Using a Graph Transformation Tool to generate a Parser

Peter van Dijk
University of Twente, NL
P.O. Box 217, 7500AE Enschede
The Netherlands
p.a.h.vandijk@student.utwente.nl

ABSTRACT

Graph transformation tools (GTTs) are powerful tools for analysing syntax trees. In the state of the art importing syntax trees in GTTs requires a format-specific converter that is prone to changes. This research solves the dependence on import tools by using the GROOVE GTT to subsume all steps involved in creating the syntax tree. Consequently our approach is wholly defined in terms of graph transformations for both the LL(1) parser and its generator. We find that our approach is indeed correct and complete, but suffers a performance penalty in comparison to traditional parsers.

Keywords

Graph transformation tool, Parser, Parser generator, Syntax tree, LL(1), GROOVE

1. INTRODUCTION

To analyse a sentence, one builds a syntax tree:, a tree depicting the grammatical structure of the English sentence. Figure 1 depicts a syntax tree for the English sentence “the man bit the dog”. Such trees can also be created for programming languages, an example of which can be seen in Figure 2 for the arithmetical sentence “2*8+4*8”. Texts can be analysed using a *grammar*, which defines the rules of the natural language or programming language. As long as the input text is valid, a syntax tree can be generated, so the process of creating a syntax tree also validates the text. That is why constructing syntax trees is an important job of parsers and compilers. Syntax trees of programming languages can also be used by compilers to create executable code [1]. There are many types of parsers that can create a syntax tree from a text; e.g. LL, LR, LALR, SLR.

Since trees are a subset of graphs, graph transformation tools (GTTs) are in principle suitable for manipulating syntax trees. They are a powerful tool for semantic analysis of syntax graphs (thus also trees), which is an helpful technique for program verification. However, since most GTTs use their own graph format, it is difficult to import generated syntax trees.

Figure 3 shows the normal tool chain needed to create a

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
21th Twente Student Conference on IT June 21st, 2014, Enschede, The Netherlands.
Copyright 2014, University of Twente, Faculty of Electrical Engineering, Mathematics and Computer Science.

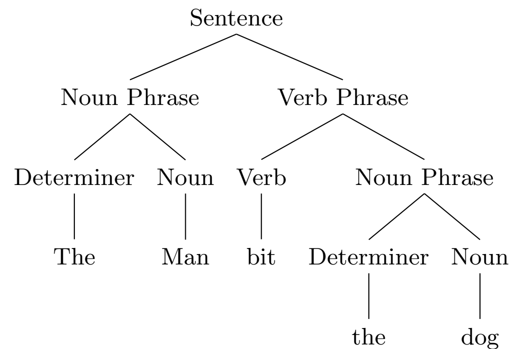


Figure 1: Syntax tree for natural language

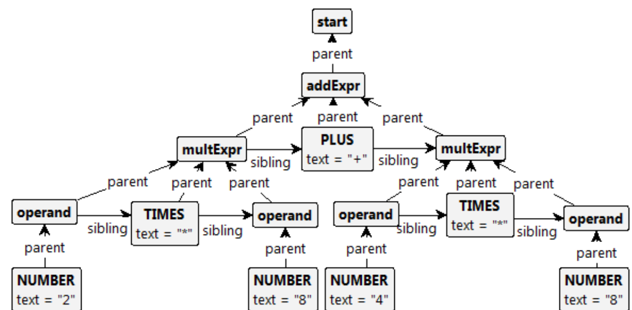


Figure 2: GROOVE syntax tree for 2*8+4*8

syntax tree and use it in a GTT, except for the lexer. The *lexer* turns an input text into a token stream; a sequence of tokens with information on each word in the input text. A *parser generator* takes a grammar and generates a parser for this grammar. A *parser* turns the token stream into a syntax tree (if the input text follows the rules of the grammar). However, to open this syntax tree with a GTT an extra conversion step must be performed, which slows down the workflow. Since GTTs have their own graph format a converter has to convert the syntax tree to the GTT’s graph format.

In this paper the following solution is studied: a parser which constructs a syntax tree using graph transformations. Subsequently it is shown how to generate this parser from the associated grammar, also using graph transformations. Both the parser and the parser generator have been created in the GTT named GROOVE [3]. By implementing the parser generating algorithm and parsing algorithm in GROOVE the converter has been made obsolete.

1.1 Research Scope

We assume that some preprocessing has already been done and the text has been converted to a token graph (which is a graph representation of a list of tokens) by a lexer.

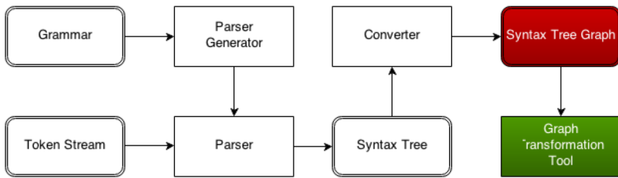


Figure 3: Existing architecture

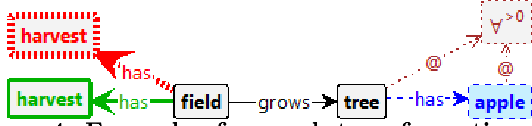
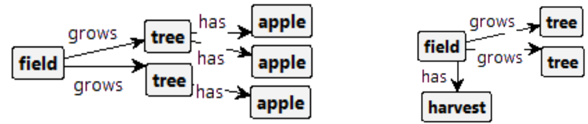
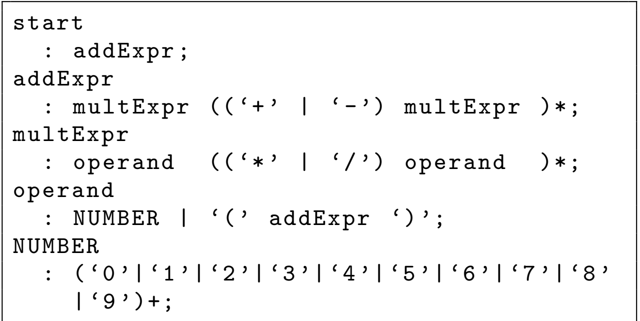


Figure 4: Example of a graph transformation rule in GROOVE



(a) Source Graph **(b) Target Graph**

Figure 5: Example application of the rule in Fig. 4



```
start
  : addExpr;
addExpr
  : multExpr  (('+' | '-') multExpr )*;
multExpr
  : operand  (('*' | '/') operand  )*;
operand
  : NUMBER | '(' addExpr ')';
NUMBER
  : ('0'|'1'|'2'|'3'|'4'|'5'|'6'|'7'|'8'
    |'9')+;
```

Figure 6: Grammar definition

There are multiple algorithms to form a syntax tree by parsing tokens; however, not all those algorithms may be compatible with graph formats. The algorithm must not only be compatible, but must also have a low (worst-case) time complexity, because the parser has to be able to parse large token graphs in a short time. Choosing the right algorithm was very important, because the grammar and parser generator are affected by the choice.

Since this research was aimed at using a GTT, it is assumed that all data is provided in graph format. This has been done for the tokens, as well as for the grammar.

2. BACKGROUND

2.1 Graph Transformation Tools

GTTs use *graphs* to represent the state of a computation and *graph transformation rules* to represent changes to states. Such rules have the form $G \rightarrow G'$ in which G is the graph that must be matched and G' is the transformed graph. One problem with graph matching is that it is NP-complete in the size of the graph to be matched [8], which could influence the time complexity of the graph transformation system.

2.1.1 GROOVE

An example of a graph transformation rule for GROOVE can be found in Figure 4. The example rule can be interpreted as "If a field has *no* harvest, remove *all* apples (at least one) from *all* the trees in the field and create a *new* harvest". An application of this rule can be seen in Figure 5, where 5a is the graph to be matched (G) and 5b the transformed graph (G'). Nodes with thin solid black lines will be matched, nodes with thick dashed red lines are forbidden, nodes with thin blue dashed lines will be matched and deleted and nodes with solid thick green lines will be created. The ∀ indicates that *all* possible matches of the connected nodes will be transformed, as opposed to regular nodes for which just one match will be made. Types are represented with bold text and flags (not present in this example) with italic text. Wildcards (also not present) are indicated with a question mark, e.g. **?x**, and can be used on node types and edges.

The type graph is, next to the input graph and graph transformation rules, another kind of input for GROOVE [3]. It specifies which nodes support which edges and flags using types. More information on the exact format of type trees can be found in Appendix A.

2.2 Grammar

A grammar definition is a formal grammar that specifies the syntax of a language. Figure 6 is an example of a grammar definition in EBNF [6] that can be used to correctly parse a mathematical expression. The grammar rules describe how to form sentences with the symbols used in the language. The symbols of grammar rules can be categorized into two types: terminals and nonterminals.

Terminals are the literal characters/strings that can be found in the text, and are shown in quotes (for example '+'). In the syntax tree they are used to label leafs of the tree.

Nonterminals are used to label the internal nodes in the syntax tree. They expand into other grammar rules (the `start` rule for example refers to `addExpr`).

2.3 Parser

A parser has a token stream as input, which is a chain of tokens with information on each symbol in the input text. The parser analyses the token stream and creates a syntax tree for the input if the input is valid according to the grammar definition. An unambiguous grammar will only have one associated syntax tree for a correct input [1].

There are multiple parsing algorithms that can construct a syntax tree. They differ on multiple points: parsing from left-to-right, right-to-left, bottom-up, top-down, in-place, side-by-side, etc. These aspects influence time complexity and grammar format.

2.4 Parser Generator

Since it is easier to write a grammar than a parser, it is a widely-used technique to use parser generators. A parser generator creates a parser with a certain parsing algorithm for the given grammar. One commonly used parser generator is ANTLR [5]. ANTLR creates parsers that use left-to-right parsing with a dynamic lookahead, also known as LL(*)-parser. Another well known parser generator is YACC, which creates LALR-parsers [4].

3. OUR APPROACH

Figure 7 shows the architecture of the solution presented in this paper. GROOVE takes two sorts of input for manipulating graphs: a set of start graphs (dark red rounded squares) and a set of graph transformation rules (blue parallelograms). For the parser generator this will be a grammar in graph format and three sets of rules that transform

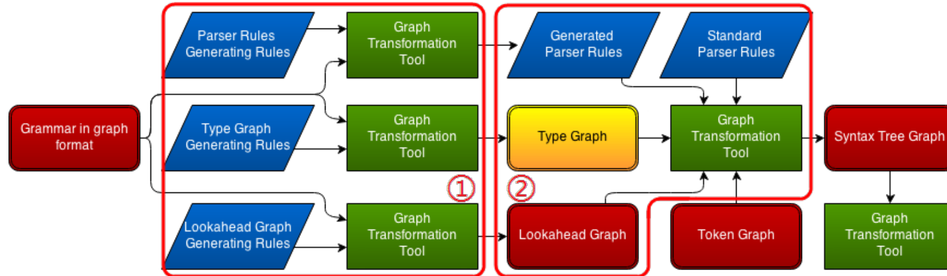


Figure 7: New architecture; ①parser generator ② parser

the grammar rules to parser rules, a lookahead graph and a type graph. The parser uses a set of common rules and a set of generated rules to transform a token graph and lookahead graph into a syntax tree. The parser also uses a type graph (yellow rounded square) to verify the graphs used while parsing.

Since the parser generator has to create a set of graph transformation rules, the GTT must be able to create graph transformation rules. This is possible in GROOVE, because it saves the rules in the same graph format as the input and output graphs [7].

4. THE PARSER

The parser uses a parsing algorithm to convert a token graph like the one in Figure 8 into a syntax tree like the one in Figure 2. It is an LL(1) parser, which is created using an LL(1) grammar. An LL(1) grammar has a unique grammar rule that can be applied for every parse decision. An LL(1) parser can parse sentences of an LL(1) grammar without backtracking and as such ensures a linear worst-case time complexity. Which grammar rule will be applied at a parse decision can be determined by looking at the first terminal it can produce and try matching it with the next token that must be parsed. To do this a lookahead table is calculated. This lookahead table is provided in graph format and is generated by the parser generator.

4.1 Graphs Used While Parsing

The parser has a start graph consisting of multiple disjoint subgraphs, namely a emphlookahead graph and a emph-token graph. The parser also uses a *type graph* which places constrains on the edges, types and flags in the used graphs. A *railroad graph* and a *syntax tree* are created as new disjoint subgraphs whilst parsing.

4.1.1 Lookahead graph

A lookahead graph maps grammar rules to the their first parseable terminal. This lookahead graph has a simplistic structure. Each nonterminal in the grammar has its own subgraph which has, at its center, a node with the name of the nonterminal as its type. From this node are edges pointing to the first terminals this nontermerminal can produce.

4.1.2 Token graph

The token graph is used to indicate which token must be parsed next and which tokens have not been parsed yet. The token graph is a linked list of token nodes which are, in principle, terminals with context information on the input text (e.g. line number, character number). Each token (except for the last) has an edge labelled "next" which points to the next node.

4.1.3 Type graph

The type graph determines the allowed combinations of node types and edges. An example of a type graph can be found in Figure A.1. The graphs used in the parser must *always* adhere to the defined graph formats in the type graph.

4.1.4 Railroad Graph

The railroad graph is used to track the state of the parser and shows which steps can be taken next. This graph only exists while parsing. Because this is an LL(1) parser only one grammar rule should be applicable at any time. This implies that there is only one subsequent state for each parser state. This railroad graph was inspired by the ANTLRWorks syntax diagram (commonly known as the railroad diagram) in which only the nodes are labelled [?].

The railroad graph has the following format. The nodes represent the terminals and nonterminals that can be parsed. They are connected in the sequence in which they can be parsed. The edges are labelled "next", "up" and "down" and indicate how the parser should traverse the syntax tree.

4.2 Syntax Tree

The output graph of the parser is a syntax tree. The tree will be constructed whilst parsing, just like the railroad graph. For each terminal and nonterminal node the parser visits in the railroad graph, a node is added to the syntax tree. Each node in the syntax tree has an edge to its parent, labelled "parent", except for the start node. Nodes in regular trees have ordered sets of children. Since GROOVE does not support node ordering in trees, children are ordered by using "sibling" edges, as shown in Figure 2. All tokens in the original token graph end up as leafs in the syntax tree, so they still contain all the information of the original tokens.

4.3 Graph Transformation Rules

The parser uses two sets of graph transformation rules: common rules and generated rules. The common rules are used to traverse the syntax tree and railroad graph, while simultaneously constructing the syntax tree. They are *not generated* because they are the same for every parser. The generated rules are used to build tracks (i.e. paths) in the railroad graph. They extend the railroad with a grammar-rule-specific graph.

4.3.1 Common Rules

Fundamentally a parser needs three actions to traverse a syntax tree, namely: going to the first child of a node (*down*), going to the parent of a node (*up*) and going to the next sibling of a node (*next*). In our implementation these three actions are used by the common rules and can be found in the railroad graph's edges.

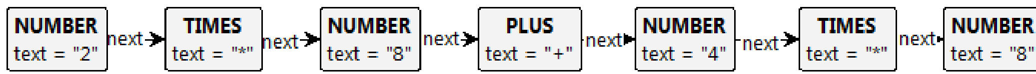


Figure 8: A token stream in graph format

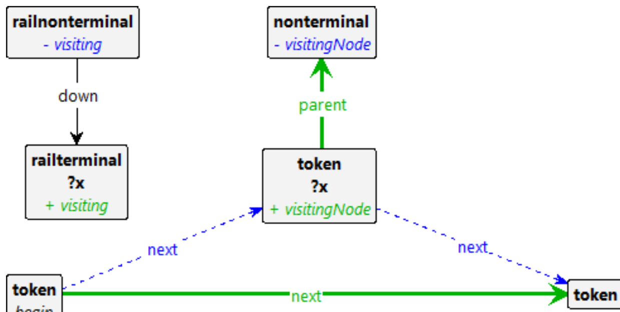


Figure 9: The 'downAndAccept' rule in GROOVE

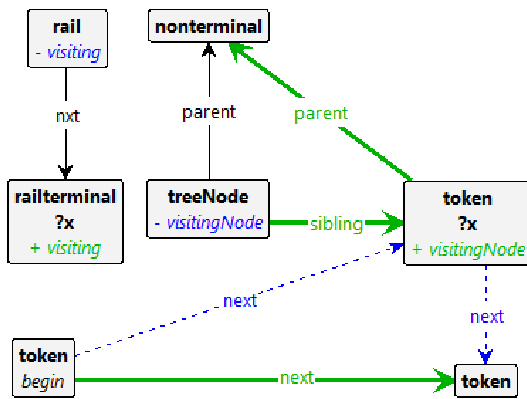


Figure 10: The 'nextAndAccept' rule in GROOVE

The parser has the following common rules.

1. init
2. finish
3. downAndAccept
4. nextAndAccept
5. down
6. next
7. up

Since the parser looks one node ahead, the next symbol it has to match can be a nonterminal as well as a terminal. That is why the *up*, *down* and *next* actions have been created for terminals (rules 3 and 4) and nonterminals (rules 5, 6 and 7). Furthermore, there are "init" and "finish" rules.

Ad rules 3 and 4: there is no "acceptAndUp" rule, because "up" edges in the railroad graph can only point to nodes with the type "done". These "done" nodes are not terminals and therefore cannot be accepted.

The implementation of rules 'downAndAccept' to 'up' can be found in Figure 9 to 13. How these rules work together can be seen in the parsing algorithm shown in Algorithm 1 in Appendix 9.

The 'init' rule creates a starting point for the railroad graph and syntax tree. Furthermore, it adds a begin token, which points to the first unparsed token in the token graph, and an end token, which the last token in the token graph points to. An empty token graph can be detected with ease, because the begin token will point to the end token.

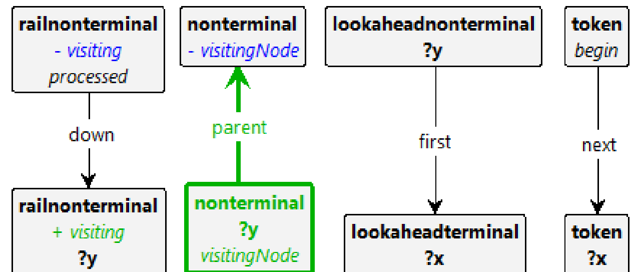


Figure 11: The 'down' rule in GROOVE

The 'finish' rule ensures only the syntax tree remains, by deleting the empty token graph, the railroad graph and the lookahead graph. This rule can only be applied if the railroad graph gives the parser the "finish" option.

The 'downAndAccept' rule can be applied when the parser just started parsing a nonterminal and has to parse the first symbol of the nonterminal's grammar rule, which is a terminal. This means that the parser now has to add this terminal as a new child in the syntax tree.

The 'nextAndAccept' rule can be applied when the parser just parsed a symbol and has to parse a terminal subsequently. This means that the parser now has to add this terminal as a sibling of the previously parsed symbol in the syntax tree.

The 'down' rule can be applied when the parser just started parsing a nonterminal and has to parse the first symbol of the nonterminal's grammar rule, which is a nonterminal. This nonterminal however, should have the token that must be parsed next in its lookahead. The parser will add this nonterminal as a new child in the syntax tree.

The 'next' rule can be applied when the parser just parsed a symbol and has to parse a nonterminal subsequently. This nonterminal however, should have the token that must be parsed next in its lookahead. The parser will add this nonterminal as a sibling of the previously parsed symbol in the syntax tree.

The 'up' rule completes the parsing of the current nonterminal and returns the focus to the parent nonterminal. This rule has a lower priority to ensure the parser does not stop parsing the current grammar rule until it cannot accept any more tokens.

If the 'up' rule would have the same priority as the other rules the parser would do unnecessary backtracking, because it cannot see which actions it can take after parsing the current nonterminal. This also means it would be able to create multiple syntax trees for ambiguous grammars.

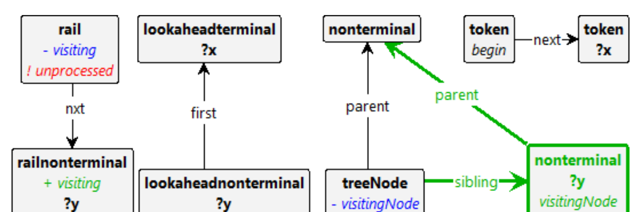


Figure 12: The 'next' rule in GROOVE

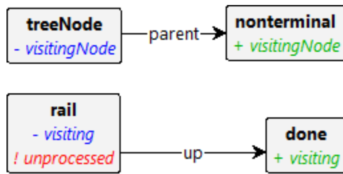


Figure 13: The 'up' rule in GROOVE

4.3.2 *The Generated Rules*

The generated rules extend the railroad when the parser visits an unprocessed nonterminal. They inject a partial railroad graph (derived from the associated grammar rule of the nonterminal) between the nonterminal and its subsequent node. That way the parser has to parse the nonterminal first before continuing. In this railroad graph all nonterminals carry an "unprocessed" flag; this indicates that the nonterminal has not been visited yet. An example of the generated rule for the "operand" nonterminal can be found in Figure 14.

5. THE PARSER GENERATOR

The parser generator consists of three parts: a parse rule generator, a lookahead graph generator and a type graph generator.

5.1 Grammar Graphs

The generators all take a grammar in graph format as input. Each rule of the grammar has its own graph. At the start of the graph is a node with the name of the grammar rule as its type. It is connected with a "to" edge to a chain of nonterminals and terminals which are connected with "next" edges, depicting the order in which they can be parsed. The last (non)terminals are connected to a node with only a "done" flag to indicate that the parser may stop parsing this grammar rule after that (non)terminal. The (non)terminals have their name as their type. The terminals must also have a "terminal" flag, to enable the parser to distinguish them from nonterminals. An example of a grammar rule can be found in Figure 15.

5.2 Output Graphs

The output of the parse rule generator is a graph transformation rule as depicted in Figure 14 for each grammar rule. The output graph of the lookahead graph generator is a lookahead graph as depicted in section 4.1.1. The output graph of the type graph generator is a type graph as depicted in Figure A.1.

5.3 Parse Rule Generation

The parser generating algorithm transforms the graph in Figure 15 to the one in Figure 14. To create the parse rules a grammar graph is transformed as follows. The "to" edge is replaced by a "down" edge and the incoming "next" edges at the node flagged "done" are replaced by "up" edges. This informs the parser when to change the depth in the syntax tree. Every node except the first one gets a "new:" attribute to indicate that these nodes must be added to the railroad graph. The "$\forall$" and "rail" node and their edges are added. This way the generated rule will reconnect every outgoing edge of the nonterminal that is being processed to the "done" node. Lastly the "terminal" flags are removed and "unprocessed" flags are added to the nonterminal nodes.

One restriction in this format is the absence of empty rules (also known as λ-rules), because the "down" and "up" edge are both necessary for the parser to work correctly. Fortunately these empty rules can be mimicked by making the nonterminals that refer to them optional in the other grammar rules. This process is called *λ-rule elimination* [9].

5.4 Lookahead Graph Generator

Creating the lookahead graph is done by loading all the grammar graphs together as disjoint subgraphs and comparing the nodes with incoming "to" edges. These nodes are the first symbols of the grammar rules. If a grammar rule starts with a terminal, it is added to the lookahead of that grammar rule. If a grammar rule starts with a nonterminal which has a terminal in its lookahead that has not been added yet, the terminal is added to the lookahead of the grammar rule as well. These steps are repeated until all terminals are added.

It is possible as well to use this lookahead graph in combination with the grammar graphs to detect LL(1) errors. However, this has not been implemented because the parser now supports LL(k) grammars by using backtracking, which could be obstructed by LL(1) errors.

5.5 Type Graph Generator

The type graph generator loads all grammar graphs together as well. It adds the nonterminals (the first node of each grammar graph) as a subtype of the rail nonterminal, the syntax tree nonterminal and the lookahead nonterminal. It also adds the terminals (which are indicated with "terminal" flags in the grammar graphs) as a subtype of the rail terminal, token and lookahead terminal. The token terminal in turn, is a subtype of the syntax tree terminal, which makes it possible to convert tokens to syntax tree terminals. An example of a generated type graph can be found in Figure A.1.

6. VALIDATION

6.1 Correctness

Multiple scenarios have been tested to validate the parser:

1. Parsing a correct sentence
   (a) using an LL(1) grammar
   (b) using an LL(k) grammar with $k > 1$
2. Parsing an incorrect sentence
   (a) a sentence with an incorrectly placed token
   (b) a sentence with a missing (trailing) token
3. Parsing an ambiguous sentence (with an ambiguous grammar)

Scenario 1.a, 2.a and 2.b have been tested with the grammar in Figure 6, while scenario 1.b has been tested by adding the following rule to that grammar:

```
modulo : NUMBER '%' NUMBER;
```

The `multExpr` rule was replaced by the following rule:

```
multExpr: operand (('*'|'/') operand)*
        | modulo  (('*'|'/') modulo )* ;
```

That way an LL(1) error occurs when parsing a `multExpr`, since the `operand` and `modulo` rules can both begin with the terminal `NUMBER`. The last scenario was tested with a small custom grammar.

6.1.1 *Results*

When testing scenario 1.a and 1.b the parser correctly created a syntax tree. In scenario 1.b however, the parser had to backtrack after trying to parse incorrect grammars rule at LL(1) errors.

In scenario 2.a the parser stopped parsing at the misplaced token, because it could not perform any further actions. In scenario 2.b the parser accepted all the tokens, but

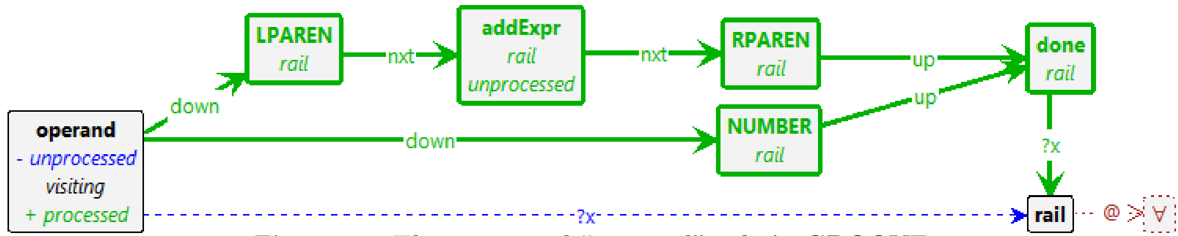


Figure 14: The generated "operand" rule in GROOVE

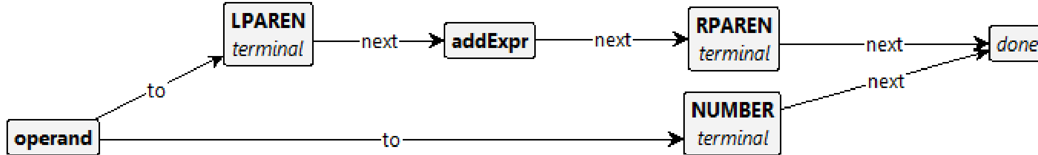


Figure 15: The grammar rule for "operand = NUMBER | LPAREN addExpr RPAREN" in GROOVE

could not apply the 'finish' rule because this option was not given by the railroad graph.

Scenario 3 was tested with the well-known *dangling else* problem [2], which can be illustrated with the following code:

```
if (a) if (b) s1; else s2;
```

The `else` can belong to the first, but also to the second `if`. That means there are *two* correct syntax trees for this input. The parser created a single syntax tree in this scenario. However, both possible syntax trees could be created by giving the 'up' rule a higher priority. In the created syntax tree the `else` was paired with the second `if`, because the parser tries to parse as many tokens as possible before returning control to the parent nonterminal (which in this case is the first `if`).

6.2 Time Performance

Time performance has been tested with an LL(1) grammar of a small programming language, consisting of 22 grammar rules. Firstly a parser was generated for this grammar. Subsequently multiple token graphs of different sizes were parsed while tracking the time and parser states it took for the parser to construct the syntax trees. The same inputs were parsed with a parser generated by ANTLR, for comparison. To ensure the resulting parsing time is consistent in its input, the same input has been replicated multiple times, i.e. "2+2;", "2+2;2+2;", etc. This assures a consistent ratio between tokens and parser states, since the parser has to apply the same parse rules every time. The results of these tests can be seen in Table 1 and 2.

Table 1: Measurements of the performance of the parser

tokens	syntax tree nodes	parser states	GROOVE parse time [ms]	ANTLR parse time [ms]
2	13	39	449	5
4	26	61	616	5
6	38	83	744	5
8	50	105	964	6
12	74	149	1461	6
24	146	281	3410	6
48	290	545	11081	6
96	578	1073	47546	8

Table 2: Measurements of the performance of the parser

tokens	syntax tree nodes	parser states	GROOVE parse time [ms]	ANTLR parse time [ms]
6	11	22	589	5
12	20	34	998	5
24	29	58	2635	6
48	74	106	8652	6
96	146	202	42189	6

7. CONCLUSIONS

It is possible to create a parser for a GTT using the algorithm provided in this research. The generated parser is deterministic: it only creates *one* syntax tree for an ambiguous grammar. This prevents unnecessary backtracking and as such improves time performance.

Even though the parser uses a parsing algorithm with a linear worst-case time complexity (for LL(1) grammars), it still runs in exponential time (cf. linear time with ANTLR). One of the reasons could be the complexity of the graph matching that is performed by GROOVE, which is NP-complete (in the size of the graph to be matched) [8]. While ANTLR has pointers to the parser state (which is saved on the program stack), GROOVE has to find the state of the parser in the railroad graph with a graph match before applying any graph transformation rule.

8. FUTURE WORK

To improve the time performance changes can be made to the parser. The unused parts of the railroad could be removed, for example. This would reduce the graph matching time, because of the reduction in the graph size. This would have a big impact on the time performance, since the railroad graph is the largest graph used by the parser.

Another welcome change would be the use of empty (λ) rules. To implement this the lookahead graph generator must also be adapted, since it only checks the first symbol of a rule, which could then be empty.

9. ACKNOWLEDGEMENTS

The author would like to thank Arend Rensink for proposing the idea of using a railroad graph as well as for the guidance throughout the research process.

10. REFERENCES

[1] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, & Tools.* Addison-Wesley, 1988.

[2] C. Clark. What to do with a dangling else. *SIGPLAN Not.*, 34(2):26–31, Feb. 1999.

[3] A. H. Ghamarian, M. J. de Mol, A. Rensink, E. Zambon, and M. V. Zimakova. Modelling and analysis using GROOVE. *International journal on software tools for technology transfer*, 14(1):15–40, February 2012.

[4] S. C. Johnson. Yacc: Yet Another Compiler Compiler. In *UNIX Programmer's Manual*, volume 2, pages 353–387. Holt, Rinehart, and Winston, New York, NY, USA, 1979.

[5] T. J. Parr and R. W. Quong. ANTLR: A predicated-$LL(k)$ parser genarator. *Softw: Pract. Exper.*, 25(7):789–810, July 1995.

[6] R. E. Pattis. Teaching EBNF first in cs 1. In R. Beck and D. Goelman, editors, *SIGCSE*, pages 300–303. ACM, 1994.

[7] A. Rensink. The GROOVE simulator: A tool for state space generation. In J. L. Pfaltz, M. Nagl, and B. Böhlen, editors, *Applications of Graph Transformations with Industrial Relevance (AGTIVE)*, volume 3062 of *Lecture Notes in Computer Science*, pages 479–485, Berlin, 2004. Springer Verlag.

[8] A. Rensink. Time and space issues in the generation of graph transition systems. *Electronic Notes in Theoretical Computer Science*, 127(1):127 – 139, 2005. Proceedings of the International Workshop on Graph-Based Tools (GraBaTs 2004) Graph-Based Tools 2004.

[9] T. Sudkamp. *Languages and machines - an introduction to the theory of computer science.* Addison-Wesley series in computer science. Addison-Wesley, third edition, 1988.

APPENDIX

A. TYPE GRAPH

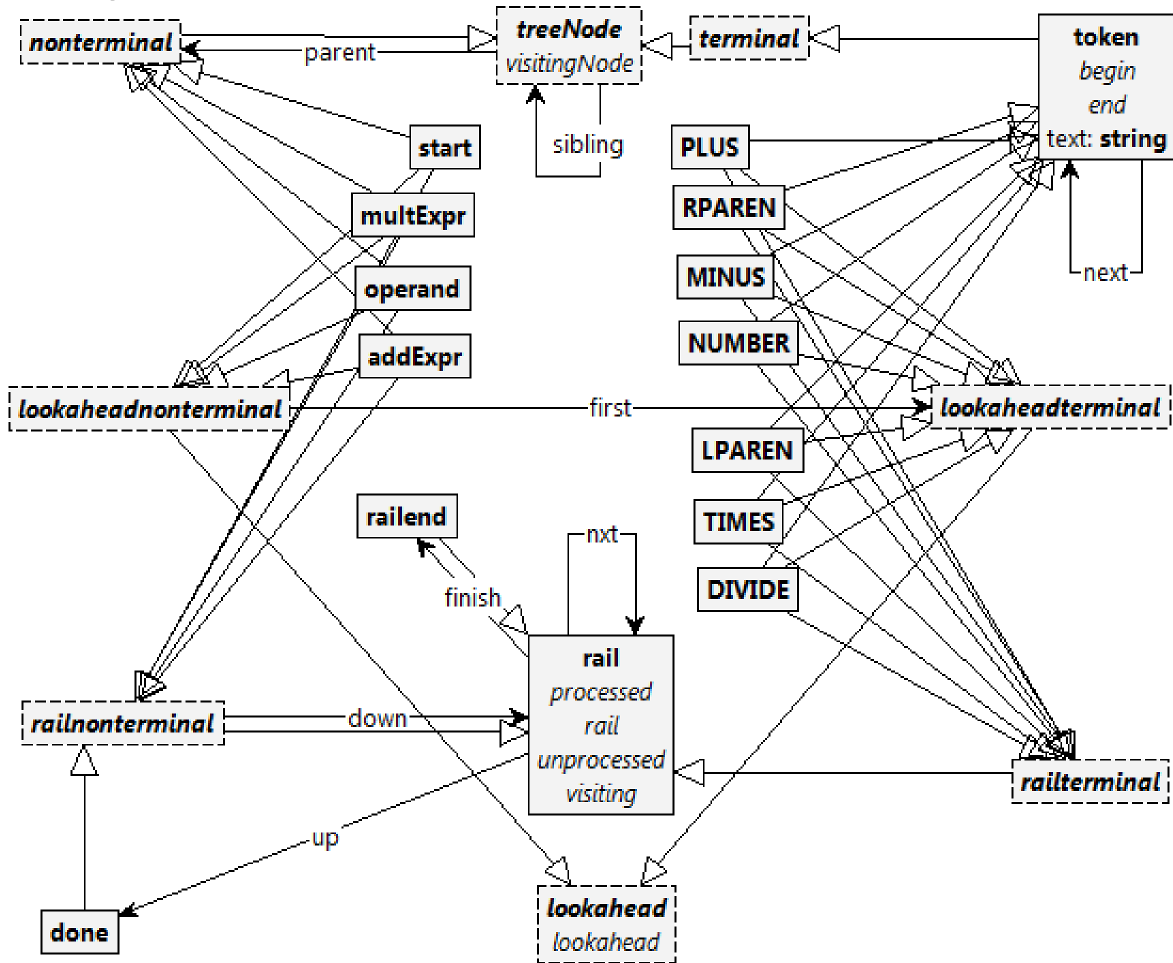


Figure A.1: Example of a generated type graph

Figure A.1 shows an example of a type graph in GROOVE. This type graph matches the grammar in Figure 6, but the terminals have been given a name ('+' is now called **PLUS** for example).

GROOVE's type graph has the following format. Nodes with dotted lines are abstract types and cannot be instantiated, but they can be subtyped. Flags are, as before, shown in italic and types in bold text. The arrows with hollow heads indicate supertypes. The arrows with filled heads are the allowed edges between nodes. The "text: **string**" indicates that a token can have a text-attribute of type **string**.

The nonterminals and terminals have multiple abstract supertypes which are necessary to compare them. Nonterminals can be found in the lookahead graph, syntax tree and railroad graph. The concrete types of the nonterminals in the example are **start**, **multExpr**, **operand** and **addExpr**. Terminals can be found in the lookahead graph, syntax tree, railroad graph and token graph. The concrete types of the terminals in the example are **PLUS**, **RPAREN**, **MINUS**, **NUMBER**, **LPAREN**, **TIMES** and **DIVIDE**.

If the parser wants to check that a railroad terminal has the same type as token, for example when accepting a token, it has to match them using their types. This is done by wildcards in the parser rules, which only match the concrete types.

B. The Parsing Algorithm

```
input  : token graph, lookahead graph, type graph, parser rules
output: syntax tree
add "start" node flagged "visitingNode" as syntax tree;
add "start" node flagged "unprocessed" and "visiting" as railroad graph;
while token graph not empty do
    if "visiting" flag on unprocessed node then
        process node by expanding it with associated railroad graph of the following format:
        --down--> symbol( --next--> symbol)* --up--> done
        where symbol is a terminal or an unprocessed nonterminal. All previous outgoing edges of the node now begin
        at the new done node;
    else
        if next node in railroad graph is a terminal and type matches first token in token graph then
            if outgoing edge to next railroad node is labelled "down" then
                remove token from token graph and place as leaf in syntax tree as child of tree node labelled
                "visitingNode";
            else if outgoing edge to next railroad node is labelled "nxt" then
                remove token from token graph and place as leaf in syntax tree as sibling of tree node labelled
                "visitingNode";
            end
            move "visitingNode" flag to added leaf;
        else if next node in railroad graph is a nonterminal and lookahead of nonterminal includes first token then
            if outgoing edge to next railroad node is labelled "down" then
                add nonterminal in syntax tree as child of tree node labelled "visitingNode";
            else if outgoing edge to next railroad node is labelled "nxt" then
                add nonterminal in syntax tree as sibling of tree node labelled "visitingNode";
            end
            move "visitingNode" flag to added node;
        else if outgoing edge to next railroad node is labelled "up" then
            move "visitingNode" flag in syntax tree to parent of visited node;
        end
        move "visiting" flag in railroad to next node ;
    end
end
delete railroad graph and lookahead graph;
remove "visitingNode" flag from syntax tree;
```

Algorithm 1: The Parsing Algorithm