ^(C ch+_p A B)
  | A B
  ;
```

Figure 3.26: A tree grammar that can parse any tree from the production rule from figure 3.22.

formulæ for the patterns from figure 3.23.

Definition 3.6.1 *For each master pattern p_i in the set RT for a rule ρ of an annotated context-free grammar G, an alternative a_{p_i} exists in the rule ρ' in its counterpart context-free tree grammar G', where:*

$$a_{p_i} = Collapse\left(\begin{array}{l}\bullet\\ \| \\ \circ - \texttt{ch+}_{p_i}\end{array} \urcorner\, Top(p_i) \urcorner \begin{array}{l}\bullet\\ \| \\ \circ - \texttt{ch-}_{p_i}\end{array}\right)$$

Definition 3.6.2 *For each slave pattern p_i in the set RT for a rule ρ of an annotated context-free grammar G, an alternative a_{p_i} exists in the rule ρ' in its counterpart context-free tree grammar G' where:*

$$a_{p_i} = Collapse\left(p_i \urcorner \begin{array}{l}\bullet\\ \| \\ \circ - \texttt{ch-}_{p_i}\end{array}\right)$$

3.6.3 Pattern rules

The rule representing the start symbol models the set RT, the sets CH^+ and CH^- for each pattern are modelled by pattern rules. Two rules exist for each pattern $p_i \in \rho$, designated by the non-terminals $\texttt{ch+}_{p_i}$ and $\texttt{ch-}_{p_i}$. The tree pattern formulæ from definition 3.6.3 and 3.6.4 abstract the alternatives for master patterns and slave patterns respectively. Figure 3.28 demonstrates these formulæ for the patterns from figure 3.23.

Definition 3.6.3 *For each master pattern p_i in a rule ρ of an annotated context-free grammar G, a rule $\texttt{ch+}_{p_i}$ exists in its counterpart context-free tree grammar G'. For each pattern q_j in the set $CH^+(p_i)$, an alternative a_{q_j} exists in the rule $\texttt{ch+}_{p_i}$, where:*

$$a_p = \mathit{Collapse}\left(\begin{array}{l}\bullet\\ \|\\ \circ - \texttt{ch+}p\end{array} \Lsh \mathit{Top}\left(\begin{array}{l}\texttt{C}\\ \|\\ \bullet\\ \|\\ \circ - \texttt{A} - \texttt{B}\end{array}\right) \Lsh \begin{array}{l}\bullet\\ \|\\ \circ - \texttt{ch-}p\end{array}\right)$$

$$= \mathit{Collapse}\left(\begin{array}{l}\bullet\\ \|\\ \circ - \texttt{ch+}p\end{array} \Lsh \begin{array}{c}\texttt{C}\\ \|\\ \bullet\\ \|\\ \circ\end{array} \Lsh \begin{array}{l}\bullet\\ \|\\ \circ - \texttt{ch-}p\end{array}\right)$$

$$= \mathit{Collapse}\left(\begin{array}{l}\texttt{C}\\ \|\\ \bullet - \texttt{ch-}p\\ \|\\ \circ - \texttt{ch+}p\end{array}\right)$$

$= \hat{}(\texttt{C ch+}p\ \texttt{ch-}p)$ *(linear notation)*

$$a_q = \mathit{Collapse}\left(\begin{array}{l}\bullet\\ \|\\ \circ - \texttt{ch+}q\end{array} \Lsh \mathit{Top}\left(\begin{array}{l}\texttt{E}\\ \|\\ \bullet\\ \|\\ \circ - \texttt{D}\end{array}\right) \Lsh \begin{array}{l}\bullet\\ \|\\ \circ - \texttt{ch-}q\end{array}\right)$$

$$= \mathit{Collapse}\left(\begin{array}{l}\bullet\\ \|\\ \circ - \texttt{ch+}q\end{array} \Lsh \begin{array}{c}\texttt{E}\\ \|\\ \bullet\\ \|\\ \circ\end{array} \Lsh \begin{array}{l}\bullet\\ \|\\ \circ - \texttt{ch-}q\end{array}\right)$$

$$= \mathit{Collapse}\left(\begin{array}{l}\texttt{E}\\ \|\\ \bullet - \texttt{ch-}q\\ \|\\ \circ - \texttt{ch+}q\end{array}\right)$$

$= \hat{}(\texttt{E ch+}q\ \texttt{ch-}q)$ *(linear notation)*

$$a_r = \mathit{Collapse}\left(\begin{array}{l}\bullet\\ \|\\ \circ - \texttt{F}\end{array} \Lsh \begin{array}{l}\bullet\\ \|\\ \circ - \texttt{ch-}r\end{array}\right)$$

$= \texttt{F ch-}r$ *(linear notation)*

Figure 3.27: Definitions 3.6.1 and 3.6.2 applied to all patterns from figure 3.23.

$$a_{q_j} = \mathit{Collapse}\left(\begin{array}{l}\bullet\\ \Vert\\ \circ - \texttt{ch+}_{q_j}\end{array} \Lsh \left(\mathit{Top}(q_j) \Lsh \begin{array}{l}\bullet\\ \Vert\\ \circ - \texttt{ch-}_{q_j}\end{array} \Lsh \mathit{Bottom}(p_i)\right)\right)$$

Any reference to $\texttt{ch+}_{\perp}$ *or* $\texttt{ch-}_{\perp}$ *will be discarded.*

Definition 3.6.4 *For each slave pattern p_i in a rule ρ of an annotated context-free grammar G, a rule $\texttt{ch-}_{p_i}$ exists in its counterpart context-free tree grammar G'. For each pattern q_j in the set $CH^-(p_i)$, an alternative a_{q_j} exists in the rule $\texttt{ch-}_{p_i}$, where:*

$$a_{q_j} = \mathit{Collapse}\left(q_j \Lsh \begin{array}{l}\bullet\\ \Vert\\ \circ - \texttt{ch-}_{q_j}\end{array}\right)$$

Any reference to $\texttt{ch-}_{\perp}$ *will be discarded.*

Figure 3.29 demonstrates the complete grammar generated using definitions 3.6.1 to 3.6.4 for the production rule from figure 3.22.

$$app = Collapse\left(\begin{array}{l}\bullet\\ \|\\ \circ - \texttt{ch+}_p\end{array} \Lsh \left(\begin{array}{l}\texttt{C}\\ \|\\ \bullet\\ \|\\ \circ\end{array} \Lsh \begin{array}{l}\bullet\\ \|\\ \circ - \texttt{ch-}_p\end{array} \Lsh \begin{array}{l}\bullet\\ \|\\ \circ - \texttt{A} - \texttt{B}\end{array}\right)\right)$$

$= \hat{}(\texttt{C}\ \texttt{ch+}_p\ \texttt{ch-}_p\ \texttt{A}\ \texttt{B})$ *(linear notation)*

$$ap\bot = Collapse\left(\begin{array}{l}\bullet\\ \|\\ \circ - \texttt{ch+}_\bot\end{array} \Lsh \left(\begin{array}{l}\bullet\\ \|\\ \circ\end{array} \Lsh \begin{array}{l}\bullet\\ \|\\ \circ - \texttt{ch-}_\bot\end{array} \Lsh \begin{array}{l}\bullet\\ \|\\ \circ - \texttt{A} - \texttt{B}\end{array}\right)\right)$$

(Note that any reference to $\texttt{ch+}_\bot$ *and* $\texttt{ch-}_\bot$ *is to be discarded.)*
$= \texttt{A}\ \texttt{B}$ *(linear notation)*

$$aqp = Collapse\left(\begin{array}{l}\bullet\\ \|\\ \circ - \texttt{ch+}_p\end{array} \Lsh \left(\begin{array}{l}\texttt{C}\\ \|\\ \bullet\\ \|\\ \circ\end{array} \Lsh \begin{array}{l}\bullet\\ \|\\ \circ - \texttt{ch-}_p\end{array} \Lsh \begin{array}{l}\bullet\\ \|\\ \circ - \texttt{D}\end{array}\right)\right)$$

$= \hat{}(\texttt{C}\ \texttt{ch+}_p\ \texttt{ch-}_p\ \texttt{D})$ *(linear notation)*

$$aqq = Collapse\left(\begin{array}{l}\bullet\\ \|\\ \circ - \texttt{ch+}_q\end{array} \Lsh \left(\begin{array}{l}\texttt{E}\\ \|\\ \bullet\\ \|\\ \circ\end{array} \Lsh \begin{array}{l}\bullet\\ \|\\ \circ - \texttt{ch-}_q\end{array} \Lsh \begin{array}{l}\bullet\\ \|\\ \circ - \texttt{D}\end{array}\right)\right)$$

$= \hat{}(\texttt{E}\ \texttt{ch+}_q\ \texttt{ch-}_q\ \texttt{D})$ *(linear notation)*

$$aq\bot = Collapse\left(\begin{array}{l}\bullet\\ \|\\ \circ - \texttt{ch+}_\bot\end{array} \Lsh \left(\begin{array}{l}\bullet\\ \|\\ \circ\end{array} \Lsh \begin{array}{l}\bullet\\ \|\\ \circ - \texttt{ch-}_\bot\end{array} \Lsh \begin{array}{l}\bullet\\ \|\\ \circ - \texttt{D}\end{array}\right)\right)$$

(Note that any reference to $\texttt{ch+}_\bot$ *and* $\texttt{ch-}_\bot$ *is to be discarded.)*
$= \texttt{D}$ *(linear notation)*

$$apr = Collapse\left(\begin{array}{l}\bullet\\ \|\\ \circ - \texttt{F}\end{array} \Lsh \begin{array}{l}\bullet\\ \|\\ \circ - \texttt{ch-}_r\end{array}\right)$$

$= \texttt{F}\ \texttt{ch-}_r$ *(linear notation)*

$$ap\bot = Collapse\left(\begin{array}{l}\bullet\\ \|\\ \circ\end{array} \Lsh \begin{array}{l}\bullet\\ \|\\ \circ - \texttt{ch-}_\bot\end{array}\right)$$

(Note that any reference to $\texttt{ch-}_\bot$ *is to be discarded.)*
$=$

$$ar\bot = Collapse\left(\begin{array}{l}\bullet\\ \|\\ \circ\end{array} \Lsh \begin{array}{l}\bullet\\ \|\\ \circ - \texttt{ch-}_\bot\end{array}\right)$$

(Note that any reference to $\texttt{ch-}_\bot$ *is to be discarded.)*
$=$

Figure 3.28: Definitions 3.6.3 and 3.6.4 applied to all patterns from figure 3.23.

```
start
  : ^(C ch+p ch-p)
  | ^(E ch+q ch-q)
  | F ch-r
  ;
ch+p
  : ^(C ch+p ch-p A B)
  | A B
  ;

ch+q
  : ^(C ch+p ch-p D)
  : ^(E ch+q ch-q D)
  | D
  ;
ch-p
  : F ch-r
  |
  ;
ch-q
  : F ch-r
  |
  ;
ch-r
  :
  ;
```

Figure 3.29: The results from figures 3.27 and 3.28 in one grammar.

Chapter 4

Optimisation of the tree grammar

The context-free tree grammar generated by the conversion algorithm from chapter 3 is a rather precise match for the parser specification. Tree grammars are usually used to implement compiler steps such as optimisers and code generators; these are designed around the more general AST. For example, implementing correct operator precedence usually takes several rules in a parser specification. A code generator specification could suffice with one rule where all operators are alternatives in that rule as the operator precedence is already correctly parsed into the AST by the parser. Therefore, a precise match with the parser is usually not needed, since the context-free tree grammar can be more general. This chapter describes how the tree walker specification can be cleaned up and generalised to make it more practical to use.

The optimisations described below are based on the manner in which the rules were generated. Note that all rules in the context-free tree grammar are generated according to the formulæ in definitions 3.6.1, 3.6.2, 3.6.3 and 3.6.4 and take the form of $\rho = a_1 \,|\, ... \,|\, a_n$, where $a_1 ... a_n$ are FCNS-trees. The set $a_1 ... a_n$ will be designated by $F(\rho)$ in this chapter.

4.1 Macro expansion

Macro expansion is a procedure that combines rules without affecting the language the grammar accepts. A basic example of this is a non-terminal that refers to a rule that only consists of one terminal. In this case, all instances of this non-terminal can be replaced by this terminal.

To prevent making the grammar more complex instead of more readable, there are some limitations to the macro expansion procedure. Rules with multiple alternatives and rules that contain FCNS-trees with a depth of more than one are not expanded. Although this would be semantically correct , it would make further optimisations more complex as the rules no longer take the form of $\rho = a_1 \,|\, ... \,|\, a_n$.

The merging procedure itself is quite similar to the stacking procedure defined in definition 3.2.4. The notation $u(\mathtt{a})$, used in the definition below, denotes the vertex in the FCNS-tree u that is labelled with the non-terminal $\mathtt{a}$.

Definition 4.1.1 *Macro expansion of two FCNS-trees is denoted by* $MacroExpand : \mathcal{F} \times \mathcal{N} \times \mathcal{F} \rightarrow \mathcal{F}$. *This expands the second FCNS-tree into the first FCNS-tree on the specified non-terminal and results in a new FCNS-tree.*

$w = MacroExpand(u, \texttt{a}, v)$, *where:*

$$
\begin{aligned}
V(w) = &\quad V(v) \cup (V(u) - \{u(\texttt{a})\}) \\
E(w) = &\quad E(v) \cup E(u) \cup \{ \\
&\quad (PreviousSibling(u(\texttt{a})) \overset{ns}{\rightarrow} Root(v)), \\
&\quad (LastSibling(Root(v)) \overset{ns}{\rightarrow} NextSibling(u(\texttt{a}))), \\
&\quad (Parent(u(\texttt{a})) \overset{fc}{\rightarrow} Root(v))\}
\end{aligned}
$$

Any dangling edges will be discarded.

For example; The rules `start` and `tail` from figure 4.1 can be expanded according to definition 4.1.1.

```
start
     A
     ‖
  :  B — tail
  ;

tail
  :  D — E
  ;
```

$$MacroExpand\left(\begin{array}{l}\texttt{A}\\ \Vert \\ \texttt{B} - \texttt{tail}\end{array}, \texttt{tail}, \texttt{D} - \texttt{E}\right) = \begin{array}{l}\texttt{A}\\ \Vert \\ \texttt{B} - \texttt{D} - \texttt{E}\end{array}$$

Figure 4.1: An example of "Macro expansion"

4.2 Rule merging

The macro expansion method will only clean up the most trivial rules from the context-free tree grammar. The rule merging method described in this section is an algorithm that can significantly generalise the context-free tree grammar. Theoretically, it could even reduce the entire context-free tree grammar to a single rule.

The procedure works as follows: if a rule σ is referred to via one or more non-terminals from rule ρ, then all alternatives of rule σ are unified with the alternatives from rule ρ. All references to rule σ are replaced by references to rule ρ. Figure 4.3 demonstrates the merging of the two rules from figure 4.2 according to definition 4.2.1.

Definition 4.2.1 *Merging of two rules is denoted by* $RuleMerge : \mathcal{R} \times \mathcal{R} \rightarrow \mathcal{R}$. *This merges all alternatives from both rules.*
$\tau = RuleMerge(\rho, \sigma)$, *where:*
$\tau = a_1 \,|\, ... \,|\, a_n$, *where* $a_1 ... a_n = F(\rho) \cup F(\sigma)$
Note that all references to ρ *and* σ *should be replaced by a reference to* τ *in the entire grammar.*

```
start
  : ^(A B C D)
  | ^(B C D rule)
  ;

rule
  : ^(E F G)
  | H
  ;
```

Figure 4.2: A trivial mergable context-free tree grammar

```
start
  : ^(A B C D)
  | ^(B C D start)
  | ^(E F G)
  | H
  ;
```

Figure 4.3: The context-free tree grammar from figure 4.2 merged according to definition 4.2.1

4.2.1 Limitations to rule merging

Rules are not to be merged if the merged rule introduces LL(1) violations and/or left-recursion anywhere in the grammar. Consider the context-free tree grammar from figure 4.4. The rules `start` and `ruleA` are not merged as it would introduce a rule that has a root that resolves to itself, as demonstrated in figure 4.5. And the rules `start` and `ruleB` are not merged as it would introduce two alternatives with roots that resolve to the same terminal, as demonstrated in figure 4.6.

As mentioned, if the parser specification is not too complex, this will can result in a tree walker specification with only one rule. Therefore, the parser designer is still able to retain specified rules by annotating them in the parser specification with a `^`, this will prevent them from being merged with another rule. It will not, however, prevent other rules being merged with the annotated rule.

```
start
  : ruleA B C
  | ^(B C D ruleB)
  ;

ruleA
  : ^(E G)
  | ^(H I)
  ;

ruleB
  : ^(B E F G)
  | ^(H I)
  ;
```

Figure 4.4: A context-free tree grammar with rules that can not be merged

```
start
  : start B C
  | ^(B C D ruleB)
  | ^(E G)
  | ^(H I)
  ;
```

Figure 4.5: The result of a merge of rules `start` and `ruleA` from 4.4

```
start
  : ruleA B C
  | ^(B C D start)
  | ^(B E F G)
  | ^(H I)
  ;
```

Figure 4.6: The result of a merge of rules `start` and `ruleB` from 4.4

4.3 Further optimisations

Besides macro expansion and rule merging, there are some trivial optimisations that hardly need to be mentioned. These optimisations are merely present to clean up things that are left behind by the other algorithms, so the implementations of these algorithms can be kept simpler. For example many implemented algorithms do not check if parentheses are really necessary, they simply always place parentheses to prevent conflicts. Excessive parentheses are removed by a simple algorithm later. None of the optimisations below make significant changes to the generated tree walker, they merely clean up the grammar.

- Rules with identical content are removed; the algorithm that generates the walker from the parser may generate several rules with identical content, these are redundant and can be represented by one rule.
- Alternatives within one rule with identical content are removed; the algorithm that generates the walker from the parser and the optimisation from section 4.2 sometimes generate rules with two or more identical alternatives, they are redundant and can be merged into one.
- Sometimes rules are generated that are not referenced by other rules; this may happen as a result of some of the other optimisations. All rules that are not referenced, either directly or indirectly, from the start rule are removed.
- The algorithm also generates optional rules as BNF rules with an empty alternative, these are changed to EBNF rules where the empty alternative is removed and all invocations of the rule are annotated with an EBNF `?` operator. If these rules invoke themselves recursively; these are changed to the EBNF `+` operator.
- And finally, all empty rules and superfluous parentheses are removed.

The optimisations in this section will clean up context-free tree grammar from figure 4.7 to the tree grammar as demonstrated in figure 4.8. First, all references to `ruleC` are replaced by references to `ruleA` as they are both identical. Next, the second alternative `^(A B C D)` is removed from rule `start`. Next, `ruleB` and `ruleC` are removed as they are not referenced by any other rule. And finally, the parentheses around `F G` in `ruleA` are removed.

```
start
  : ^(A B C D)
  | ^(B ruleA)
  | ^(A B C D)
  | ^(D ruleC)
  ;

ruleA
  : ^(E (F G))
  | A
  ;

ruleB
  : ^(A B C D)
  ;

ruleC
  : ^(E (F G))
  | A
  ;
```

Figure 4.7: An unoptimised context-free tree grammar

```
start
  : ^(A B C D)
  | ^(B ruleA)
  | ^(D ruleA)
  ;

ruleA
  : ^(E F G)
  | A
  ;
```

Figure 4.8: The optimisations from section 4.3 applied to the tree grammar from figure 4.7

Chapter 5

Implementation of ANTLR$_{TG}$

To be able to perform a case study, a Java implementation of the algorithm has been made. Java has been chosen as implementation language to ease the integration with ANTLR, since it is also implemented in Java. This section will briefly describe the design and some important details of this implementation. To aid understanding the code, several classes that implement important parts of the algorithm will be explained briefly in this section. For exact details, please refer to the JavaDoc documentation and the source code.

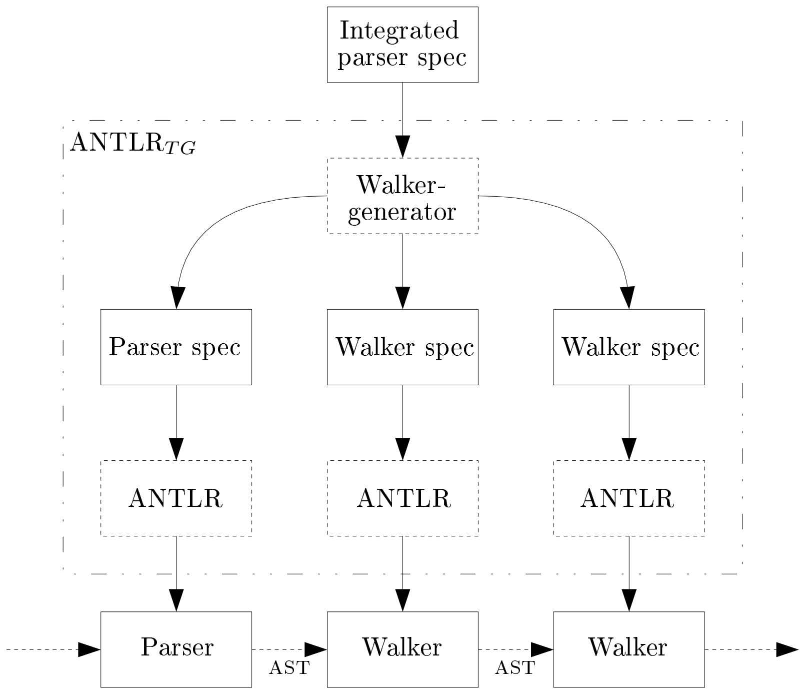


Figure 5.1: Architectural overview

5.1 Architectural overview

Figure 5.1 demonstrates an abstracted flow of information for a compiler with a parser and two tree walkers. The dashed boxes represent automated processes or tools. The input for this process is an integrated parser specification, containing the information needed to generate the parser and tree walkers. A walker generator splits this integrated specification into the different specifications needed by ANTLR. The last step is to invoke ANTLR with the generated specifications to generate the code for the parser and tree walkers.

5.1.1 Integrated parser specification

The integrated parser specification plays a key role in this process. It needs to contain the information that is required to generate a parser and the tree walkers, but must not be redundant. ANTLR parser and tree walker specifications contain context-free grammars, that specify the language that is to be parsed, and source code annotations that are inserted into the generated code. For the integrated parser specification these separate specifications are merged into one. Only the context-free grammar for the ANTLR parser specification is used in the integrated parser specification, the context-free grammars for the tree walkers are internally generated from the parser specification. The source code annotations for the parser and tree walkers are all attached to the same context-free grammar, a specialised editor shall be provided to keep this comprehensible to the user.

5.1.2 Walker generator

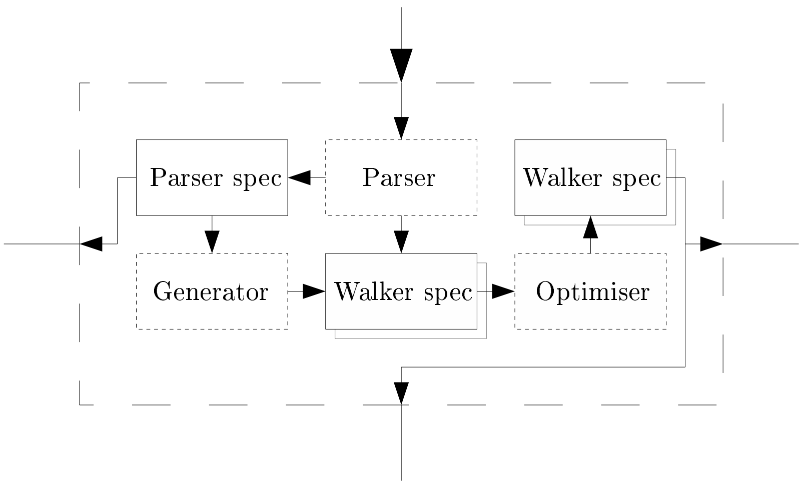


Figure 5.2: Walker generator

An important node in the architecture is the walker generator. Figure 5.2 zooms in on the walker generator, the dashed boxes again represent automated processes. The parser parses the integrated parser specification, which generates the ANTLR parser specification and sets aside the source code annotations for the tree walkers. Next the generator will generate the raw tree walker specifications for each tree walker and merges the source code annotations. Finally the optimiser will optimise walker specifications so that is practically usable.

5.2 Design

The architecture is broken down into several parts. The part that handles the integrated parser specification is implemented in the Java package *org.antlrtg.grammar*. The part that implements the algorithms described in chapter 3 is implemented in the Java package *org.antlrtg.algo* with some overlap with the *org.antlrtg.grammar* package. Two more packages are provided with some helper classes and JUnit unit tests in the packages *org.antlrtg.utilities* and *org.antlrtg.test*. The dependencies between these packages are depicted in figure 5.3.

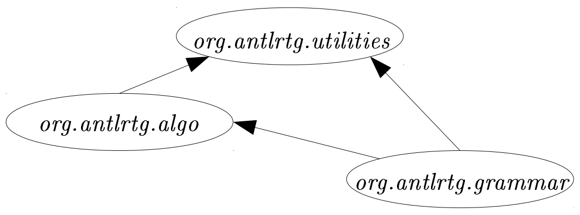


Figure 5.3: The dependencies between packages in *org.antlrtg.**

5.2.1 Algorithm

The Java package *org.antlrtg.algo*, for which the simplified UML class diagram is depicted in figure 5.4, contains the code that implements most of the algorithms described in chapter 3. It contains a representation of the FCNS tree, the tree pattern and a rule generator that implements the formulæ from section 3.6. Note that extracting the sets RT, CH^+ and CH^- is provided by the interface *org.antlrtg.algo.PatternSets*, but it is actually implemented in the grammar package by the class *org.antlrtg.grammar.PatternSets* as it requires access to the grammar.

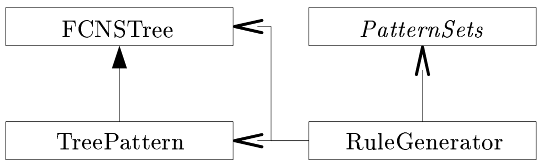


Figure 5.4: The simplified UML class diagram for *org.antlrtg.algo*

The class *org.antlrtg.algo.FCNSTree* implements the FCNS tree as described in section 2.2.2, the source code for this class has been provided in appendix A.1. The class has a constructor that restores the FCNS tree from its linear representation and a counterpart-method *toString()* that converts it back to a linear representation. Furthermore, it provides some methods that provide access to the root node of the FCNS tree and query properties such as the depth of the FCNS tree. The class provides one subclass *org.antlrtg.algo.FCNSTree.Node* which models a node in the FCNS tree. It provides methods to query the first child, next sibling and last sibling of the node.

The class *org.antlrtg.algo.TreePattern* extends the class *FCNSTree* and implements the tree pattern as described in section 3.2, the source code for this class has been provided in appendix A.2. The class also has a constructor that restores the tree pattern from its linear representation and its counterpart *toString()*. Furthermore it has a constructor to construct one of the elementary tree patterns. The class provides several methods to query properties such as if it is a master or slave pattern. The class also provides methods to stack two tree patterns, retrieve the top or bottom part, collapse it to an FCNS tree and decompose it to a set of elementary tree patterns.

The class *org.antlrtg.algo.RuleGenerator* generates the rules of the tree grammar as described in section 3.6, it uses the information provided by the *PatternSets* interface to build them. The rules are provided as a set of FCNS trees which are adapted by the class *PatternGrammarGen* to the *Grammar* interface.

5.2.2 Grammar

The Java package *org.antlrtg.grammar*, for which the simplified UML class diagram is depicted in figure 5.5, contains the code that handles the integrated parser specification. It is built around the main grammar file from ANTLR, *ANTLRv3.g*, which has been annotated with calls to the *GrammarGenerator* interface according to the visitor design pattern [15], the source code for this class has been provided in appendix A.3. The *ANTLRv3.g* grammar file is converted to the class *ANTLRv3Parser* by ANTLR. The *GrammarGenerator* interface has been implemented by the class *PatternGrammarGen*, which is responsible for building an annotated context-free grammar as described in section 3.4 from the integrated parser specification. The class *PatternGrammarNF* is able to take the generated annotated context-free grammar and bring it to normal-form as described in the same section.

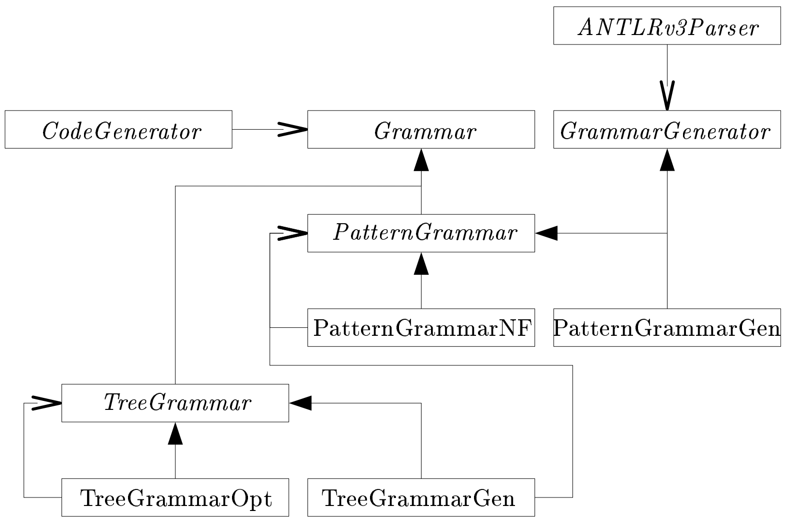


Figure 5.5: The simplified UML class diagram for *org.antlrtg.grammar*

The next step is to generate a context-free tree grammar according to section 3.5, this is implemented in the class *TreeGrammarGen*. This class takes a *PatternGrammar* and builds a context-free tree grammar from it. The class *TreeGrammarOpt* is able to take the generated context-free tree grammar and optimise it according to chapter 4.

The last step is to generate code again that can be parsed by ANTLR to generate a lexical analyser, parser, or tree walker. This has been implemented in the class *CodeGenerator*, which can take any grammar and generate code that can be parsed by ANTLR.

5.3 Usage

ANTLR$_{TG}$ requires the user to make a standard ANTLR/EBNF parser specification, this does not deviate from the normal ANTLR way of working. The user should then annotate the rules with carets to specify the root nodes for each rule, these will structure the AST. Figure 5.6 demonstrates a simple ANTLR grammar named "Test", which parses sentences like "36 + 22". The corresponding tree walker, generated with ANTLR$_{TG}$, is depicted in figure 5.7.

5.3.1 Action annotations

ANTLR allows users to annotate terminals and non-terminals in a grammar specification with (Java) code. This code is then executed as the generated parser parses a token, or when the generated tree walker visits a node in the AST. These code annotations are placed between brackets:

```
{ System.out.println("Hello world"); }
```

Figure 5.6 already contains such an annotation in the `WHITESPACE` lexical analyser rule, which is an ANTLR specific statement that hides all white-space tokens from the AST.

Annotating integrated parser specifications

ANTLR$_{TG}$ extends the syntax for action annotations slightly to support placing the annotations for all tree walkers in the integrated parser specification. In ANTLR$_{TG}$, action annotations can be started with '`{@`' followed by a number and a white-space to specify the tree walker to be placed in:

```
{@1 System.out.println("Hello world"); }
```

The number following the '`@`' denotes the tree walker the action annotation is placed in, 1 is the first tree walker to be generated, 2 the second, etc. The action annotation is placed in the generated parser specification if the '`@`' is followed by the number 0. Omitting this prefix on action annotations will place the action in the all generated specifications, note that lexer rules (such as `WHITESPACE` and `INTEGER_LITERAL` in figure 5.6) are only present in the parser specification.

Tree walkers

The structure of the tree walker specification may differ significantly from the parser specification. The order of the terminals and non-terminals may have changed, additional non-terminals may have been introduced as explained in chapter 3 and other non-terminals may have been optimised out as explained in chapter 4. To be sure all non-terminals can be annotated with an

```
grammar Test;

options {
  output=AST;
}

tokens {
  PLUS = '+' ;
}

// Parser rules
add
  : INTEGER_LITERAL PLUS^ INTEGER_LITERAL
  ;

// Lexer rules
INTEGER_LITERAL
  : ('0' .. '9')+
  ;

WHITESPACE
  : (' ' | '\t')+
    { $channel=HIDDEN; }
  ;
```

Figure 5.6: An ANTLR/EBNF parser specification

```
tree grammar TestChecker;

options {
  output = AST;
  backtrack = true;
}

tokens {
  PLUS;
  WHITESPACE;
}

add
  : ^( PLUS INTEGER_LITERAL INTEGER_LITERAL )
  ;
```

Figure 5.7: The generated tree walker for figure 5.6.

action from the integrated parser specification, a '+' followed by a number can be appended to the '`@`' notation:
`{@2+1 System.out.println("Hello world"); }`
The number following the '+' denotes the number of terminals/non-terminals this action annotation should be delayed from the perspective of the tree walker. Figure 5.8 presents some action annotation examples.

```
// Integrated parser specification
add
  : INTEGER_LITERAL PLUS^
    {@1 System.out.println("1"); }
    {@1+1 System.out.println("1+1"); }
    INTEGER_LITERAL
    {@1 System.out.println("X"); }
  ;

// First tree walker
add
  : ^( PLUS
    { System.out.println("1"); }
    INTEGER_LITERAL
    { System.out.println("1+1"); }
    INTEGER_LITERAL
    { System.out.println("X"); }
    )
  ;
```

Figure 5.8: Some action annotation examples.

5.3.2 Invocation

ANTLR$_{TG}$ can be invoked either from another Java application or from the console. Analogous to ANTLR, ANTLR$_{TG}$ has an *org.antlrtg.Tool* class that provides a constructor that takes a list of arguments *String[] args* and provides a *process()* method that actually processes the request.

From the console, ANTLR$_{TG}$ can be invoked by executing the main function in the class *org.antlrtg.Tool*. The following files need to be in the Java classpath: *stringtemplate.jar*, *antlr3.jar* and either *antlrtg.jar* or the output directory where ANTLR$_{TG}$ is built. The arguments consist of:
`[-O<n>] <inputfile> [-op <outputfile>] [-ow[<n>] <outputfile>]`
Where `-op` specifies the output file for the parser specification, `-ow<n>` specifies the output file for the (n^{th}) tree walker specification and `-O<n>` the level of optimisation to use (0 = no optimisation, 1 = full optimisation (default)). Figure 5.9 shows an example ANT build rule to add ANTLR$_{TG}$ to an ANT build script.

```
<target name="-pre-compile">
  <exec dir="src" executable="java">
    <arg value="-cp"/>
    <arg value="../ext/stringtemplate-3.2.jar:
                ../ext/antlr-3.1.3.jar:
                ../ext/antlrtg.jar"/>
    <arg value="org.antlrtg.Tool"/>
    <arg value="IntegratedParserSpec.tg"/>
    <arg value="-op"/><arg value="Parser.g"/>
    <arg value="-ow1"/><arg value="Checker.g"/>
    <arg value="-ow2"/><arg value="Generator.g"/>
  </exec>
  <exec dir="src" executable="java">
    <arg value="-cp"/>
    <arg value="../ext/stringtemplate-3.2.jar:
                ../ext/antlr-3.1.3.jar"/>
    <arg value="org.antlr.Tool"/>
    <arg value="Parser.g"/>
    <arg value="Checker.g"/>
    <arg value="Generator.g"/>
  </exec>
</target>
```

Figure 5.9: Adding ANTLR$_{TG}$ to an ANT build script

Chapter 6

Case study: Triangle

The previous chapters established an algorithm and implementation to obtain an ANTLR tree-walker from an ANTLR parser specification. To confirm the practical usability of this algorithm, a case study has been executed for a compiler for a language called Triangle [7]. Triangle is a Pascal-like language, but much simpler and more regular. A simple Triangle example is demonstrated in figure 6.1.

```
let
  const newline ~ 0x0A;

  proc getline (var length: Integer, var content: array 80 of Char) ~
          let var c : Char
          in
            begin
            length := 0;
            c := getchar();
            while c \ newline do
              begin
              content[length] := c;
              length := length + 1;
              c := getchar()
              end
            end;

  var length: Integer;
  var content: array 80 of Char

in
  begin
  printf('Please enter a line of text: ');
  getline(var length, var content);
  content[length] := newline; content[length+1] := 0;
  printf('You provided the following line: %s', content)
  end
```

Figure 6.1: A simple Triangle example

This case study will implement a parser for this language and automatically let ANTLR$_{TG}$ generate two tree-walkers for it: a checker and a generator. Afterwards, the tree walkers are evaluated for correctness and practical usability.

6.1 Design of the Triangle compiler

The compiler for this case study will parse a Triangle program and check it for semantic correctness and will output ISO/ANSI C code. C has been chosen for this case study as it is can be easily verified manually. Figure 6.2 presents the three logical components of the triangle compiler.

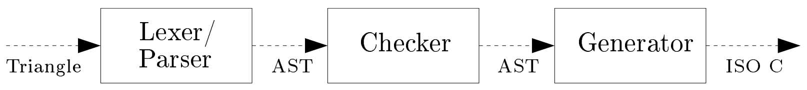


Figure 6.2: Triangle compiler overview

The Triangle compiler will be written in Java using ANTLR$_{TG}$. The specification for the parser will be written in an ANTLR$_{TG}$ integrated parser specification (.tg file) which contains a standard ANTLR parser specification annotated with the custom actions for the tree walkers that will be generated. This ANTLR$_{TG}$ integrated parser specification will then be processed by the ANTLR$_{TG}$ tool presented in chapter 5, resulting in an ANTLR parser specification and two ANTLR tree walker specifications (.g files). The ANTLR specifications can then be processed by ANTLR resulting in a set of Java files that form the most significant parts of the compiler.

6.1.1 Lexical analyser and parser

The first stages of the compiler are the lexical analyser and the parser, these will parse the input file and build an AST for it. Both are specified in the ANTLR$_{TG}$ integrated parser specification. The procedure for specifying these does not deviate from the procedure for specifying a standard ANTLR lexical analyser and parser [8]. Figure 6.3 demonstrates a code snippet of the ANTLR/EBNF specification of the parser, the fully annotated ANTLR$_{TG}$/EBNF integrated parser specification is provided in Appendix B.1. No custom actions are required for the parser, a standard ANTLR parser is used for this case study.

Note that, in the design of this compiler, the only task for the parser is to check the syntax of the input and generate an AST. Therefore the only deviations of the parser specification from a pure EBNF specification are the `^` and `!` annotations on terminals to determine the shape of the AST. The rest of the compiler tasks as performed by the checker and generator.

6.1.2 Checker

The checker is used to check the semantic correctness of the file (e.g. confirm that a variable is declared before it is used). The checker is a tree walker, which means it will traverse the AST generated by the parser. The checker will cross reference identifiers that are related, e.g. variables with their declarations and types with their definitions.

```
declaration
  : single_declaration (SEMICOLON! single_declaration)*
  ;

single_declaration^
  : CONST^ IDENTIFIER EQUIV! expression
  | VAR^ IDENTIFIER COLON! type_denoter
  | PROC^ IDENTIFIER
    LPAREN! formal_parameter_sequence RPAREN!
    EQUIV! single_command
  | FUNC^ IDENTIFIER
    LPAREN! formal_parameter_sequence RPAREN!
    COLON! type_denoter EQUIV! single_command
  | TYPE^ IDENTIFIER EQUIV! type_denoter
  ;
```

Figure 6.3: ANTLR/EBNF specification of the parser

Checking declarations

To be able to cross reference the type definitions and declarations, the compiler uses special AST nodes that are annotated with a reference to another AST node. This reference is used to follow declarations, for example: `p := 0;`, where the identifier `p` is declared by `var p : Point` and therefore the AST node for `p` is linked to the AST node for `Point`. In term, the identifier `Point` is defined by `type Point ~ Integer;` and therefore the AST node for `Point` is linked to the AST node for `Integer`. Figure 6.4 presents a graphical representation of this concept.

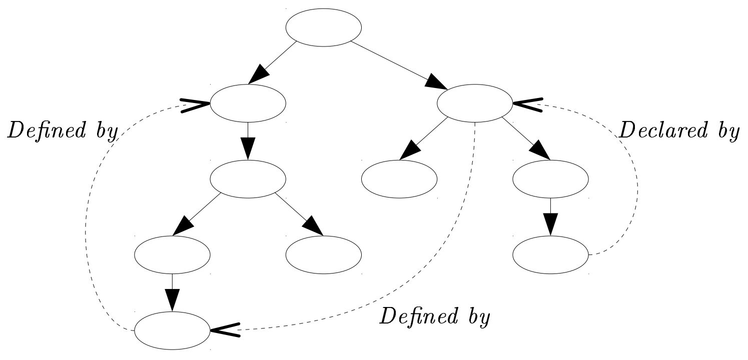


Figure 6.4: Additional AST annotations follow the declarations and definitions.

To be able to correctly cross reference the type definitions and declarations, the checker maintains a stack of scopes (named `scopeStack`). This stack contains the hierarchical structure of all type definitions and declarations, where a new entry is pushed each time a scope is entered and popped each time a scope is left. Hence, new declarations and definitions are added to the scope on the top of the stack. An error is generated when a reference can not be made, i.e. a declaration or type definition is not available.

To be able to recursively specify types, the checker maintains a stack of types (named `typeStack`) and a stack of records (named `recordStack`). An entry is pushed on the respective stack each

```
declaration
  : single_declaration (SEMICOLON! single_declaration)*
  ;

single_declaration^
  : CONST^ IDENTIFIER EQUIV! const_denoter
    {@1 $IDENTIFIER.dataType = typeStack.pop();
        scopeStack.peek().variables.add($IDENTIFIER.text, $IDENTIFIER); }

  | VAR^ IDENTIFIER COLON! type_denoter
    {@1 $IDENTIFIER.dataType = typeStack.pop();
        scopeStack.peek().variables.add($IDENTIFIER.text, $IDENTIFIER); }

  | PROC^ IDENTIFIER
    {@1 scopeStack.push(new TriangleTreeNode.ProcType()); }
    LPAREN! formal_parameter_sequence RPAREN!
    EQUIV! single_command
    {@1 $IDENTIFIER.dataType = scopeStack.pop();
        scopeStack.peek().variables.add($IDENTIFIER.text, $IDENTIFIER); }

  | FUNC^ IDENTIFIER
    {@1 scopeStack.push(new TriangleTreeNode.FuncType()); }
    LPAREN! formal_parameter_sequence RPAREN!
    COLON! type_denoter EQUIV! single_command
    {@1 $IDENTIFIER.dataType = scopeStack.pop();
        ((TriangleTreeNode.FuncType)$IDENTIFIER.dataType).type =
          typeStack.pop();
        scopeStack.peek().variables.add($IDENTIFIER.text, $IDENTIFIER); }

  | TYPE^ IDENTIFIER EQUIV! type_denoter
    {@1 $IDENTIFIER.dataType = typeStack.pop();
        scopeStack.peek().typedefs.add($IDENTIFIER.text, $IDENTIFIER); }
  ;
```

Figure 6.5: Figure 6.3 annotated with the checker code.

```
declaration_suf_2
  : (single_declaration )+
  ;

single_declaration
  : ^( CONST type_denoter_pre_3 IDENTIFIER const_denoter
      { $IDENTIFIER.dataType = typeStack.pop();
        scopeStack.peek().variables.add($IDENTIFIER.text, $IDENTIFIER); }
     )
  | ^( VAR type_denoter_pre_3 IDENTIFIER type_denoter
      { $IDENTIFIER.dataType = typeStack.pop();
        scopeStack.peek().variables.add($IDENTIFIER.text, $IDENTIFIER); }
     )
  | ^( PROC type_denoter_pre_3 IDENTIFIER
      { scopeStack.push(new TriangleTreeNode.ProcType()); }
     formal_parameter_sequence? single_command
      { $IDENTIFIER.dataType = scopeStack.pop();
        scopeStack.peek().variables.add($IDENTIFIER.text, $IDENTIFIER); }
     )
  | ^( FUNC type_denoter_pre_3 IDENTIFIER
      { scopeStack.push(new TriangleTreeNode.FuncType()); }
     formal_parameter_sequence? type_denoter single_command
      { $IDENTIFIER.dataType = scopeStack.pop();
        ((TriangleTreeNode.FuncType)$IDENTIFIER.dataType).type =
        typeStack.pop();
        scopeStack.peek().variables.add($IDENTIFIER.text, $IDENTIFIER); }
     )
  | ^( TYPE type_denoter_pre_3 IDENTIFIER type_denoter
      { $IDENTIFIER.dataType = typeStack.pop();
        scopeStack.peek().typedefs.add($IDENTIFIER.text, $IDENTIFIER); }
     )
  ;
```

Figure 6.6: The checker code generated by ANTLR$_{TG}$ for figure 6.5.

time a type or record is denoted, an entry is popped each time it is used in a definition or declaration.

Source code

The tree walker specification for the checker is not explicitly specified, instead it is generated by ANTLR$_{TG}$. The custom actions to be performed on the visiting of an AST node by the checker are specified in the integrated parser specification, ANTLR$_{TG}$ will put them at the correct position in the ANTLR tree walker specification as explained in section 5.3.1. Figure 6.5 demonstrates the code snippet from figure 6.3 with the action annotations for the checker appended. Note that the rule `type_denoter` pushes one element on the `typeStack`, see Appendix B.2 for the generated tree walker specification for the checker.

Figure 6.6 contains the generated tree walker specification for the rules presented in figure 6.5. The `declaration` rule has been renamed as a side effect of the optimiser, this could have been prevented by annotating the rule with a `^` like the `single_declaration` rule, and has been simplified because the `SEMICOLON` terminal was annotated with a `!` and therefore left out of the AST. The `single_declaration` rule is a clean rewrite of the parser rule.

6.1.3 Generator

The generator will output ISO/ANSI C code for the Triangle program. Like the checker, the generator is also a tree walker, which will walk the same tree that the checker has annotated. The ANTLR tree walker specification for the generator is also generated by ANTLR$_{TG}$, the custom actions for the generator are therefore also specified in the integrated parser specification.

Generating code

The actual code generation has been implemented in a separate class `CGenerator` which implements the interface `Generator`. This interface is used for the output language specific parts, the more generic parts are kept in the tree walker. Similar to the checker, the generator also maintains a stack to deal with scopes. This stack, named `codeStack` contains all the generated code for the current scope, which is merged with the rest of the program once the scope is closed.

Source code

Also the tree walker specification for the generator is generated by ANTLR$_{TG}$. Figure 6.7 demonstrates the code snippet from figure 6.3 with the action annotations for the checker appended, see Appendix B.3 for the generated tree walker specification for the generator.

6.2 Generated C code

Several types and functions from the standard C library have been directly made available to Triangle. These types from `stdint.h` are `int8_t`, `int16_t`, `int32_t` and `int64_t`, which are respectively exposed as `Byte`, `Short`, `Integer` and `Long`. The following functions from `stdio.h`

```
declaration
  : single_declaration (SEMICOLON! single_declaration)*
  ;

single_declaration^
  : CONST^ IDENTIFIER EQUIV! const_denoter
    {@2 codeStack.peek().addType(
          generator.generateDefinition($IDENTIFIER)); }

  | VAR^ IDENTIFIER COLON! type_denoter
    {@2 codeStack.peek().addDeclaration(
          generator.generateDeclaration($IDENTIFIER)); }

  | PROC^ IDENTIFIER
    {@2 codeStack.push(generator.openScope(
          generator.generateDefinition($IDENTIFIER), codeStack.peek())); }
    LPAREN! formal_parameter_sequence RPAREN!
    EQUIV! single_command
    {@2 String code = generator.closeScope(codeStack.pop());
        codeStack.peek().addMethod(code); }

  | FUNC^ IDENTIFIER
    {@2 codeStack.push(generator.openScope(
          generator.generateDefinition($IDENTIFIER), codeStack.peek())); }
    LPAREN! formal_parameter_sequence RPAREN!
    COLON! type_denoter EQUIV! single_command
    {@2 String code = generator.closeScope(codeStack.pop());
        codeStack.peek().addMethod(code); }

  | TYPE^ IDENTIFIER EQUIV! type_denoter
    {@2 codeStack.peek().addType(
          generator.generateDeclaration($IDENTIFIER)); }
  ;
```

Figure 6.7: Figure 6.3 annotated with the generator code.

are exposed under the same name: `getchar`, `printf` and `putchar`. Figure 6.9 presents the generated ISO/ANSI C code for the Triangle example of figure 6.1.

The set of reserved keywords in Triangle is different from that of C. For example, it is fully legal to define a Triangle variable with the name `volatile`, which is a reserved keyword in C. Therefore, all Triangle identifiers have been prefixed with "`tr_`" to prevent name clashes with standard C library methods and reserved keywords. Furthermore, as constants are untyped in Triangle, all constants are defined using the C pre-processor with a `#define` directive.

Triangle allows passing variables by reference to a function or procedure, this has been replaced by passing a pointer to a variable to a function in the generated C code. In this case, the function argument is explicitly dereferenced everywhere it is used.

The C code should compile on any ISO/ANSI compatible C compiler, but only GCC 4.3.4 has been tested. Figure 6.8 demonstrates how to compile and execute the generated program.

```
bash$ cc fullexample.c -o fullexample
bash$ ./fullexample
Please enter a line of text: This is a simple test!
You provided the following line: This is a simple test!
bash$
```

Figure 6.8: Compiling the generated C code.

6.3 Evaluation

During the building of the Triangle compiler, it became almost immediately apparent that building a compiler with ANTLR$_{TG}$ is significantly easier compared to manually writing a parser and tree walkers. There is more freedom to iteratively improve the parser specification without the risk of accidentally invalidating one of the tree walkers. Several bugs in the implementation of ANTLR$_{TG}$ were discovered during the development of the Triangle compiler, some of these bugs resulted in a tree walker that did not properly match the AST generated by the parser. Detecting the mismatches of the tree walker with the AST alone took many hours, which shows that automatically generating tree walkers does improve compiler building with ANTLR.

The usefulness of the generated tree walkers is mainly determined by the correctness, the readability and the possibility to correctly position the action annotations in the tree walkers from the integrated parser specification. Chapter 3 presented some theoretical examples where the generated tree walker would differ significantly and many additional rules could be generated, in practise these constructions are hardly used and no more than one terminal annotated with a ^ is used in one alternative. Figures 6.5 and 6.6 are typical for the complexity encountered and demonstrate that it is not very hard to follow what the tree walker generator is doing.

```
#include <sys/types.h>
#include <stdint.h>
#include <stdio.h>

#define tr_newline 0x0A

void tr_getline(int32_t  * tr_length, char  * tr_content)
{
  {
    char tr_c;

    {
      *tr_length = 0;
      tr_c = getchar();
      while (tr_c != tr_newline)
      {
        tr_content[*tr_length] = tr_c;
        *tr_length = *tr_length + 1;
        tr_c = getchar();
      }
    }
  }
}

int main(int argc, char **argv)
{
  int32_t tr_length;
  char tr_content[80];

  {
    printf("Please enter a line of text: ");
    tr_getline(&tr_length, tr_content);
    tr_content[tr_length] = tr_newline;
    tr_content[tr_length + 1] = 0;
    printf("You provided the following line: %s", tr_content);
  }

  return 0;
}
```

Figure 6.9: The generated C code for the example of figure 6.1.

Chapter 7

Conclusions

The first chapter of this thesis identified a problem with the development and maintainability of compilers using ANTLR. The following chapters provided a possible solution to this problem by means of an algorithm and implementation of an automatic tree walker generator. This chapter will evaluate the provided solution.

7.1 Summary

An ANTLR parser specification not only contains the (E)BNF grammar definition of the language to be parsed, it also contains the description of the abstract syntax tree to be generated in the form of tree shaping operators. It therefore makes sense to automatically derive the parsers for these abstract syntax trees from the same specification. This thesis introduced ANTLR$_{TG}$, a tool that can generate ANTLR tree walkers given an annotated ANTLR parser specification. It uses a powerful algorithm by means of a tree pattern algebra. Several ANTLR$_{TG}$ specific annotations allow the integrated parser specification to contain all necessary information for the parser and all tree walkers. The usefulness of this tool was demonstrated with a case study building a Triangle compiler, which showed to significantly reduce the identified issues.

7.2 Evaluation of ANTLR$_{TG}$

Automatically generating a tree walker for a specific parser helps developing a compiler significantly, as it removes the problems of debugging AST mismatches. The maintainability of the compiler code is improved by an enhancement of the ANTLR action annotations that allows the developer to place all action annotations for all tree walkers in the integrated parser specification. Although the need for a specialised editor to help the user correctly position these action annotations was foreseen, after completion of the case study it became apparent that manually positioning these action annotations was relatively easy in practise. The case study has shown that this resulted in significantly more freedom to iteratively improve the parser specification without the risk of accidentally invalidating one of the tree walkers compared to the normal ANTLR work flow.

ANTLR version 3 provides a rewriting notation that enables the developer to manually structure the abstract syntax tree. This diminishes the need to know the exact details of the tree shaping operators, and therefore reduces a part of the problem described in the first chapter of this thesis. However it introduces a new redundancy and it does not solve the redundancy between the parser specification and the tree walker specifications, which introduces the most significant problems.

ANTLR$_{TG}$ relies on the usage of tree shaping operators of ANTLR to shape the abstract syntax tree, these operators can be hard to understand for new users and the complexity of these operators has been identified as part of the problem described in the first chapter of this thesis. However, ANTLR$_{TG}$ is able to provide the user with direct feedback as the user can immediately generate a tree walker specification and evaluate the effects of the used tree shaping operators on the tree walker.

7.3 Future work

The algorithms introduced in chapters 3 and 4 have shown to provide a useful tree walker. The case-study has shown that integrating all grammar components into a single specification can be done and remains maintainable, at least for small compilers. It can be expected that for larger compilers, the integrated parser specification will grow significantly. This might reduce the maintainability, several possible solutions for this problem have been identified:

- A very simple solution would be to allow the user to split up the integrated parser specification into several files using `#include` directives, similar to the C pre-processor. This allows the user to split up the grammar into logical sections, for example declarations, commands and expressions. Although this would still require that the code for the different tree walkers are provided in the same file, which might become confusing if there are many tree walkers.
- Another solution would be to automatically generate a tree walker that invokes a method on an interface for each visited terminal, according to the visitor pattern [15]. This allows the user to directly write the code for the tree walker in a Java class, instead of adding action annotations to the integrated parser specification. This approach is already more or less visible in the case-study, where most code for the generator has been placed in a separate class.
- A more advanced solution would be a dedicated editor that provides the user with multiple views of the integrated parser specification. This editor will show the user the parser specification with only the action annotations for the parser when the user works on the parser specification, hiding all other information. When the user works on one of the tree walkers instead, the editor will generate and show the requested tree walker. The user can then change the action annotations in this view, which the editor will then merge back into the integrated parser specification transparently to the user.

Another topic for future work might be to investigate integrating ANTLR$_{TG}$ into ANTLRWorks [18]. Several new features can be provided:

- Automatically generating a skeleton tree walker from an ANTLR parser specification. For example, when the user chooses to create a new tree walker, ANTLRWorks could automatically provide the user with an automatically generated tree walker based on the parser specification that was already made. Although it would probably not be possible to keep the two synchronised after changes have been made to both.

- A more thorough approach would be to integrate the editor mentioned above. This would probably be significantly more work, since now also the integrated debugger of ANTLR-Works needs to be taken into account, but it could become a very powerful combination.

Finally, it might be worth to investigate what ANTLR$_{TG}$ can bring to JavaCC [22] [23]. JavaCC is at certain points very similar to ANTLR and therefore a JavaCC$_{TG}$ may be useful.

Appendix A

ANTLR$_{TG}$ sources

This appendix contains the most significant source files for ANTLR$_{TG}$. Explanations on the software design is provided in chapter 5.

A.1 FCNSTree class

```
////////////////////////////////////////////////////////////////////////////////
// ANTLR tree parser generator                                                //
// Copyright (C) 2010 A.J. Admiraal (mailto:code@admiraal.dds.nl)            //
//                                                                            //
// This program is free software; you can redistribute it and/or modify it    //
// under the terms of the GNU General Public License version 2 as published   //
// by the Free Software Foundation.                                           //
//                                                                            //
// This program is distributed in the hope that it will be useful, but        //
// WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY //
// or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License    //
// for more details.                                                          //
//                                                                            //
// You should have received a copy of the GNU General Public License along    //
// with this program; if not, write to the Free Software Foundation, Inc., 51 //
// Franklin Street, Fifth Floor, Boston, MA 02110-1301, USA.                  //
////////////////////////////////////////////////////////////////////////////////

package org.antlrtg.algo;

import java.lang.String;
import java.util.Collection;
import java.util.HashMap;
import java.util.Vector;
import org.antlrtg.utilities.Payload;
import org.antlrtg.utilities.GenericPayload;

/** The implementation of the FCNS tree
    @author <a href="mailto:code@admiraal.dds.nl">A.J. Admiraal</a>
```

```
 */
public class FCNSTree
{
  public enum NodeType { NODE, ROOTCONNECTOR, CHILDCONNECTOR }
  public enum Tag { LABEL, ACTIVE_CHIDREN, INACTIVE_CHIDREN }


  /** A FCNS tree node
      @author <a href="mailto:code@admiraal.dds.nl">A.J. Admiraal</a>
   */
  public static class Node
  {
    // Attributes
    private Payload payload;
    private NodeType type;
    private Node parent;
    private Node firstChild;
    private Node nextSibling;

    // Simple functions
    public Payload getPayload() { return payload; }
    public NodeType getType() { return type; }
    public void setType(NodeType t) { type = t; }
    public Node getParent() { return parent; }
    public Node getFirstChild() { return firstChild; }
    public Node getNextSibling() { return nextSibling; }

    /** Constructs a new Node.
        @param payload          The payload for the node.
        @param type             The type of the node.
        @param firstChild       The first child of the node.
        @param nextSibling      The next sibling of the node.
     */
    public Node(Payload payload,
                NodeType type,
                Node firstChild,
                Node nextSibling)
    {
      this.payload = payload;
      this.type = type;
      this.parent = null;
      setFirstChild(firstChild);
      setNextSibling(nextSibling);
    }

    /** Returns a deep copy of the node.
     */
    protected Node clone()
    {
      Node fc = (firstChild != null) ? firstChild.clone() : null;
      Node ns = (nextSibling != null) ? nextSibling.clone() : null;

      return new Node(payload, type, fc, ns);
```

```
}

/** Sets the first child of this node
    @param n                The new first child.
 */
public void setFirstChild(Node n)
{
  firstChild = n;

  if (firstChild != null)
    firstChild.setParent(this);
}

/** Sets the next sibling of this node
    @param n                The new next sibling.
 */
public void setNextSibling(Node n)
{
  nextSibling = n;

  if (nextSibling != null)
    nextSibling.setParent(this);
}

/** Returns either the previous sibling of this node, or null if it has no
    previous sibling.
 */
public Node getPreviousSibling()
{
  if (parent != null)
  if (parent.getNextSibling() == this)
    return parent;

  return null;
}

/** Returns the previous last sibling of this node.
 */
public Node getLastSibling()
{
  Node l = this;
  while (l.getNextSibling() != null)
    l = l.getNextSibling();

  return l;
}

/** Returns a collection of all siblings of this node, including this node.
 */
public Collection<Node> getAllSiblings()
{
  Vector<Node> nodes = new Vector<Node>();
```

```
    for (Node i=this; i!=null; i=i.getNextSibling())
      nodes.add(i);

    for (Node i=getPreviousSibling(); i!=null; i=i.getPreviousSibling())
      nodes.add(i);

    return nodes;
  }

  public String toString()
  {
    if (type == NodeType.ROOTCONNECTOR)
      return "âŮŔ";
    else if (type == NodeType.CHILDCONNECTOR)
      return "âŮŃ";
    else if (payload != null)
    {
      if (payload.isOptional())
        return payload.getLabel() + "?";
      else if (payload.isRepetetive())
        return payload.getLabel() + "*";
      else
        return payload.getLabel();
    }
    else
      return null;
  }

  public boolean equals(Object o)
  {
    return toString().equals(o.toString());
  }

  private String toLinear()
  {
    String linear;
    Node fc = getFirstChild();
    Node ns = getNextSibling();

    if (fc != null)
      linear = "^( " + toString() + " " + fc.toLinear() + ") ";
    else
      linear = toString() + " ";

    if (ns != null)
      linear += ns.toLinear();

    return linear;
  }

  protected void setParent(Node n)
  {
    if (parent != null)
```

```
        parent.remove(this);

      parent = n;
    }

    protected void remove(Node n)
    {
      if (firstChild == n)
        firstChild = null;
      if (nextSibling == n)
        nextSibling = null;
    }
  }

  // Attributes
  private Node root;
  private HashMap<Tag, Object> tags;
  private boolean repetetive;

  // Simple functions
  public Node getRoot() { return root; }
  public Object getTag(Tag id) { return tags.get(id); }
  public void setTag(Tag id, Object tag) { tags.put(id, tag); }
  public boolean getRepetetive() { return repetetive; }
  public void setRepetetive(boolean r) { repetetive = r; }

  /** Constructs a new FCNS tree with a specified root node.
      @param root               The root node.
   */
  protected FCNSTree(Node root)
  {
    this.root = root;
    this.repetetive = false;
    tags = new HashMap<Tag, Object>();
  }

  /** Reconstructs a new FCNS tree from a linear representation.
      @param linear             The linear representation of the FCNS tree.
   */
  public FCNSTree(String linear)
  {
    // Parameter by reference (pointer)
int[] parsePos = new int[1];
    parsePos[0] = 0;

    this.root = parseLinear(linear.split(" "), parsePos, false);
    this.repetetive = false;
  }

  /** Returns a deep copy of the FCNS tree.
   */
  public FCNSTree clone()
  {
```

```
  if (root != null)
    return new FCNSTree(root.clone());
  else
    return new FCNSTree((Node)null);
}

public String toString()
{
  Node root = getRoot();

  if (root != null)
  {
    if (repetetive)
      return "(" + root.toLinear() + ")+";
    else
      return root.toLinear();
  }
  else
    return "";
}

public boolean equals(Object o)
{
  return toString().equals(o.toString());
}

/** Sets a new root node.
    @param node               The new root node.
 */
public void setRoot(Node n)
{
  root = n;

  if (root != null)
    root.setParent(null);
}

/** Returns the depth of the FCNS tree.
 */
public int getDepth()
{
  int depth = 0;
  for (Node i=getRoot(); i!=null; i=i.getFirstChild())
    depth++;

  return depth;
}

/** Parses a linear FCNS representation into a (root) node.
    @param linear             The linear FCNS representation.
    @param parsePos           The current position in the linear FCNS
                              representation.
    @param firstRoot          True if this is the first entry in a by
```

```
                                parentheses enclosed sequence, where all other
                                entries are children of this one.
   */
  protected Node parseLinear(String[] linear, int[] parsePos, boolean firstRoot)
  {
    Node n = null;
    Node nx = null;

    for (; parsePos[0]<linear.length; parsePos[0]++)
    {
      Node nt = null;

      if (linear[parsePos[0]].equals("^("))
      {
        parsePos[0]++;
        nt = parseLinear(linear, parsePos, true);
      }
      else if (linear[parsePos[0]].equals(")"))
        return n;
      else
        nt = new Node(new GenericPayload<String>(linear[parsePos[0]]),
                      NodeType.NODE,
                      null,
                      null);

      if (nx == null)
        n = nx = nt;
      else if (firstRoot == true)
      {
        nx.setFirstChild(nt);
        nx = nt;
        firstRoot = false;
      }
      else
      {
        nx.setNextSibling(nt);
        nx = nt;
      }
    }

    return n;
  }
}
```

A.2 TreePattern class

```
////////////////////////////////////////////////////////////////////////////////
// ANTLR tree parser generator                                                //
// Copyright (C) 2010 A.J. Admiraal (mailto:code@admiraal.dds.nl)             //
//                                                                            //
// This program is free software; you can redistribute it and/or modify it    //
// under the terms of the GNU General Public License version 2 as published   //
// by the Free Software Foundation.                                         //
//                                                                          //
// This program is distributed in the hope that it will be useful, but      //
// WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY //
// or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License  //
// for more details.                                                        //
//                                                                          //
// You should have received a copy of the GNU General Public License along  //
// with this program; if not, write to the Free Software Foundation, Inc., 51 //
// Franklin Street, Fifth Floor, Boston, MA 02110-1301, USA.                //
////////////////////////////////////////////////////////////////////////////////

package org.antlrtg.algo;

import java.util.Collection;
import java.util.LinkedList;
import org.antlrtg.utilities.Payload;

/** The implementation of the Tree pattern
    @author <a href="mailto:code@admiraal.dds.nl">A.J. Admiraal</a>
 */
public class TreePattern extends FCNSTree
{
  private static TreePattern    nullPattern = null;

  /** Returns the null tree pattern (i.e. "^( âŮŔ âŮŃ )")
    */
  public static TreePattern GetNullPattern()
  {
    if (nullPattern == null)
      nullPattern = new TreePattern();

    return nullPattern;
  }

  private TreePattern()
  {
    super((Node)null);

    Node cc = new Node(null, NodeType.CHILDCONNECTOR, null, null);
    setRoot(new Node(null, NodeType.ROOTCONNECTOR, cc, null));
  }

  /** Constructs a new elementary tree pattern.
      @param payload            The payload for the node in the elementary tree
                                pattern.
      @param active             True if this will be the active elementary tree
                                pattern (i.e. "^( Îś ^( âŮŔ âŮŃ ) )"), otherwise it
                                will be the inactive elementary tree pattern
                                (i.e. "^( âŮŔ âŮŃ Îś )").
   */
  public TreePattern(Payload payload, boolean active)
  {
    super((Node)null);

    Node cc = new Node(null, NodeType.CHILDCONNECTOR, null, null);
    Node rc = new Node(null, NodeType.ROOTCONNECTOR, cc, null);

    if (active)
      setRoot(new Node(payload, NodeType.NODE, rc, null));
    else
    {
      cc.setNextSibling(new Node(payload, NodeType.NODE, rc, null));
      setRoot(rc);
    }
  }

  /** Constructs a new elementary tree pattern.
      @param node               The node in the elementary tree pattern.
      @param active             True if this will be the active elementary tree
                                pattern ("^( A ^( âŮŔ âŮŃ ) )"), otherwise it will
                                be the inactive elementary tree pattern
                                ("^( âŮŔ âŮŃ A )").
   */
  public TreePattern(Node node, boolean active)
  {
    super((Node)null);

    Node cc = new Node(null, NodeType.CHILDCONNECTOR, null, null);
    Node rc = new Node(null, NodeType.ROOTCONNECTOR, cc, null);

    if (active)
    {
      setRoot(node);
      node.setFirstChild(rc);
    }
    else
    {
      cc.setNextSibling(node);
      setRoot(rc);
    }
  }

  /** Constructs a new tree pattern with the specified node as root, note that
      this constructor can not guarantee that it is a real tree pattern (i.e. it
      has a root and child connector).
      @param root               The root in the tree pattern.
   */
  protected TreePattern(Node root)
  {
    super(root);
  }

  /** Constructs a new tree pattern from a linear representation, note that
      this constructor can not guarantee that it is a real tree pattern (i.e. it
      has a root and child connector).
    @param linear            The linear representation of the tree pattern.
 */
public TreePattern(String linear)
{
  super(linear);

  // Mark the root and child connector appropriately
  for (Node i=getRoot(); i!=null; i=i.getFirstChild())
  if (i.toString().equals("âŮŔ"))
    i.setType(NodeType.ROOTCONNECTOR);
  else if (i.toString().equals("âŮŃ"))
    i.setType(NodeType.CHILDCONNECTOR);
}

/** Returns a deep copy of the tree pattern.
 */
public TreePattern clone()
{
  return new TreePattern(getRoot().clone());
}

/** Returns true if this is an active pattern.
 */
public boolean isActive()
{
  if (getRoot().getType() == NodeType.ROOTCONNECTOR)
    // The root connector is the root, the pattern is only active iff the
    // child connector and the root connector have no siblings
    return (getRoot().getNextSibling() == null) &&
           (getRoot().getFirstChild().getNextSibling() == null);
  else
    // The root connector is not the root, hence the pattern is active
    return true;
}

/** Returns true if this is the null pattern (i.e. "^( âŮŔ âŮŃ )").
 */
public boolean isNull()
{
  if (getRoot().getType() == NodeType.ROOTCONNECTOR)
    // The root connector is the root, the pattern is null iff the
    // child connector and the root connector have no siblings
    return (getRoot().getNextSibling() == null) &&
           (getRoot().getFirstChild().getNextSibling() == null);
  else
    // The root connector is not the root, hence the pattern is not null
    return false;
}

/** Returns the first node that is not a root or child connector in the
 *  pattern.
 */
public Node getFirstNode()
```

```
{
  Node node = getRoot();

  while (node.getType() != NodeType.NODE)
  {
    if (node.getType() == NodeType.ROOTCONNECTOR)
      node = node.getFirstChild();
    else if (node.getType() == NodeType.CHILDCONNECTOR)
      node = node.getNextSibling();
  }

  return node;
}

/** Concatenates a pattern with this pattern and returns the result.
    @param addPattern         The pattern to concatenate to this pattern
                              (i.e. this Âů addPattern).
 */
public TreePattern concat(TreePattern addPattern)
{
  // Make sure we're not modifying existing patterns.
  TreePattern con = addPattern.clone();
  Node root = getRoot().clone();

  // Find the root connector in the addPattern
  for (Node i=con.getRoot(); i!=null; i=i.getFirstChild())
  if (i.getType() == NodeType.ROOTCONNECTOR)
  {
    // Retrieve the parent, child connector and next sibling
    Node p = i.getParent();
    Node c = i.getFirstChild().getNextSibling();
    Node n = i.getNextSibling();

    // Replace the root connector with the child pattern
    if (p != null)
      p.setFirstChild(root);
    else
      con.setRoot(root);

    // Add the next sibling of the root connector to the last sibling of
    // the root of the child pattern
    root.getLastSibling().setNextSibling(n);

    // Add the next sibling of the child connector to the last sibling of
    // the first child of the root of the child pattern
    root.getFirstChild().getLastSibling().setNextSibling(c);

    // Return the result
    return con;
  }

  // Failed (corrupted pattern?)
  return null;
}

/** Returns the top of this pattern.
 */
public TreePattern getTop()
{
  // Make sure we're not modifying existing patterns.
  TreePattern top = clone();

  // Find the root connector in the addPattern
  for (Node i=top.getRoot(); i!=null; i=i.getFirstChild())
  if (i.getType() == NodeType.CHILDCONNECTOR)
  {
    // Remove all siblings of the child connector
    i.setNextSibling(null);

    // Return the result
    return top;
  }

  // Failed (corrupted pattern?)
  return null;
}

/** Returns the bottom of this pattern.
 */
public TreePattern getBottom()
{
  // Make sure we're not modifying existing patterns.
  TreePattern bot = clone();

  // Find the root connector in the addPattern
  for (Node i=bot.getRoot(); i!=null; i=i.getFirstChild())
  if (i.getType() == NodeType.ROOTCONNECTOR)
  {
    // Remove all siblings of the root connector
    i.setNextSibling(null);

    // Make the root connector the root of the patern
    bot.setRoot(i);

    // Return the result
    return bot;
  }

  // Failed (corrupted pattern?)
  return null;
}

/** Collapses this pattern into an FCNS tree (i.e. removes the root and child
    connectors).
 */
public FCNSTree collapse()
```

```
{
  // Make sure we're not modifying existing patterns.
  TreePattern col = clone();

  // Find the root connector in the addPattern
  for (Node i=col.getRoot(); i!=null; i=i.getFirstChild())
  if (i.getType() == NodeType.ROOTCONNECTOR)
  {
    // Retrieve the parent, child connector and next sibling
    Node p = i.getParent();
    Node c = i.getFirstChild().getNextSibling();
    Node n = i.getNextSibling();

    // Replace the root connector with the siblings of the child connector
    if (p != null)
      p.setFirstChild(c);
    else
      col.setRoot(c);

    // Add the siblings of the root connector to the siblings of the child
    // connector
    if (c != null)
      c.getLastSibling().setNextSibling(n);

    // Return the result
    return new FCNSTree(col.getRoot());
  }

  // Failed (corrupted pattern?)
  System.err.println("Found corrupted pattern: " + toString());
  return null;
}

/** Decomposes the pattern into a collection of elementary tree patterns.
 */
public Collection<TreePattern> decompose()
{
  // Make sure we're not modifying existing patterns.
  TreePattern cln = clone();

  // Find the root connector in the addPattern
  for (Node i=cln.getRoot(); i!=null; i=i.getFirstChild())
  if (i.getType() == NodeType.ROOTCONNECTOR)
  {
    // Retrieve the parent, child connector
    Node p = i.getParent();
    Node c = i.getFirstChild().getNextSibling();

    if (c != null)
    {
      // Remove it from the pattern
      c.getParent().setNextSibling(c.getNextSibling());
      c.setNextSibling(null);
```

```
        c.setFirstChild(null);

        // Add it to the list
        LinkedList<TreePattern> dec = (LinkedList<TreePattern>)cln.decompose();
        dec.add(0, new TreePattern(c, false));

        return dec;
      }
      else if (p != null)
      {
        // Remove it from the pattern
        i.getFirstChild().setNextSibling(i.getNextSibling());
        i.setNextSibling(p.getNextSibling());
        p.setNextSibling(null);
        p.setFirstChild(null);

        if (p.getParent() != null)
          p.getParent().setFirstChild(i);
        else
          cln.setRoot(i);

        // Add it to the list
        LinkedList<TreePattern> dec = (LinkedList<TreePattern>)cln.decompose();
        dec.add(0, new TreePattern(p, true));

        return dec;
      }
    }

    // Empty pattern, return empty list
    return new LinkedList<TreePattern>();
  }
}
```

A.3 ANTLR$_{TG}$ grammar

```
/*
 [The "BSD licence"]
 Copyright (c) 2005-2007 Terence Parr
 ANTLRTG modifications - Copyright (c) 2010 A.J. Admiraal
 All rights reserved.

 Redistribution and use in source and binary forms, with or without
 modification, are permitted provided that the following conditions
 are met:
 1. Redistributions of source code must retain the above copyright
    notice, this list of conditions and the following disclaimer.
 2. Redistributions in binary form must reproduce the above copyright
    notice, this list of conditions and the following disclaimer in the
    documentation and/or other materials provided with the distribution.
 3. The name of the author may not be used to endorse or promote products
    derived from this software without specific prior written permission.
 THIS SOFTWARE IS PROVIDED BY THE AUTHOR ``AS IS'' AND ANY EXPRESS OR
 IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES
 OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED.
 IN NO EVENT SHALL THE AUTHOR BE LIABLE FOR ANY DIRECT, INDIRECT,
 INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT
 NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE,
 DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY
 THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
 (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF
 THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
*/

/** ANTLR v3 grammar written in ANTLR v3 with AST construction */
grammar ANTLRv3;

options {
  output=AST;
  ASTLabelType=CommonTree;
}

tokens {
  DOC_COMMENT;
  PARSER;
  LEXER;
  RULE;
  BLOCK;
  OPTIONAL;
  CLOSURE;
  POSITIVE_CLOSURE;
  SYNPRED;
  RANGE;
  CHAR_RANGE;
  EPSILON;
  ALT;
  EOR;
  EOB;
  EOA; // end of alt
  ID;
  ARG;
  ARGLIST;
  RET='returns';
  LEXER_GRAMMAR;
  PARSER_GRAMMAR;
  TREE_GRAMMAR;
  COMBINED_GRAMMAR;
  LABEL; // $x used in rewrite rules
  TEMPLATE;
  SCOPE='scope';
  SEMPRED;
  GATED_SEMPRED; // {p}? =>
  SYN_SEMPRED; //(...) => it's a manually-specified synpred converted to sempred
  BACKTRACK_SEMPRED; // auto backtracking mode syn pred converted to sempred
  FRAGMENT='fragment';
  TREE_BEGIN='^(';
  ROOT='^';
  BANG='!';
  RANGE='..';
  REWRITE='->';
  AT='@';
  LABEL_ASSIGN='=';
  LIST_LABEL_ASSIGN='+=';
}

@parser::header
{
  package org.antlrtg.grammar;
}

@lexer::header
{
  package org.antlrtg.grammar;
}

@members {
  int gtype;
  private GrammarGenerator gen;
  public void setGenerator(GrammarGenerator g) { gen = g; }
}

grammarDef
    :   DOC_COMMENT?
      ( 'lexer'  { gtype=LEXER_GRAMMAR; gen.setEnabled(false); }
      | 'parser' { gtype=PARSER_GRAMMAR; gen.setEnabled(true); }
      | 'tree'   { gtype=TREE_GRAMMAR; gen.setEnabled(false); }
      |          { gtype=COMBINED_GRAMMAR; gen.setEnabled(true); }
      )
      g='grammar' id ';' { gen.setParserName($id.text); } optionsSpec?
      tokensSpec? attrScope* action*
      rule+
      EOF
      -> ^( {adaptor.create(gtype,$g)}
          id DOC_COMMENT? optionsSpec? tokensSpec? attrScope* action* rule+
         )
    ;

tokensSpec
@init{ gen.genParserTokensSpec(true); }
@after{ gen.genParserTokensSpec(false); }
  : TOKENS tokenSpec+ '}' -> ^(TOKENS tokenSpec+)
  ;

tokenSpec
  : t1=TOKEN_REF
    ( '=' (lit=STRING_LITERAL|lit=CHAR_LITERAL)
    { gen.genParserTokensSpec($t1.text, $lit.text); } -> ^('=' TOKEN_REF $lit)
    | { gen.genParserTokensSpec($t1.text, null); } -> TOKEN_REF
    )
    ';'
  ;

attrScope
  : 'scope' id a=ACTION { gen.appendAction($id.text, $a); }
    -> ^('scope' id ACTION)
  ;

/** Match stuff like @parser::members {int i;} */
action
  : '@' (actionScopeName
    { gen.setActionScopeName($actionScopeName.text); }
    '::')? id a=ACTION { gen.appendAction($id.text, $a); }
    -> ^('@' actionScopeName? id ACTION)
  ;

/** Sometimes the scope names will collide with keywords; allow them as
 *  ids for action scopes.
 */
actionScopeName
  : id
  | l='lexer' -> ID[$l]
    |   p='parser' -> ID[$p]
  ;

optionsSpec
//@init{ gen.genParserOptionsSpec(true); }
//@after{ gen.genParserOptionsSpec(false); }
  : OPTIONS (option ';')+ '}' -> ^(OPTIONS option+)
  ;

option
  : id '=' optionValue
    { gen.genParserOptionsSpec($id.text, $optionValue.text); }
    -> ^('=' id optionValue)
  ;

optionValue
  : qid
  | STRING_LITERAL
  | CHAR_LITERAL
  | INT
  | s='*' -> STRING_LITERAL[$s]  // used for k=*
  ;

rule
scope {
  String name;
}   @after{ gen.closeRule(); }
  : DOC_COMMENT?
    ( modifier=('protected'|'public'|'private'|'fragment') )?
    id {$rule::name = $id.text; gen.setRuleName($id.text);}
    ('^' { gen.setPrimaryRule(); } )? /* ANTLRTG grammar supplement */
    '!'?
    ( arg=ARG_ACTION )?
    ( 'returns' rt=ARG_ACTION { gen.setRuleReturns($rt); } )?
    throwsSpec? optionsSpec? ruleScopeSpec? ruleAction*
    ':' altList ';'
    exceptionGroup?
      -> ^( RULE id {modifier!=null?adaptor.create(modifier):null}
           ^(ARG[$arg] $arg)? ^('returns' $rt)?
           throwsSpec? optionsSpec? ruleScopeSpec? ruleAction*
           altList
           exceptionGroup?
           EOR["EOR"]
         )
  ;

/** Match stuff like @init {int i;} */
ruleAction
  : '@' id a=ACTION { gen.appendAction($id.text, $a); } -> ^('@' id ACTION)
  ;

throwsSpec
  : 'throws' id ( ',' id )* -> ^('throws' id+)
  ;

ruleScopeSpec
  : 'scope' a=ACTION { gen.appendAction($a); } -> ^('scope' ACTION)
  | 'scope' id (',' id)* ';' -> ^('scope' id+)
  | 'scope' a=ACTION { gen.appendAction($a); }
    'scope' id (',' id)* ';'
    -> ^('scope' ACTION id+ )
  ;

block
  : lp='('
    ( (opts=optionsSpec)? ':' )?
    altpair ( '|' { gen.nextAlternative(); } altpair )*
    rp=')'
    -> ^( BLOCK[$lp,"BLOCK"] optionsSpec? altpair+ EOB[$rp,"EOB"] )
  ;

altpair : alternative rewrite ;

altList
@init {
// must create root manually as it's used by invoked rules in real antlr tool.
// leave here to demonstrate use of {...} in rewrite rule
// it's really BLOCK[firstToken,"BLOCK"]; set line/col to previous ( or : token.
  CommonTree blkRoot = (CommonTree)adaptor.create(BLOCK,input.LT(-1),"BLOCK");
}
  : altpair ( '|' { gen.nextAlternative(); } altpair )*
    -> ^( {blkRoot} altpair+ EOB["EOB"] )
  ;

alternative
@init {
  Token firstToken = input.LT(1);
  Token prevToken = input.LT(-1); // either : or | I think
}
  : element+ -> ^(ALT[firstToken,"ALT"] element+ EOA["EOA"])
  | -> ^(ALT[prevToken,"ALT"] EPSILON[prevToken,"EPSILON"] EOA["EOA"])
  ;

exceptionGroup
  : ( exceptionHandler )+ ( finallyClause )?
  | finallyClause
  ;

exceptionHandler
  : 'catch' ARG_ACTION ACTION -> ^('catch' ARG_ACTION ACTION)
  ;

finallyClause
  : 'finally' ACTION -> ^('finally' ACTION)
  ;

element
  : id (labelOp='='|labelOp='+=')
    { gen.openScope(); gen.setRuleAssignID($id.text); }
    atom
    ( ebnfSuffix
      -> ^( ebnfSuffix ^(BLOCK["BLOCK"] ^(ALT["ALT"]
            ^($labelOp id atom) EOA["EOA"]) EOB["EOB"]))
    | { gen.cancelScope(); }   -> ^($labelOp id atom)
    )
  | id (labelOp='='|labelOp='+=')
    { gen.openScope(); gen.setRuleAssignID($id.text); }
    block
    ( ebnfSuffix
      -> ^( ebnfSuffix ^(BLOCK["BLOCK"] ^(ALT["ALT"]
            ^($labelOp id block) EOA["EOA"]) EOB["EOB"]))
    | { gen.cancelScope(); }   -> ^($labelOp id block)
    )
  | { gen.openScope(); } atom
    ( ebnfSuffix
      -> ^( ebnfSuffix ^(BLOCK["BLOCK"]
            ^(ALT["ALT"] atom EOA["EOA"]) EOB["EOB"]) )
    | { gen.cancelScope(); }   -> atom
    )
  | ebnf
  |   a=ACTION { gen.appendAction($a); }
  |   SEMPRED ( g='=>' -> GATED_SEMPRED[$g] | -> SEMPRED )
  |   { gen.openScope(); } treeSpec
    ( ebnfSuffix
      -> ^( ebnfSuffix ^(BLOCK["BLOCK"]
            ^(ALT["ALT"] treeSpec EOA["EOA"]) EOB["EOB"]) )
    | { gen.cancelScope(); }   -> treeSpec
    )
  ;

atom @init{ int flags = (int)(' '); }
  : terminal
  | range
    ( (  op='^' { flags |= GrammarGenerator.CARET_ENABLED; }
      |  op='!' { flags |= GrammarGenerator.BANG_ENABLED; }) -> ^($op range)
    |     -> range
    ) { gen.parseTerminal($range.text, flags); }
  | notSet
    ( (op='^'|op='!') -> ^($op notSet)
    |     -> notSet
    )
  |   rr=RULE_REF (aa=ARG_ACTION { gen.setArgAction(aa); } )?
    ( (  op='^' { flags |= GrammarGenerator.CARET_ENABLED; }
      |  op='!' { flags |= GrammarGenerator.BANG_ENABLED; } )
         -> ^($op RULE_REF ARG_ACTION?)
    |     -> ^(RULE_REF ARG_ACTION?)
    ) { gen.parseRuleRef(rr, flags); }
  ;

notSet
  : '~'
    ( notTerminal elementOptions? -> ^('~' notTerminal elementOptions?)
    | block elementOptions?  -> ^('~' block elementOptions?)
    )
  ;

notTerminal
  :   CHAR_LITERAL
  | TOKEN_REF
  | STRING_LITERAL
  ;

elementOptions
  : '<' qid '>'      -> ^(OPTIONS qid)
  | '<' option (';' option)* '>' -> ^(OPTIONS option+)
  ;

elementOption
  : id '=' optionValue -> ^('=' id optionValue)
  ;

treeSpec
  : '^(' element ( element )+ ')' -> ^(TREE_BEGIN element+)
  ;

range!
  : c1=CHAR_LITERAL RANGE c2=CHAR_LITERAL elementOptions?
    -> ^(CHAR_RANGE[$c1,".."] $c1 $c2 elementOptions?)
  ;

terminal @init { int flags = (int)(' '); }
  : ( t=CHAR_LITERAL elementOptions? -> ^(CHAR_LITERAL elementOptions?)
      // Args are only valid for lexer rules
    | t=TOKEN_REF (aa=ARG_ACTION  { gen.setArgAction(aa); })? elementOptions?
      -> ^(TOKEN_REF ARG_ACTION? elementOptions?)
    | t=STRING_LITERAL elementOptions?    -> ^(STRING_LITERAL elementOptions?)
    | '.' elementOptions?        -> ^('.' elementOptions?)
    )
    ( '^' { flags |= GrammarGenerator.CARET_ENABLED; }     -> ^('^' $terminal)
    | '!' { flags |= GrammarGenerator.BANG_ENABLED; }      -> ^('!' $terminal)
    )?
    { gen.parseTerminal(t, flags); }
  ;

/** Matches ENBF blocks (and token sets via block rule) */
ebnf
@init {
  Token firstToken = input.LT(1);
  int flags = (int)(' ');
  gen.openScope();
}
@after {
  $ebnf.tree.getToken().setLine(firstToken.getLine());
  $ebnf.tree.getToken().setCharPositionInLine(
    firstToken.getCharPositionInLine());
  gen.closeScope(flags);
}
  : block
    ( op='?' { flags = (flags & ~GrammarGenerator.SCOPECHAR) | (int)('?'); }
      -> ^(OPTIONAL[op] block)
    | op='*' { flags = (flags & ~GrammarGenerator.SCOPECHAR) | (int)('*'); }
      -> ^(CLOSURE[op] block)
    | op='+' { flags = (flags & ~GrammarGenerator.SCOPECHAR) | (int)('+'); }
      -> ^(POSITIVE_CLOSURE[op] block)
    | '=>' // syntactic predicate
      -> {gtype==COMBINED_GRAMMAR &&
         Character.isUpperCase($rule::name.charAt(0))}?
         // if lexer rule in combined, leave as pred for lexer
         ^(SYNPRED["=>"] block)
         // in real antlr tool, text for SYN_SEMPRED is predname
        -> SYN_SEMPRED
    |   -> block
    )
  ;

ebnfSuffix
@init {
  Token op = input.LT(1);
    int flags = (int)(' ');
}
@after {
    gen.closeScope(flags);
}
  : '?' { flags = (flags & ~GrammarGenerator.SCOPECHAR) | (int)('?'); }
    -> OPTIONAL[op]
  | '*' { flags = (flags & ~GrammarGenerator.SCOPECHAR) | (int)('*'); }
    -> CLOSURE[op]
  | '+' { flags = (flags & ~GrammarGenerator.SCOPECHAR) | (int)('+'); }
    -> POSITIVE_CLOSURE[op]
  ;


// R E W R I T E   S Y N T A X

rewrite
@init {
  Token firstToken = input.LT(1);
}
  : (rew+='->' preds+=SEMPRED predicated+=rewrite_alternative)*
    rew2='->' last=rewrite_alternative
        -> ^($rew $preds $predicated)* ^($rew2 $last)
  |
  ;

rewrite_alternative
options {backtrack=true;}
  : rewrite_template
  | rewrite_tree_alternative
    |   /* empty rewrite */ -> ^(ALT["ALT"] EPSILON["EPSILON"] EOA["EOA"])
  ;

rewrite_tree_block
  : lp='(' rewrite_tree_alternative ')'
    -> ^(BLOCK[$lp,"BLOCK"] rewrite_tree_alternative EOB[$lp,"EOB"])
  ;

rewrite_tree_alternative
  : rewrite_tree_element+ -> ^(ALT["ALT"] rewrite_tree_element+ EOA["EOA"])
  ;

rewrite_tree_element
  : rewrite_tree_atom
  | rewrite_tree_atom ebnfSuffix
    -> ^( ebnfSuffix ^(BLOCK["BLOCK"]
          ^(ALT["ALT"] rewrite_tree_atom EOA["EOA"]) EOB["EOB"]))
  |   rewrite_tree
    ( ebnfSuffix
      -> ^(ebnfSuffix ^(BLOCK["BLOCK"]
           ^(ALT["ALT"] rewrite_tree EOA["EOA"]) EOB["EOB"]))
    | -> rewrite_tree
    )
  |   rewrite_tree_ebnf
  ;
```

```
rewrite_tree_atom
  : CHAR_LITERAL
  | TOKEN_REF ARG_ACTION? -> ^(TOKEN_REF ARG_ACTION?) // for imaginary nodes
  | RULE_REF
  | STRING_LITERAL
  | d='$' id -> LABEL[$d,$id.text] // reference to a label in a rewrite rule
  | ACTION
  ;

rewrite_tree_ebnf
@init {
  Token firstToken = input.LT(1);
}
@after {
  $rewrite_tree_ebnf.tree.getToken().setLine(firstToken.getLine());
  $rewrite_tree_ebnf.tree.getToken().setCharPositionInLine(
    firstToken.getCharPositionInLine());
}
  : rewrite_tree_block ebnfSuffix -> ^(ebnfSuffix rewrite_tree_block)
  ;

rewrite_tree
  : '^(' rewrite_tree_atom rewrite_tree_element* ')'
    -> ^(TREE_BEGIN rewrite_tree_atom rewrite_tree_element* )
  ;

/** Build a tree for a template rewrite:
      ^(TEMPLATE (ID|ACTION) ^(ARGLIST ^(ARG ID ACTION) ...) )
    where ARGLIST is always there even if no args exist.
    ID can be "template" keyword.  If first child is ACTION then it's
    an indirect template ref

    -> foo(a={...}, b={...})
    -> ({string-e})(a={...}, b={...})  // e evaluates to template name
    -> {%{$ID.text}} // create literal template from string
    -> {st-expr} // st-expr evaluates to ST
 */
rewrite_template
  :   // -> template(a={...},...) "..."    inline template
    id lp='(' rewrite_template_args ')'
    ( str=DOUBLE_QUOTE_STRING_LITERAL | str=DOUBLE_ANGLE_STRING_LITERAL )
    -> ^(TEMPLATE[$lp,"TEMPLATE"] id rewrite_template_args $str)

  | // -> foo(a={...}, ...)
    rewrite_template_ref

  | // -> ({expr})(a={...}, ...)
    rewrite_indirect_template_head

  | // -> {...}
    ACTION
  ;
```

```
/** -> foo(a={...}, ...) */
rewrite_template_ref
  : id lp='(' rewrite_template_args ')'
    -> ^(TEMPLATE[$lp,"TEMPLATE"] id rewrite_template_args)
  ;

/** -> ({expr})(a={...}, ...) */
rewrite_indirect_template_head
  : lp='(' ACTION ')' '(' rewrite_template_args ')'
    -> ^(TEMPLATE[$lp,"TEMPLATE"] ACTION rewrite_template_args)
  ;

rewrite_template_args
  : rewrite_template_arg (',' rewrite_template_arg)*
    -> ^(ARGLIST rewrite_template_arg+)
  | -> ARGLIST
  ;

rewrite_template_arg
  :   id '=' ACTION -> ^(ARG[$id.start] id ACTION)
  ;

qid : id ('.' id)* ;

id : TOKEN_REF -> ID[$TOKEN_REF]
  | RULE_REF  -> ID[$RULE_REF]
  ;

// L E X I C A L   R U L E S

SL_COMMENT
  : '//'
    ( ' $ANTLR ' SRC // src directive
    | ~('\r'|'\n')*
    )
    '\r'? '\n'
    {$channel=HIDDEN;}
  ;

ML_COMMENT
  : '/*'
    {if (input.LA(1)=='*') $type=DOC_COMMENT; else $channel=HIDDEN;} .*
    '*/'
  ;

CHAR_LITERAL
  : '\'' LITERAL_CHAR '\''
  ;

STRING_LITERAL
  : '\'' LITERAL_CHAR LITERAL_CHAR* '\''
  ;
```

```
fragment
LITERAL_CHAR
  : ESC
  | ~('\''|'\\')
  ;

DOUBLE_QUOTE_STRING_LITERAL
  : '"' (ESC | ~('\\'|'"'))* '"'
  ;

DOUBLE_ANGLE_STRING_LITERAL
  : '<<' .* '>>'
  ;

fragment
ESC : '\\'
    ( 'n'
    | 'r'
    | 't'
    | 'b'
    | 'f'
    | '"'
    | '\''
    | '\\'
    | '>'
    | 'u' XDIGIT XDIGIT XDIGIT XDIGIT
    | . // unknown, leave as it is
    )
  ;

fragment
XDIGIT :
    '0' .. '9'
  | 'a' .. 'f'
  | 'A' .. 'F'
  ;

INT : '0'..'9'+
  ;

ARG_ACTION
  : NESTED_ARG_ACTION
  ;

fragment
NESTED_ARG_ACTION :
  '['
  ( options {greedy=false; k=1;}
  : NESTED_ARG_ACTION
  | ACTION_STRING_LITERAL
  | ACTION_CHAR_LITERAL
  | .
  )*
  ']'
  //{setText(getText().substring(1, getText().length()-1));}
  ;

ACTION
  : NESTED_ACTION ( '?' {$type = SEMPRED;} )?
  ;

fragment
NESTED_ACTION :
  '{'
  ( options {greedy=false; k=2;}
  : NESTED_ACTION
  | SL_COMMENT
  | ML_COMMENT
  | ACTION_STRING_LITERAL
  | ACTION_CHAR_LITERAL
  | .
  )*
  '}'
  ;

fragment
ACTION_CHAR_LITERAL
  : '\'' (ACTION_ESC|~('\\'|'\'')) '\''
  ;

fragment
ACTION_STRING_LITERAL
  : '"' (ACTION_ESC|~('\\'|'"'))* '"'
  ;

fragment
ACTION_ESC
  : '\\\''
  | '\\' '"' // ANTLR doesn't like: '\\"'
  | '\\' ~('\''|'"')
  ;

TOKEN_REF
  : 'A'..'Z' ('a'..'z'|'A'..'Z'|'_'|'0'..'9')*
  ;

RULE_REF
  : 'a'..'z' ('a'..'z'|'A'..'Z'|'_'|'0'..'9')*
  ;

/** Match the start of an options section.  Don't allow normal
 *  action processing on the {...} as it's not a action.
 */
OPTIONS
  : 'options' WS_LOOP '{'
  ;
```

```
TOKENS
  : 'tokens' WS_LOOP '{'
  ;

/** Reset the file and line information; useful when the grammar
 *  has been generated so that errors are shown relative to the
 *  original file like the old C preprocessor used to do.
 */
fragment
SRC : 'src' ' ' file=ACTION_STRING_LITERAL ' ' line=INT
  ;

WS : ( ' '
    | '\t'
    | '\r'? '\n'
    )+
    {$channel=HIDDEN;}
  ;

fragment
WS_LOOP
  : ( WS
    | SL_COMMENT
    | ML_COMMENT
    )*
  ;
```

Appendix B

Triangle compiler sources

This appendix contains the integrated parser specification for the Triangle compiler presented in chapter 6 and the generated parser and tree walkers for this file.

B.1 Integrated parser specification for Triangle

```
////////////////////////////////////////////////////////////////////////////////
// Integrated parser specification for the Triangle compiler                 //
// Copyright (C) 2010 A.J. Admiraal (mailto:code@admiraal.dds.nl)             //
//                                                                            //
// This program is free software; you can redistribute it and/or modify it    //
// under the terms of the GNU General Public License version 2 as published   //
// by the Free Software Foundation.                                           //
//                                                                            //
// This program is distributed in the hope that it will be useful, but        //
// WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY //
// or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License    //
// for more details.                                                          //
//                                                                            //
// You should have received a copy of the GNU General Public License along    //
// with this program; if not, write to the Free Software Foundation, Inc., 51 //
// Franklin Street, Fifth Floor, Boston, MA 02110-1301, USA.                  //
////////////////////////////////////////////////////////////////////////////////

grammar Triangle;

options {
  output=AST;
  ASTLabelType=TriangleTreeNode;
}

tokens {
  ARRAY     = 'array';    BEGIN     = 'begin';    CONST     = 'const';
  DO        = 'do';       ELSE      = 'else';     END       = 'end';
  FUNC      = 'func';     IF        = 'if';       IN        = 'in';
  LET       = 'let';      OF        = 'of';       PROC      = 'proc';
  RECORD    = 'record';   SKIP      = 'skip';     THEN      = 'then';
  TYPE      = 'type';     VAR       = 'var';      WHILE     = 'while';

  ASSIGN    = ':=';       LPAREN    = '(';        RPAREN    = ')';
  LBRACE    = '{';        RBRACE    = '}';        LBLOCK    = '[';
  RBLOCK    = ']';        COLON     = ':';        SEMICOLON = ';';
  DOT       = '.';        COMMA     = ',';        EQUIV     = '~';
}

@header {
  package triangle;

  import java.util.LinkedList;
}

////////////////////////////////////////////////////////////////////////////////
// Checker members

@members {@1
  private KeyValueList<TriangleTreeNode.Type> builtinTypes;
  private KeyValueList<TriangleTreeNode.Type> builtinVariables;
  private Stack<TriangleTreeNode.Type> typeStack =
    new Stack<TriangleTreeNode.Type>();
  private Stack<TriangleTreeNode.RecordType> recordStack =
    new Stack<TriangleTreeNode.RecordType>();
  private Stack<TriangleTreeNode.Scope> scopeStack =
    new Stack<TriangleTreeNode.Scope>();
  private TriangleTreeNode.RecordType currentRecord = null;
  private boolean errorFound = false;

  public void setBuiltins(KeyValueList<TriangleTreeNode.Type> bt,
                          KeyValueList<TriangleTreeNode.Type> bv)
  {
    builtinTypes = bt;
    builtinVariables = bv;
  }

  public boolean allOk()
  {
    return !errorFound;
  }

  private void findDeclaration(TriangleTreeNode node)
  {
    TriangleTreeNode.Type builtin =
      builtinVariables.get(node.getText());

    if (builtin != null)
    {
      node.dataType = builtin;
      return;
    }
  for (int i=scopeStack.size()-1; i>=0; i--)
  {
    TriangleTreeNode declarator =
      scopeStack.get(i).variables.get(node.getText());

    if (declarator != null)
    {
      node.declarator = declarator;

      TriangleTreeNode.Type type = declarator.findType();
      if (type instanceof TriangleTreeNode.RecordType)
        currentRecord = (TriangleTreeNode.RecordType)type;
      else
        currentRecord = null;

      return;
    }
  }

  errorFound = true;
  System.err.println(node.getLine() +
    ": Failed to find a declaration for token \"" +
    node.getText() + "\"");
}

private void findRecElemDeclaration(TriangleTreeNode node)
{
  if (currentRecord != null)
  {
    TriangleTreeNode.Type type =
      currentRecord.elements.get(node.getText());

    if (type != null)
    {
      node.dataType = type;

      if (type instanceof TriangleTreeNode.RecordType)
        currentRecord = (TriangleTreeNode.RecordType)type;
      else
        currentRecord = null;

      return;
    }
  }

  errorFound = true;
  System.err.println(node.getLine() +
    ": Failed to find a record declaration for token \"" +
    node.getText() + "\"");
}

private void findDefinition(TriangleTreeNode node)
```

```
  {
    TriangleTreeNode.Type builtin =
      builtinTypes.get(node.getText());

    if (builtin != null)
    {
      node.dataType = builtin;
      return;
    }

    for (int i=scopeStack.size()-1; i>=0; i--)
    {
      TriangleTreeNode declarator =
        scopeStack.get(i).typedefs.get(node.getText());

      if (declarator != null)
      {
        node.declarator = declarator;
        return;
      }
    }

    errorFound = true;
    System.err.println(node.getLine() +
      ": Failed to find a definition for token \"" +
      node.getText() + "\"");
  }
}

////////////////////////////////////////////////////////////////////////////////
// Generator members

@members {@2
  private Stack<Generator.CodeBlock> codeStack =
    new Stack<Generator.CodeBlock>();
  private Stack<Generator.ExpressionSequence> expressionStack =
    new Stack<Generator.ExpressionSequence>();
  private int references = 0;
  private Generator generator;

  public void setGenerator(Generator g)
  {
    generator = g;
    codeStack.clear();
    codeStack.push(null);
    expressionStack.clear();
    expressionStack.push(new Generator.ExpressionSequence());
  }

  private void closeScope()
  {
    Generator.CodeBlock block = codeStack.pop();
    if (codeStack.peek() != null)
      codeStack.peek().addCode(generator.closeScope(block));
    else
      generator.setRootScope(block);
  }
}

////////////////////////////////////////////////////////////////////////////////
// Commands

command
  : single_command (SEMICOLON! single_command)*
  ;

single_command^
  : IDENTIFIER
    {@1 findDeclaration($IDENTIFIER); }
    {@2 expressionStack.peek().sequence.add($IDENTIFIER); }
    {@2+1 codeStack.peek().addCode(
            generator.generateCommand(expressionStack.peek())); }
    single_command_operator

  | BEGIN^
    {@2 codeStack.push(generator.openScope(codeStack.peek())); }
    command
    END
    {@2 closeScope(); }

  | LET^
    {@1 scopeStack.push(new TriangleTreeNode.Scope()); }
    {@2 codeStack.push(generator.openScope(codeStack.peek())); }
    declaration IN! single_command
    {@1 $LET.dataType = scopeStack.pop(); }
    {@2 closeScope(); }

  | IF^ expression
    THEN! single_command
    ELSE! single_command

  | WHILE^ expression DO
    {@2 codeStack.peek().addCode(
          generator.generateWhile(expressionStack.peek())); }
    single_command
  ;

single_command_operator
  : ((vname_modifier)*
    ASSIGN^
    {@2+1 generator.generateOperator(expressionStack.peek(), $ASSIGN.text); }
    expression
    {@2 codeStack.peek().addCode(
          generator.generateCommand(expressionStack.peek())); }
    )
  | (LPAREN^
    {@2 generator.generateOperator(expressionStack.peek(), "()"); }
    actual_parameter_sequence RPAREN
    {@2 generator.generateExpression(expressionStack.peek()); }
    )
  ;

////////////////////////////////////////////////////////////////////////////////
// Expressions

expression^
  : secondary_expression
  | LET^ declaration IN! expression
  | IF^ expression THEN! expression ELSE! expression
  ;

secondary_expression
  : primary_expression (OPERATOR
    {@2 generator.generateOperator(expressionStack.peek(), $OPERATOR.text); }
    primary_expression)*
  ;

primary_expression
  : IDENTIFIER
    {@1 findDeclaration($IDENTIFIER); }
    {@2 expressionStack.peek().sequence.add($IDENTIFIER); }
    primary_expression_operator

  | INTEGER_LITERAL
    {@2 expressionStack.peek().sequence.
          add(generator.generateIntLiteral($INTEGER_LITERAL)); }
  | CHARACTER_LITERAL
    {@2 expressionStack.peek().sequence.
          add(generator.generateCharLiteral($CHARACTER_LITERAL)); }
  | LPAREN^ expression RPAREN!
  | LBRACE^ record_aggregate RBRACE!
  | LBLOCK^
    {@2 expressionStack.push(new Generator.ExpressionSequence()); }
    array_aggregate RBLOCK
    {@2 String expr = generator.generateExpression(expressionStack.pop());
        expressionStack.peek().sequence.add(expr); }
  ;

primary_expression_operator
  : (vname_modifier)*
  | (LPAREN^
    {@2 expressionStack.push(new Generator.ExpressionSequence());
        generator.generateOperator(expressionStack.peek(), "()"); }
    actual_parameter_sequence RPAREN
    {@2 String expr = generator.generateExpression(expressionStack.pop());
        expressionStack.peek().sequence.add(expr); }
    )
  ;
```

```
record_aggregate
  : IDENTIFIER
    {@1 findDeclaration($IDENTIFIER); }
    EQUIV! expression (COMMA! record_aggregate)?
  ;

array_aggregate
  : expression (COMMA! expression)*
  ;

vname
  : IDENTIFIER
    {@1 findDeclaration($IDENTIFIER); }
    {@2 $IDENTIFIER.references = references;
        references = 0;
        expressionStack.peek().sequence.add($IDENTIFIER); }
    (vname_modifier)*
  ;

vname_modifier
  : DOT^ IDENTIFIER
    {@1 findRecElemDeclaration($IDENTIFIER); }
    {@2 expressionStack.peek().sequence.add($IDENTIFIER); }

  | LBLOCK^
    {@2 expressionStack.push(new Generator.ExpressionSequence()); }
    expression RBLOCK
    {@2 String expr = generator.generateExpression(expressionStack.pop());
        expressionStack.peek().sequence.add(expr); }
  ;

////////////////////////////////////////////////////////////////////////////////
// Declarations

declaration
  : single_declaration (SEMICOLON! single_declaration)*
  ;

single_declaration^
  : CONST^ IDENTIFIER EQUIV! const_denoter
    {@1 $IDENTIFIER.dataType = typeStack.pop();
        scopeStack.peek().variables.add($IDENTIFIER.text, $IDENTIFIER); }
    {@2 codeStack.peek().addType(
          generator.generateDefinition($IDENTIFIER)); }

  | VAR^ IDENTIFIER COLON! type_denoter
    {@1 $IDENTIFIER.dataType = typeStack.pop();
        scopeStack.peek().variables.add($IDENTIFIER.text, $IDENTIFIER); }
    {@2 codeStack.peek().addDeclaration(
          generator.generateDeclaration($IDENTIFIER)); }

  | PROC^ IDENTIFIER
```

```
    {@1 scopeStack.push(new TriangleTreeNode.ProcType()); }
    {@2 codeStack.push(generator.openScope(
          generator.generateDefinition($IDENTIFIER), codeStack.peek())); }
    LPAREN! formal_parameter_sequence RPAREN!
    EQUIV! single_command
    {@1 $IDENTIFIER.dataType = scopeStack.pop();
        scopeStack.peek().variables.add($IDENTIFIER.text, $IDENTIFIER); }
    {@2 String code = generator.closeScope(codeStack.pop());
        codeStack.peek().addMethod(code); }

  | FUNC^ IDENTIFIER
    {@1 scopeStack.push(new TriangleTreeNode.FuncType()); }
    {@2 codeStack.push(generator.openScope(
          generator.generateDefinition($IDENTIFIER), codeStack.peek()));
        expressionStack.push(new Generator.ExpressionSequence()); }
    LPAREN! formal_parameter_sequence RPAREN!
    COLON! type_denoter EQUIV! expression
    {@1 $IDENTIFIER.dataType = scopeStack.pop();
        ((TriangleTreeNode.FuncType)$IDENTIFIER.dataType).type =
          typeStack.pop();
        scopeStack.peek().variables.add($IDENTIFIER.text, $IDENTIFIER); }
    {@2 codeStack.peek().addCode(
          generator.generateFunctionBody(expressionStack.pop()));
        String code = generator.closeScope(codeStack.pop());
        codeStack.peek().addMethod(code); }

  | TYPE^ IDENTIFIER EQUIV! type_denoter
    {@1 $IDENTIFIER.dataType = typeStack.pop();
        scopeStack.peek().typedefs.add($IDENTIFIER.text, $IDENTIFIER); }
    {@2 codeStack.peek().addType(
          generator.generateDeclaration($IDENTIFIER)); }
  ;

formal_parameter_sequence^
  : (formal_parameter (COMMA! formal_parameter)*)?
  ;

formal_parameter
  : IDENTIFIER COLON^ type_denoter
    {@1 $IDENTIFIER.dataType = typeStack.pop();
        ((TriangleTreeNode.ProcType)scopeStack.peek()).
          variables.add($IDENTIFIER.text, $IDENTIFIER);
        ((TriangleTreeNode.ProcType)scopeStack.peek()).
          parameters.add($IDENTIFIER.text, $IDENTIFIER); }

  | VAR^ IDENTIFIER COLON! type_denoter
    {@1 $IDENTIFIER.dataType =
          new TriangleTreeNode.PointerType(typeStack.pop());
        ((TriangleTreeNode.ProcType)scopeStack.peek()).
          variables.add($IDENTIFIER.text, $IDENTIFIER);
        ((TriangleTreeNode.ProcType)scopeStack.peek()).
          parameters.add($IDENTIFIER.text, $IDENTIFIER); }
  | PROC^ IDENTIFIER
    LPAREN! formal_parameter_sequence RPAREN!
  | FUNC^ IDENTIFIER
    LPAREN! formal_parameter_sequence RPAREN!
    COLON! type_denoter
  ;

actual_parameter_sequence^
  : (actual_parameter (COMMA
    {@2 expressionStack.peek().sequence.add(generator.generateNext()); }
    actual_parameter)*)?
  ;

actual_parameter
  : expression
  | VAR^ {@2 references++; } vname
  | PROC^ IDENTIFIER
  | FUNC^ IDENTIFIER
  ;

////////////////////////////////////////////////////////////////////////////////
// Types

const_denoter^
  : INTEGER_LITERAL
    {@1 typeStack.push(
          new TriangleTreeNode.ConstType($INTEGER_LITERAL)); }
  | CHARACTER_LITERAL
    {@1 typeStack.push(
          new TriangleTreeNode.ConstType($CHARACTER_LITERAL)); }
  ;

type_denoter^
  : IDENTIFIER
    {@1 findDefinition($IDENTIFIER);
        typeStack.push(
          new TriangleTreeNode.SingleType($IDENTIFIER)); }

  | ARRAY^ INTEGER_LITERAL OF! type_denoter
    {@1 typeStack.push(
          new TriangleTreeNode.ArrayType(typeStack.pop(),
            Integer.valueOf($INTEGER_LITERAL.text))); }

  | RECORD^
    {@1 recordStack.push(new TriangleTreeNode.RecordType()); }
    record_type_denoter END
    {@1 typeStack.push(recordStack.pop()); }
  ;

record_type_denoter
  : IDENTIFIER COLON! type_denoter
    {@1 recordStack.peek().elements.add($IDENTIFIER.text, typeStack.pop()); }
    (COMMA! record_type_denoter)?
  ;


////////////////////////////////////////////////////////////////////////////////
// Lexical analyzer

INTEGER_LITERAL   : ('0x' ('0' .. '9' |
                           'A' .. 'F' |
                           'a' .. 'f')+) |
                    ('0' .. '9')+;
CHARACTER_LITERAL : '\'' (('\t') | (' ' .. '&') |
                          ('(' .. '[') | (']' .. '~'))*
                    '\'';
IDENTIFIER        : ('a'..'z' | 'A' .. 'Z')
                    ('a'..'z' | 'A' .. 'Z' | '0' .. '9')*;
OPERATOR          : ('+' | '-' | '*' | '/' | '=' | '<' | '>' |
                     '\\' | '&' | '@' | '%' | '^' | '?')+;
COMMENT           : ('!' (('\t') | (' '..'!') |
                         ('#'..'[') | (']'..'~'))* '\r'? '\n'
                    ) { $channel=HIDDEN; };
WHITESPACE        : (' ' | '\t' | '\f' | '\r'? '\n')+
                    { $channel=HIDDEN; };
```

B.2 Generated checker

```
tree grammar TriangleChecker;

options {
  output = AST;
  ASTLabelType = TriangleTreeNode;
  backtrack = true;
}

tokens {
  ARRAY; BEGIN; CONST; DO; ELSE; END; FUNC; IF; IN; LET; OF; PROC; RECORD;
  SKIP; THEN; TYPE; VAR; WHILE; ASSIGN; LPAREN; RPAREN; LBRACE; RBRACE;
  LBLOCK; RBLOCK; COLON; SEMICOLON; DOT; COMMA; EQUIV; INTEGER_LITERAL;
  CHARACTER_LITERAL; IDENTIFIER; OPERATOR; COMMENT; WHITESPACE;
}

@header {
  package triangle;

  import java.util.LinkedList;
}

@members {
  private KeyValueList<TriangleTreeNode.Type> builtinTypes;
  private KeyValueList<TriangleTreeNode.Type> builtinVariables;
  private Stack<TriangleTreeNode.Type> typeStack =
    new Stack<TriangleTreeNode.Type>();
  private Stack<TriangleTreeNode.RecordType> recordStack =
  new Stack<TriangleTreeNode.RecordType>();
private Stack<TriangleTreeNode.Scope> scopeStack =
  new Stack<TriangleTreeNode.Scope>();
private TriangleTreeNode.RecordType currentRecord = null;
private boolean errorFound = false;

public void setBuiltins(KeyValueList<TriangleTreeNode.Type> bt,
                        KeyValueList<TriangleTreeNode.Type> bv)
{
  builtinTypes = bt;
  builtinVariables = bv;
}

public boolean allOk()
{
  return !errorFound;
}

private void findDeclaration(TriangleTreeNode node)
{
  TriangleTreeNode.Type builtin =
    builtinVariables.get(node.getText());

  if (builtin != null)
  {
    node.dataType = builtin;
    return;
  }

  for (int i=scopeStack.size()-1; i>=0; i--)
  {
    TriangleTreeNode declarator =
      scopeStack.get(i).variables.get(node.getText());

    if (declarator != null)
    {
      node.declarator = declarator;

      TriangleTreeNode.Type type = declarator.findType();
      if (type instanceof TriangleTreeNode.RecordType)
        currentRecord = (TriangleTreeNode.RecordType)type;
      else
        currentRecord = null;

      return;
    }
  }

  errorFound = true;
  System.err.println(node.getLine() +
    ": Failed to find a declaration for token \"" +
    node.getText() + "\"");
}
```

```
private void findRecElemDeclaration(TriangleTreeNode node)
{
  if (currentRecord != null)
  {
    TriangleTreeNode.Type type =
      currentRecord.elements.get(node.getText());

    if (type != null)
    {
      node.dataType = type;

      if (type instanceof TriangleTreeNode.RecordType)
        currentRecord = (TriangleTreeNode.RecordType)type;
      else
        currentRecord = null;

      return;
    }
  }

  errorFound = true;
  System.err.println(node.getLine() +
    ": Failed to find a record declaration for token \"" +
    node.getText() + "\"");
}

private void findDefinition(TriangleTreeNode node)
{
  TriangleTreeNode.Type builtin =
    builtinTypes.get(node.getText());

  if (builtin != null)
  {
    node.dataType = builtin;
    return;
  }

  for (int i=scopeStack.size()-1; i>=0; i--)
  {
    TriangleTreeNode declarator =
      scopeStack.get(i).typedefs.get(node.getText());

    if (declarator != null)
    {
      node.declarator = declarator;
      return;
    }
  }

  errorFound = true;
  System.err.println(node.getLine() +
    ": Failed to find a definition for token \"" +
      node.getText() + "\"");
  }
}

command
  : single_command command_suf_2?
  ;

command_suf_2
  : (single_command )+
  ;

single_command
  : single_command_pre_1
  | ^( BEGIN type_denoter_pre_3 single_command command_suf_2? END )
  | ^( LET
      { scopeStack.push(new TriangleTreeNode.Scope()); }
     type_denoter_pre_3 single_declaration declaration_suf_2? single_command
      { $LET.dataType = scopeStack.pop(); }
     )
  | ^( IF type_denoter_pre_3 expression single_command single_command )
  | ^( WHILE type_denoter_pre_3 expression DO single_command )
  ;

single_command_pre_1
  : IDENTIFIER
      { findDeclaration($IDENTIFIER); }
     single_command_pre_1
  | vname_modifier single_command_operator_suf_2?
  | ^( ASSIGN single_command_operator_suf_2? expression
    single_command_operator_suf_2? )
  | ^( LPAREN actual_parameter_sequence? RPAREN )
  ;

single_command_operator_suf_2
  : (vname_modifier )+
  ;

expression
  : expression_pre_1
  | ^( LET type_denoter_pre_3 single_declaration declaration_suf_2? expression
    )
  | ^( IF type_denoter_pre_3 expression expression expression )
  ;

expression_pre_1
  : secondary_expression_suf_1 secondary_expression_suf_1?
  ;

secondary_expression_suf_1
  : OPERATOR secondary_expression_suf_1 secondary_expression_suf_2?
  | IDENTIFIER
      { findDeclaration($IDENTIFIER); }
     primary_expression_operator?
  | INTEGER_LITERAL
  | CHARACTER_LITERAL
  | ^( LPAREN expression )
  | ^( LBRACE IDENTIFIER
      { findDeclaration($IDENTIFIER); }
     expression record_aggregate_suf_1? )
  | ^( LBLOCK expression array_aggregate_suf_2? RBLOCK )
  ;

secondary_expression_suf_2
  : (OPERATOR secondary_expression_suf_1 )+
  ;

primary_expression_operator
  : vname_modifier primary_expression_operator_suf_1?
  | ^( LPAREN actual_parameter_sequence? RPAREN )
  ;

primary_expression_operator_suf_1
  : (vname_modifier )+
  ;

record_aggregate
  : IDENTIFIER
      { findDeclaration($IDENTIFIER); }
     expression record_aggregate_suf_1?
  ;

record_aggregate_suf_1
  : record_aggregate
  ;

array_aggregate_suf_2
  : (expression )+
  ;

vname_suf_2
  : (vname_modifier )+
  ;

vname_modifier
  : ^( DOT IDENTIFIER
      { findRecElemDeclaration($IDENTIFIER); }
     )
  | ^( LBLOCK expression RBLOCK )
  ;

declaration_suf_2
  : (single_declaration )+
  ;

single_declaration
```

```
  : ^( CONST type_denoter_pre_3 IDENTIFIER const_denoter
      { $IDENTIFIER.dataType = typeStack.pop();
        scopeStack.peek().variables.add($IDENTIFIER.text, $IDENTIFIER); }
     )
  | ^( VAR type_denoter_pre_3 IDENTIFIER type_denoter
      { $IDENTIFIER.dataType = typeStack.pop();
        scopeStack.peek().variables.add($IDENTIFIER.text, $IDENTIFIER); }
     )
  | ^( PROC type_denoter_pre_3 IDENTIFIER
      { scopeStack.push(new TriangleTreeNode.ProcType()); }
     formal_parameter_sequence? single_command
      { $IDENTIFIER.dataType = scopeStack.pop();
        scopeStack.peek().variables.add($IDENTIFIER.text, $IDENTIFIER); }
     )
  | ^( FUNC type_denoter_pre_3 IDENTIFIER
      { scopeStack.push(new TriangleTreeNode.FuncType()); }
     formal_parameter_sequence? type_denoter expression
      { $IDENTIFIER.dataType = scopeStack.pop();
        ((TriangleTreeNode.FuncType)$IDENTIFIER.dataType).type =
        typeStack.pop();
        scopeStack.peek().variables.add($IDENTIFIER.text, $IDENTIFIER); }
     )
  | ^( TYPE type_denoter_pre_3 IDENTIFIER type_denoter
      { $IDENTIFIER.dataType = typeStack.pop();
        scopeStack.peek().typedefs.add($IDENTIFIER.text, $IDENTIFIER); }
     )
  ;

formal_parameter_sequence
  : formal_parameter_sequence_pre_2 formal_parameter_sequence_suf_2?
  ;

formal_parameter_sequence_pre_2
  : formal_parameter formal_parameter_sequence_suf_2?
  ;

formal_parameter_sequence_suf_2
  : formal_parameter_sequence_pre_2
  ;

formal_parameter
  : ^( COLON IDENTIFIER type_denoter
      { $IDENTIFIER.dataType = typeStack.pop();
        ((TriangleTreeNode.ProcType)scopeStack.peek()).
        variables.add($IDENTIFIER.text, $IDENTIFIER);
        ((TriangleTreeNode.ProcType)scopeStack.peek()).
        parameters.add($IDENTIFIER.text, $IDENTIFIER); }
     )
  | ^( VAR IDENTIFIER type_denoter
      { $IDENTIFIER.dataType =
        new TriangleTreeNode.PointerType(typeStack.pop());
        ((TriangleTreeNode.ProcType)scopeStack.peek()).
        variables.add($IDENTIFIER.text, $IDENTIFIER);
        ((TriangleTreeNode.ProcType)scopeStack.peek()).
        parameters.add($IDENTIFIER.text, $IDENTIFIER); }
     )
  | ^( PROC IDENTIFIER formal_parameter_sequence? )
  | ^( FUNC IDENTIFIER formal_parameter_sequence? type_denoter )
  ;

actual_parameter_sequence
  : actual_parameter_sequence_pre_1 actual_parameter_sequence_suf_2?
  ;

actual_parameter_sequence_pre_1
  : actual_parameter_sequence_pre_2 actual_parameter_sequence_suf_2?
  ;

actual_parameter_sequence_pre_2
  : COMMA actual_parameter_sequence_pre_2 actual_parameter_sequence_suf_2?
  | expression
  | ^( VAR IDENTIFIER
      { findDeclaration($IDENTIFIER); }
     vname_suf_2? )
  | ^( PROC IDENTIFIER )
  | ^( FUNC IDENTIFIER )
  ;

actual_parameter_sequence_suf_2
  : actual_parameter_sequence_pre_2
  ;

const_denoter
  : const_denoter_pre_1
  | const_denoter_pre_2
  ;

const_denoter_pre_1
  : INTEGER_LITERAL
      { typeStack.push(
        new TriangleTreeNode.ConstType($INTEGER_LITERAL)); }

  ;

const_denoter_pre_2
  : CHARACTER_LITERAL
      { typeStack.push(
        new TriangleTreeNode.ConstType($CHARACTER_LITERAL)); }

  ;

type_denoter
  : type_denoter_pre_1
  | ^( ARRAY type_denoter_pre_3 INTEGER_LITERAL type_denoter
      { typeStack.push(
        new TriangleTreeNode.ArrayType(typeStack.pop(),
        Integer.valueOf($INTEGER_LITERAL.text))); }
     )
  | ^( RECORD
      { recordStack.push(new TriangleTreeNode.RecordType()); }
     type_denoter_pre_3 IDENTIFIER type_denoter
      { recordStack.peek().elements.add($IDENTIFIER.text, typeStack.pop()); }
     record_type_denoter_suf_1? END
      { typeStack.push(recordStack.pop()); }
     )
  ;

type_denoter_pre_1
  : IDENTIFIER
      { findDefinition($IDENTIFIER);
        typeStack.push(
        new TriangleTreeNode.SingleType($IDENTIFIER)); }

  ;

type_denoter_pre_3
  :
  ;

record_type_denoter
  : IDENTIFIER type_denoter
      { recordStack.peek().elements.add($IDENTIFIER.text, typeStack.pop()); }
     record_type_denoter_suf_1?
  ;

record_type_denoter_suf_1
  : record_type_denoter
  ;
```

B.3 Generated generator

```
tree grammar TriangleGenerator;

options {
  output = AST;
  ASTLabelType = TriangleTreeNode;
  backtrack = true;
}

tokens {
  ARRAY; BEGIN; CONST; DO; ELSE; END; FUNC; IF; IN; LET; OF; PROC; RECORD;
  SKIP; THEN; TYPE; VAR; WHILE; ASSIGN; LPAREN; RPAREN; LBRACE; RBRACE;
  LBLOCK; RBLOCK; COLON; SEMICOLON; DOT; COMMA; EQUIV; INTEGER_LITERAL;
  CHARACTER_LITERAL; IDENTIFIER; OPERATOR; COMMENT; WHITESPACE;
}

@header {
  package triangle;

  import java.util.LinkedList;
}

@members {
  private Stack<Generator.CodeBlock> codeStack =
    new Stack<Generator.CodeBlock>();
  private Stack<Generator.ExpressionSequence> expressionStack =
    new Stack<Generator.ExpressionSequence>();
  private int references = 0;
  private Generator generator;

  public void setGenerator(Generator g)
  {
    generator = g;
    codeStack.clear();
    codeStack.push(null);
    expressionStack.clear();
    expressionStack.push(new Generator.ExpressionSequence());
  }

  private void closeScope()
  {
    Generator.CodeBlock block = codeStack.pop();

    if (codeStack.peek() != null)
      codeStack.peek().addCode(generator.closeScope(block));
    else
      generator.setRootScope(block);
  }
}

command
  : single_command command_suf_2?
  ;

command_suf_2
  : (single_command )+
  ;

single_command
  : single_command_pre_1
  | ^( BEGIN
      { codeStack.push(generator.openScope(codeStack.peek())); }
     type_denoter_pre_3 single_command command_suf_2? END
      { closeScope(); }
     )
  | ^( LET
      { codeStack.push(generator.openScope(codeStack.peek())); }
     type_denoter_pre_3 single_declaration declaration_suf_2? single_command
      { closeScope(); }
     )
  | ^( IF type_denoter_pre_3 expression single_command single_command )
  | ^( WHILE type_denoter_pre_3 expression DO
      { codeStack.peek().addCode(
        generator.generateWhile(expressionStack.peek())); }
     single_command )
  ;

single_command_pre_1
  : IDENTIFIER
      { expressionStack.peek().sequence.add($IDENTIFIER); }
     single_command_pre_1
      { codeStack.peek().addCode(
        generator.generateCommand(expressionStack.peek())); }

  | vname_modifier single_command_operator_suf_2?
  | ^( ASSIGN single_command_operator_suf_2?
      { generator.generateOperator(expressionStack.peek(), $ASSIGN.text); }
     expression
      { codeStack.peek().addCode(
        generator.generateCommand(expressionStack.peek())); }
     single_command_operator_suf_2? )
  | ^( LPAREN
      { generator.generateOperator(expressionStack.peek(), "()"); }
     actual_parameter_sequence? RPAREN
      { generator.generateExpression(expressionStack.peek()); }
     )
  ;

single_command_operator_suf_2
  : (vname_modifier )+
  ;

expression
  : expression_pre_1
  | ^( LET type_denoter_pre_3 single_declaration declaration_suf_2? expression
    )
  | ^( IF type_denoter_pre_3 expression expression expression )
  ;

expression_pre_1
  : secondary_expression_suf_1 secondary_expression_suf_1?
  ;

secondary_expression_suf_1
  : OPERATOR
      { generator.generateOperator(expressionStack.peek(), $OPERATOR.text); }
     secondary_expression_suf_1 secondary_expression_suf_2?
  | IDENTIFIER
      { expressionStack.peek().sequence.add($IDENTIFIER); }
     primary_expression_operator?
  | INTEGER_LITERAL
      { expressionStack.peek().sequence.
        add(generator.generateIntLiteral($INTEGER_LITERAL)); }
  | CHARACTER_LITERAL
      { expressionStack.peek().sequence.
        add(generator.generateCharLiteral($CHARACTER_LITERAL)); }

  | ^( LPAREN expression )
  | ^( LBRACE IDENTIFIER expression record_aggregate_suf_1? )
  | ^( LBLOCK
      { expressionStack.push(new Generator.ExpressionSequence()); }
     expression array_aggregate_suf_2? RBLOCK
      { String expr = generator.generateExpression(expressionStack.pop());
        expressionStack.peek().sequence.add(expr); }
     )
  ;

secondary_expression_suf_2
  : (OPERATOR
      { generator.generateOperator(expressionStack.peek(), $OPERATOR.text); }
     secondary_expression_suf_1 )+
  ;

primary_expression_operator
  : vname_modifier primary_expression_operator_suf_1?
  | ^( LPAREN
      { expressionStack.push(new Generator.ExpressionSequence());
        generator.generateOperator(expressionStack.peek(), "()"); }
     actual_parameter_sequence? RPAREN
      { String expr = generator.generateExpression(expressionStack.pop());
        expressionStack.peek().sequence.add(expr); }
     )
  ;

primary_expression_operator_suf_1
  : (vname_modifier )+
  ;

record_aggregate
  : IDENTIFIER expression record_aggregate_suf_1?
  ;

record_aggregate_suf_1
  : record_aggregate
  ;

array_aggregate_suf_2
  : (expression )+
  ;

vname_suf_2
  : (vname_modifier )+
  ;

vname_modifier
```

```
  : ^( DOT IDENTIFIER
      { expressionStack.peek().sequence.add($IDENTIFIER); }
     )
  | ^( LBLOCK
      { expressionStack.push(new Generator.ExpressionSequence()); }
     expression RBLOCK
      { String expr = generator.generateExpression(expressionStack.pop());
        expressionStack.peek().sequence.add(expr); }
     )
  ;

declaration_suf_2
  : (single_declaration )+
  ;

single_declaration
  : ^( CONST type_denoter_pre_3 IDENTIFIER const_denoter
      { codeStack.peek().addType(
        generator.generateDefinition($IDENTIFIER)); }
     )
  | ^( VAR type_denoter_pre_3 IDENTIFIER type_denoter
      { codeStack.peek().addDeclaration(
        generator.generateDeclaration($IDENTIFIER)); }
     )
  | ^( PROC type_denoter_pre_3 IDENTIFIER
      { codeStack.push(generator.openScope(
        generator.generateDefinition($IDENTIFIER), codeStack.peek())); }
     formal_parameter_sequence? single_command
      { String code = generator.closeScope(codeStack.pop());
        codeStack.peek().addMethod(code); }
     )
  | ^( FUNC type_denoter_pre_3 IDENTIFIER
      { codeStack.push(generator.openScope(
        generator.generateDefinition($IDENTIFIER), codeStack.peek()));
        expressionStack.push(new Generator.ExpressionSequence()); }
     formal_parameter_sequence? type_denoter expression
      { codeStack.peek().addCode(
        generator.generateFunctionBody(expressionStack.pop()));
        String code = generator.closeScope(codeStack.pop());
        codeStack.peek().addMethod(code); }
     )
  | ^( TYPE type_denoter_pre_3 IDENTIFIER type_denoter
      { codeStack.peek().addType(
        generator.generateDeclaration($IDENTIFIER)); }
     )
  ;

formal_parameter_sequence
  : formal_parameter_sequence_pre_2 formal_parameter_sequence_suf_2?
  ;

formal_parameter_sequence_pre_2
  : formal_parameter formal_parameter_sequence_suf_2?
  ;

formal_parameter_sequence_suf_2
  : formal_parameter_sequence_pre_2
  ;

formal_parameter
  : ^( COLON IDENTIFIER type_denoter )
  | ^( VAR IDENTIFIER type_denoter )
  | ^( PROC IDENTIFIER formal_parameter_sequence? )
  | ^( FUNC IDENTIFIER formal_parameter_sequence? type_denoter )
  ;

actual_parameter_sequence
  : actual_parameter_sequence_pre_1 actual_parameter_sequence_suf_2?
  ;

actual_parameter_sequence_pre_1
  : actual_parameter_sequence_pre_2 actual_parameter_sequence_suf_2?
  ;

actual_parameter_sequence_pre_2
  : COMMA
      { expressionStack.peek().sequence.add(generator.generateNext()); }
     actual_parameter_sequence_pre_2 actual_parameter_sequence_suf_2?
  | expression
  | ^( VAR
      { references++; }
     IDENTIFIER
      { $IDENTIFIER.references = references;
        references = 0;
        expressionStack.peek().sequence.add($IDENTIFIER); }
     vname_suf_2? )
  | ^( PROC IDENTIFIER )
  | ^( FUNC IDENTIFIER )
  ;

actual_parameter_sequence_suf_2
  : actual_parameter_sequence_pre_2
  ;

const_denoter
  : const_denoter_pre_1
  | const_denoter_pre_2
  ;

const_denoter_pre_1
  : INTEGER_LITERAL
  ;

const_denoter_pre_2
  : CHARACTER_LITERAL
  ;
```

```
type_denoter
  : type_denoter_pre_1
  | ^( ARRAY type_denoter_pre_3 INTEGER_LITERAL type_denoter )
  | ^( RECORD type_denoter_pre_3 IDENTIFIER type_denoter
    record_type_denoter_suf_1? END )
  ;

type_denoter_pre_1
  : IDENTIFIER
  ;

type_denoter_pre_3
  :
  ;

record_type_denoter
  : IDENTIFIER type_denoter record_type_denoter_suf_1?
  ;

record_type_denoter_suf_1
  : record_type_denoter
  ;
```

Software licenses

This appendix contains the licenses under which the software is distributed. All code reused from the ANTLR project is licensed under the BSD license as provided in section B.3, all other code is licensed under the General Public License as provided in section B.3.

ANTLR BSD license

General Public License

Preamble

The licenses for most software are designed to take away your freedom to share and change it. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change free software—to make sure the software is free for all its users. This General Public License applies to most of the Free Software Foundation's software and to any other program whose authors commit to using it. (Some other Free Software Foundation software is covered by the GNU Library General Public License instead.) You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for this service if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs; and that you know you can do these things.

To protect your rights, we need to make restrictions that forbid anyone to deny you these rights or to ask you to surrender the rights. These restrictions translate to certain responsibilities for you if you distribute copies of the software, or if you modify it.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must give the recipients all the rights that you have. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

We protect your rights with two steps: (1) copyright the software, and (2) offer you this license which gives you legal permission to copy, distribute and/or modify the software.

Also, for each author's protection and ours, we want to make certain that everyone understands that there is no warranty for this free software. If the software is modified by someone else and passed on, we want its recipients to know that what they have is not the original, so that any problems introduced by others will not reflect on the original authors' reputations.

Finally, any free program is threatened constantly by software patents. We wish to avoid the danger that redistributors of a free program will individually obtain patent licenses, in effect making the program proprietary. To prevent this, we have made it clear that any patent must be licensed for everyone's free use or not licensed at all.

The precise terms and conditions for copying, distribution and modification follow.

Terms and Conditions For Copying, Distribution and Modification

0. This License applies to any program or other work which contains a notice placed by the copyright holder saying it may be distributed under the terms of this General Public License. The "Program", below, refers to any such program or work, and a "work based on the Program" means either the Program or any derivative work under copyright law: that is to say, a work containing the Program or a portion of it, either verbatim or with modifications and/or translated into another language. (Hereinafter, translation is included without limitation in the term "modification".) Each licensee is addressed as "you".

   Activities other than copying, distribution and modification are not covered by this License; they are outside its scope. The act of running the Program is not restricted, and the output from the Program is covered only if its contents constitute a work based on the Program (independent of having been made by running the Program). Whether that is true depends on what the Program does.

1. You may copy and distribute verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this License and to the absence of any warranty; and give any other recipients of the Program a copy of this License along with the Program.

   You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.

2. You may modify your copy or copies of the Program or any portion of it, thus forming a work based on the Program, and copy and distribute such modifications or work under the terms of Section 1 above, provided that you also meet all of these conditions:

   (a) You must cause the modified files to carry prominent notices stating that you changed the files and the date of any change.

   (b) You must cause any work that you distribute or publish, that in whole or in part contains or is derived from the Program or any part thereof, to be licensed as a whole at no charge to all third parties under the terms of this License.

   (c) If the modified program normally reads commands interactively when run, you must cause it, when started running for such interactive use in the most ordinary way, to print or display an announcement including an appropriate copyright notice and a notice that there is no warranty (or else, saying that you provide a warranty) and that users may redistribute the program under these conditions, and telling the user how to view a copy of this License. (Exception: if the Program itself is interactive but does not normally print such an announcement, your work based on the Program is not required to print an announcement.)

   These requirements apply to the modified work as a whole. If identifiable sections of that work are not derived from the Program, and can be reasonably considered independent and separate works in themselves, then this License, and its terms, do not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on the Program, the distribution of the whole must be on the terms of this License, whose permissions for other licensees extend to the entire whole, and thus to each and every part regardless of who wrote it.

   Thus, it is not the intent of this section to claim rights or contest your rights to work written entirely by you; rather, the intent is to exercise the right to control the distribution of derivative or collective works based on the Program.

   In addition, mere aggregation of another work not based on the Program with the Program (or with a work based on the Program) on a volume of a storage or distribution medium does not bring the other work under the scope of this License.

3. You may copy and distribute the Program (or a work based on it, under Section 2) in object code or executable form under the terms of Sections 1 and 2 above provided that you also do one of the following:

(a) Accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

(b) Accompany it with a written offer, valid for at least three years, to give any third party, for a charge no more than your cost of physically performing source distribution, a complete machine-readable copy of the corresponding source code, to be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

(c) Accompany it with the information you received as to the offer to distribute corresponding source code. (This alternative is allowed only for noncommercial distribution and only if you received the program in object code or executable form with such an offer, in accord with Subsection b above.)

The source code for a work means the preferred form of the work for making modifications to it. For an executable work, complete source code means all the source code for all modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the executable. However, as a special exception, the source code distributed need not include anything that is normally distributed (in either source or binary form) with the major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompanies the executable.

If distribution of executable or object code is made by offering access to copy from a designated place, then offering equivalent access to copy the source code from the same place counts as distribution of the source code, even though third parties are not compelled to copy the source along with the object code.

4. You may not copy, modify, sublicense, or distribute the Program except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense or distribute the Program is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

5. You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to modify or distribute the Program or its derivative works. These actions are prohibited by law if you do not accept this License. Therefore, by modifying or distributing the Program (or any work based on the Program), you indicate your acceptance of this License to do so, and all its terms and conditions for copying, distributing or modifying the Program or works based on it.

6. Each time you redistribute the Program (or any work based on the Program), the recipient automatically receives a license from the original licensor to copy, distribute or modify the Program subject to these terms and conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted herein. You are not responsible for enforcing compliance by third parties to this License.

7. If, as a consequence of a court judgment or allegation of patent infringement or for any other reason (not limited to patent issues), conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot distribute so as to satisfy simultaneously your obligations under this License and any other pertinent obligations,

Bibliography

[1] G. Hopper. The education of a computer. *Proceedings of the Association for Computing Machinery Conference, May 1952*, pages 243–249, 1952.

[2] A. Karatsuba and Yu Ofman. Multiplication of many-digital numbers by automatic computers. *Physics-Doklady*, 7:595–596, 1963.

[3] B. W. Leverett, R. G. Cattell, S. O. Hobbs, J. M. Newcomer, A. H. Reiner, B. R. Schatz, and W. A. Wulf. An overview of the production-quality compiler-compiler project. *IEEE Computer*, 13:38–49, 1980.

[4] T. J. Parr and R. W. Quong. Antlr: A predicated-ll(k) parser generator. *Software-practice and experience*, 25, 1995.

[5] ISO/IEC. *ISO/IEC 14977:1996(E) Extended BNF.*
`http://www.iso.org/cate/d26153.html`.

[6] Antlr project home.
`http://antlr.org/`.

[7] D. A. Watt and D. F. Brown. *Programming language processors in Java.* Pearson Education Limited, Edinburgh Gate, 2000. ISBN:0-130-25786-9.

[8] T. J. Parr. *The Definitive ANTLR Reference: Building Domain-Specific Languages.* The Pragmatic Bookshelf, 2007. ISBN:978-0-9787-3925-6.

[9] N. Wirth. What can we do about the unnecessary diversity of notation for syntactic definitions? *CACM*, 20:822–823, 1977.

[10] W3C. *Extensible Markup Language (XML) 1.0 (Fourth Edition).*
`http://www.w3.org/TR/REC-xml/`.

[11] J. W. Backus. The syntax and semantics of the proposed international algebraic language of zürich acm-gamm conference. *Proceedings of the International Conference on Information Processing*, pages 125–132, 1959.

[12] N. Chomsky. Three models for the description of language. *IEEE Transactions on Information Theory*, 2:113–124, 1956.

[13] K. E. Iverson. *A Programming Language.* Wiley, New York, 1962. ISBN:0-471430-14-5.

[14] T. J. Parr. Translators should use tree grammars, 2004.
`http://www.antlr.org/article/1100569809276/use.tree.grammars.tml`.

[15] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns: elements of reusable object-oriented software.* Addison-Wesley Professional, 1994. ISBN:978-0201633610.

[16] A. Tripp. Manual tree walking is better than tree grammars, 2006.
`http://www.antlr.org/article/1170602723163/treewalkers.html`.

[17] T. J. Parr. Rewrite rules, 2008.
`http://www.antlr.org/wiki/display/~admin/2008/04/11/Rewrite+rules`.

[18] J. Bovet and T. J. Parr. Antlrworks: an antlr grammar development environment. *Software: Practice and Experience*, 38:1305–1332, 2008.

[19] H. G. Rice. Classes of recursively enumerable sets and their decision problems. *Transactions of the American Mathematical Society*, 74:358–366, 1953.

[20] H.V. Jagadish, Laks V.S. Lakshmanan, Divesh Srivastava, and Keith Thompson. Tax: A tree algebra for xml. *Lecture Notes in Computer Science*, pages 149–164, 2002. ISBN:978-3-540-44080-2.

[21] C.M. Hoffmann and M.J. O'Donnell. An interpreter generator using tree pattern matching. *JACM*, 29:68–95, 1982.

[22] Javacc project home.
`https://javacc.dev.java.net/`.

[23] V. Kodaganallur. Incorporating language processing into java applications: A javacc tutorial. *IEEE Software*, 21:70–77, 2004.