Differencing Context-free Grammars

Randy Fluit

August the 30^{th}, 2011

Master Software Engineering

Thesis Supervisor : Tijs van der Storm

Internship Supervisor: Jurgen Vinju

Company: CWI

Availability: Public domain

University of Amsterdam

Contents

Preface

This thesis describes my research project, which concludes the Master's degree of Software Engineering at the University of Amsterdam. The research has been performed at the Centrum Wiskunde & Informatica in Amsterdam.

Acknowledgments

Several people contributed to the successful completion of the research project and this thesis. I would therefore like to express my appreciation for their help. First of all, I would like to thank my supervisor Tijs van der Storm for his support and guidance. Furthermore I would like to thank the researchers at CWI, especially Jurgen Vinju, Paul Klint and Vadim Zaytsev for their assistance. Finally I would like to thank the people of CWI and the master Software Engineering at the UvA for making this possible.

Abstract

Grammars form an integral part of grammarware. When a grammar evolves, the grammarware has to evolve with it. In this process, the exact changes in the grammar have to be found and changes have to be made to parts that rely on the changed grammar parts. This thesis presents an algorithm, *cfgDiff*, which can find the difference between two context-free grammars. Using a model which handles both the ordered and unordered elements of a grammars, the algorithm can provide concise edit scripts.

Chapter 1

Introduction and Motivation

When grammars evolve, the grammarware needs to be adapted to reflect these changes. Although grammars have been around for decades, grammarware evolution still is a manual process.

The foundations of grammar-dependent software and grammatically structured data lay within the grammar. If a grammar changes, they probably need to be modified as well [10]. To see which parts of the grammarware need to be modified, the engineer needs to be able to see what modifications have been made to a grammar. Diff tools can be used to compare two grammars and mark any changes.

1.1 Context

The CWI developed Rascal[1] as a domain specific language for source code analysis and manipulation. To facilitate source code analysis, Rascal has his own formalizations to define grammars.

To ease the development of grammars in Rascal, the Rascal team at the CWI wants to create an integrated development environment for grammars in Rascal. Part of this project is a tool which can calculate the difference between two context-free grammars. Although Vadim Zaytsev already developed a tool to difference two grammars, it can only output a list of non-matching non-terminals.

[1]http://www.rascal-mpl.org/

1.2 Problem definition

Although the difference between two context-free grammars can be calculated by a text-based diff algorithm, such as GNU diff[2], the results of this comparison will presumably yield very little information about the semantic difference between the two grammars. This is caused by the way text-based diff algorithms operate: they difference the files on a line-to-line basis. Because context-free grammars are written down in a human readable form, characters like whitespacing are inserted to make the grammar more readable and are semantical irrelevant. More importantly, text-based diff algorithm disregard the hierarchical structure of context-free grammars since it is designed to compute the difference on a character to character basis. Since some parts of a context-free grammar are unordered lists, rearranging these lists can result in a lot of semantically irrelevant changes. Furthermore, these changes are based on the difference between a textual representation of the two grammars and may be difficult to interpret.

To counteract the shortcomings of text-based diff algorithms, a tree based diff algorithm can be used. A context-free grammar can be structured as a tree. By doing so all formatting can be ignored. Also, a tree based diff algorithm will yield much less semantically irrelevant changes. Although tree diff algorithms for ordered and unordered trees exist in many forms, none can handle both ordered and unordered lists mixed within the same tree, like they occur in context-free grammars. This occurrence of both ordered and unordered lists in context-free grammars are caused by the associativity of context-free grammar constructs. For some constructs the ordering of its children is irrelevant, for example in a choice. In other constructs ordering is important, because the order of its children reflect the order in which the subproductions should occur. Since no existing tree algorithm allows the combination of both ordered and unordered lists within a tree, a new diff algorithm has to be developed.

1.2.1 Scope

A lot of data formats and notations to represent context-free grammars exist. To develop a tool which can read all these notations is a resource intensive operation. Therefore the scope of this research is limited to the notation of context-free grammars Rascal uses.

Furthermore this research is limited to detecting changes in the tree structure of the context-free grammar. The transformation of tree edit operations into grammar edit operations will not be part of this research.

1.2.2 Research question

Based on the problem definition, the following research question has been formulated:

[2]http://www.gnu.org/s/diffutils/

How can a context-free grammars be differenced?

In order to answer this questions I will explore existing diff algorithms and their shortcoming when they are applied to context-free grammars. Based on these findings a diff algorithm for context-free grammars will be presented and evaluated.

1.3 Approach

1.3.1 Literary study

At first a literary study will be conducted on existing differencing algorithms. This research will focus on how existing algorithms work and what they do to work effective and efficient and their applicability on context-free grammars will be researched.

1.3.2 Developing *cfgDiff*

At this point it has been established that existing tree diff algorithms have trouble differencing context-free grammars. Unordered tree diff algorithms can not detect all the necessary types of changes made to a tree and ordered tree diff algorithm can detect irrelevant changes. Therefore a new tree diff algorithm, *cfgDiff*, has to be designed. An existing tree diff algorithm may serve as a basis for this new algorithm. If a suitable algorithm has been found, it will be modified into *cfgDiff* using the following steps:

- Rewrite the algorithm in Rascal[3].
- Add code to load and preprocess the grammars.
- Define the cost model and edit operations.
- Modify the algorithm to work on grammar data types.
- Test and improve the algorithm.

If adapting an existing algorithm isn't viable, developing a new algorithm may be opted. Factors which can contribute to this problem are unreadable or lacking source code, incomplete pseudo code listing or descriptions and approach or lack of certain edit operations may yield unsuitable algorithm.

[3]Parts of the algorithm may be skipped as they will be replaced in later steps.

Testing *cfgDiff*

Once parts of the *cfgDiff* algorithm are written, it can be tested incrementally. At first only one kind of change operation will be applied to a random tree structure at a time. This way, we can easily verify that each type of change is recognized correctly by the algorithm.

The algorithm will be validated with two tests. The first test checks the edit cost. The cost of the edit script should be equal to or smaller than the cost of the generated changes. If this is true we can apply the edit script to the original tree. If the edit script is applied successfully, the resulting tree should match the tree with the generated changes.

If these tests succeed for all edit operations on their own, combinations of edit operations can be applied to the tree. These changes can be validated with the same tests as used before.

1.4 Outline

Chapter 2 describes existing work, its shortcomings and a introduction to grammars in Rascal. Chapter 3 describes the algorithm in detail. Chapter 4 gives an overview of the capabilities of the algorithm using test result. Chapter 5 provides an overview of the difference between the algorithm and XDiff. Chapter 6 concludes with weaknesses and trade offs in the algorithm and hints for future work.

Chapter 2

Background and Context

Before addressing the research question of this thesis, existing literature needs to be reviewed to be able to understand the problem. This overview will address related work and its shortcomings.

2.1 Tree diff algorithms

Tree diff algorithms are a well researched subject. However, all research are focused on ordered [26, 4, 19, 15] or unordered trees [20]. No prior work has been found on trees which contain both ordered and unordered lists. Some of these algorithms are specialized for certain problems by making certain assumptions, for instance algorithms that diff labeled trees [26, 19, 15], XML [4, 20] or UML [7, 22] documents. Other algorithms are optimized for certain fields, for example by sacrificing accuracy for faster run times [4, 19].

Most of these algorithms work by traversing the tree bottom up [26, 4, 20]. Using this approach a set containing the leaf nodes is composed for each tree. The algorithm tries to match the nodes in these sets with each other by finding the edit distance between the nodes. When all nodes are processed, the sets are replaced by the set of parents of all nodes currently in the sets. The algorithm than tries to match these subtrees with each other, utilizing the previously calculated edit distance between the children. This process is repeated until the distance between the roots of both trees is calculated.

2.1.1 Related work

As mentioned earlier, related work exists. However, each of these algorithms have shortcomings or restrictions, making it hard or impractical to use the algorithm as a grammar differencing algorithm. Although this overview is incomplete, it gives a good summary of the related work.

The tree-to-tree editing problem[15]

The tree-to-tree edit problem by S. M. Selkow presents an algorithm to calculate the distance between two labeled trees using an ordered model. The algorithm builds a distance table while recursively going deeper into the tree, without trying to find matching subtrees to reduce the matching space. The algorithm supports inserting and deleting subtrees and the relabeling of labels.

X-Diff: an effective change detection algorithm for XML documents [20]

X-Diff: an effective change detection algorithm for XML documents by Wang et. al presents an algorithm to calculate an edit script between two XML documents using an unordered model. The algorithm tries to remove identical subtrees using a bottom-up approach to reduce the matching space. It also limits the matching of nodes to nodes with the same signature (the name of the XML elements). It supports inserting and deleting subtrees/leafs, relabeling of attributes and changing the values of leaf nodes. Although the algorithm is very quick, this is mainly so because it uses early rejection on nodes with different names. Since the names of nodes in a context-free grammar describe the action applied to its contents, this limits the applicability of this algorithm for grammars.

Detecting changes in XML documents [4]

Detecting changes in XML documents by Cobena et al. presents an algorithm to calculate an edit script between two XML documents using an ordered model. Since it has been developed for use in a XML data warehouse, the algorithm sacrifices accuracy in exchange for faster run times. The algorithm uses a priority queue to determine which unmatched subtree is the heaviest, so a match can be found for this subtree first. If no match is found, the subtree is split up into its subtrees and inserted in the priority queue.

Vadim Zaytsev's grammar diff tool

Vadim Zaytsev build a simple grammar diff tool for his work on grammar convergence. Although it supports multiple file formats as input grammars, the output of the tool is very limited. The tool checks if each non-terminal is defined in both grammars, if this is not the case a message is printed. For all non-terminals that are defined in both grammars, their productions are compared. If they do not match, the tool simply prints

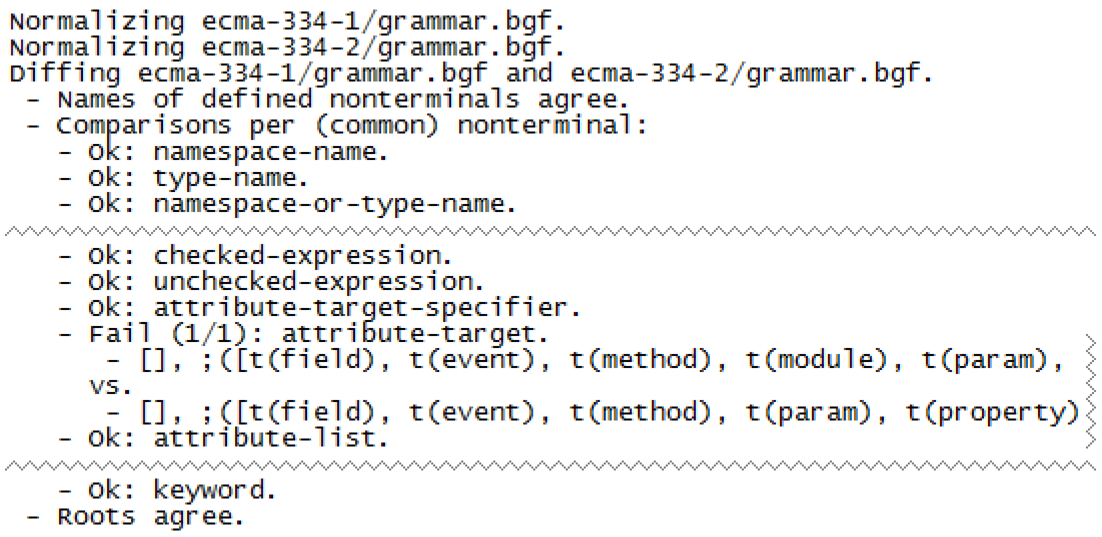

```
Normalizing ecma-334-1/grammar.bgf.
Normalizing ecma-334-2/grammar.bgf.
Diffing ecma-334-1/grammar.bgf and ecma-334-2/grammar.bgf.
 - Names of defined nonterminals agree.
 - Comparisons per (common) nonterminal:
   - Ok: namespace-name.
   - Ok: type-name.
   - Ok: namespace-or-type-name.
   - Ok: checked-expression.
   - Ok: unchecked-expression.
   - Ok: attribute-target-specifier.
   - Fail (1/1): attribute-target.
     - [], ;([t(field), t(event), t(method), t(module), t(param),
   vs.
     - [], ;([t(field), t(event), t(method), t(param), t(property)
   - Ok: attribute-list.
   - Ok: keyword.
 - Roots agree.
```

Figure 2.1: Excerpts from Vadim Zaytsev's grammar diff tool.

the name of the non-terminal and both productions. No attempts are made to find renamed non-terminals or to pinpoint differences between non matching productions.

2.1.2 Shortcomings in existing work

As mentioned in section 1.2, existing diff algorithms have inherent shortcomings when they are applied to context-free grammars. Text-based diff algorithms ignore the structure of context-free grammars and are sensitive to changes in white spacing and reordering.

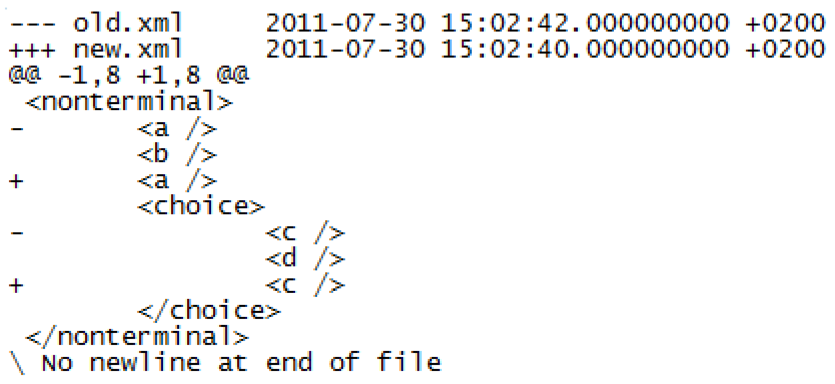
Although unordered tree diff algorithms respect the structure of the file format, it ignores any changes in the ordering of elements. Ordered tree diff algorithms respect the structure too, but treat any unordered elements as ordered which increases the size of the resulting edit script.

These shortcomings can be illustrated by the following minimal example (described using EBNF):

nonterminal = A B (C | D) ;

Which will be represented using the following XML:

```
<nonterminal>
        <A />
        <B />
        <choice>
```



```
--- old.xml     2011-07-30 15:02:42.000000000 +0200
+++ new.xml     2011-07-30 15:02:40.000000000 +0200
@@ -1,8 +1,8 @@
 <nonterminal>
-       <a />
        <b />
+       <a />
        <choice>
-               <c />
                <d />
+               <c />
        </choice>
 </nonterminal>
\ No newline at end of file
```

Figure 2.2: The difference between both XML document according to GNU diff 2.8.1.

```
            <C />
            <D />
        </choice>
</nonterminal>
```

Which is changed to the following non-terminal to difference against:

nonterminal = B A (D | C) ;

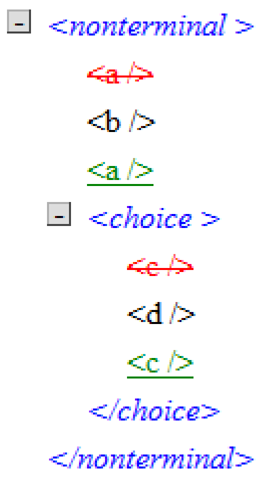
Represented by the following XML:

```
<nonterminal>
        <B />
        <A />
        <choice>
            <D />
            <C />
        </choice>
</nonterminal>
```

The semantical difference between the two grammars is the reordering of A and B. The reordering of C and D are irrelevant, since the are part of an unordered choice.

Text-based diff algorithms

Text-based diff algorithms lack important information about the exact changes between the two grammars. Instead, they only specify which lines need to be added or removed from the original file to make the new file and do not specify which line replaces which line. Because of this, text-based diff algorithms only point out which lines have not been



```
<nonterminal >
    ~~<a />~~
    <b />
    <a />
  <choice >
      ~~<c />~~
      <d />
      <c />
    </choice>
  </nonterminal>
```

Figure 2.3: The difference between both XML documents according to DeltaXML (http://www.deltaxml.com). DeltaXML, an online ordered tree diff tool, was used because I was not able to run XyDiff.

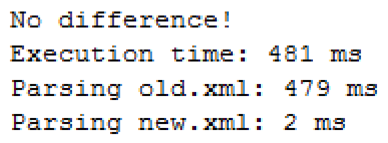
changed between the two files. No effort is put into finding the closest match for a line or to pinpoint which parts of the lines have been changed.

Text-based diff algorithms detect the changes made to the ordering of A and B, and C and D in the example since they work on an ordered model. As seen in figure 2.2, the edit script contains the deletion of <A />, followed by inserting it back after <B />. The same changes are made for <C /> and <D />.

Ordered tree diff algorithms

Ordered tree diff algorithms are better at pinpointing differences between two grammars. Because tree diff algorithms work on nodes instead of whole lines, changes can be detected at a much smaller scale than text-based diff algorithms do. Since most lines in a grammar will consist of multiple nodes, parts of a line can be identified as changed instead of the whole line. Most tree diff algorithms also support a replacement operation, for which the algorithm tries to match changes in both files with each other. This provides more insight about the changes that have been made to the files.

Ordered tree diff algorithms, like DeltaXML detect the same changes for this example as text-based diff algorithms do (see figure 2.3). This is because they use the same ordered model, although tree based diff algorithms work on the tree representation of the XML document instead. An advantage of unordered XML diff algorithms over text-based diff algorithms is that they ignore any changes in whitespacing around XML elements.



```
No difference!
Execution time: 481 ms
Parsing old.xml: 479 ms
Parsing new.xml: 2 ms
```

Figure 2.4: The difference between both XML document according to XDiff.

Unordered tree diff algorithms

Unordered tree diff algorithms go one step further than ordered tree diff algorithms by treating all children of a node as an unordered collection. This results in better matchings for the parts of a grammar where the order of elements is irrelevant, since there is no penalty involved when elements are reordered. On the other hand it can rearrange the ordered parts of a grammar because changes are found between the closest matching pairs of elements without taking the order of the elements into account.

X-Diff[20], an unordered tree diff algorithm, does not detect any changes made in the example (see figure 2.4). This is because all nodes still contain the same children, only their ordering is changed. In the unordered model, reordering of children is ignored.

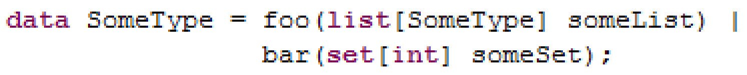
The ideal algorithm

The ideal algorithm will only recognize the reordering of the ordered A and b. The reordering of C and D will be ignored since they are unordered.

2.2 Grammar convergence

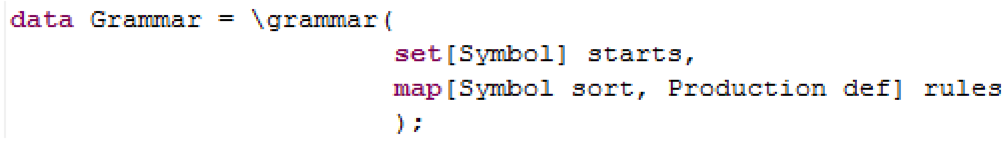
The end result of Grammar convergence [25] is comparable to an edit script. Although tools exists to aid in the converging process, no known tool can automatically convergence two grammars. Therefore a grammar expert has to execute the converging process manually by finding a difference between the two grammars and find a transformation to describe this difference. This process is time consuming and requires an expert to succeed.

Although grammar converging results in an edit script, it is written by the grammar engineer himself in a manual process. Therefore, grammar convergence can not be seen as a replacement for a grammar diff tool because the grammar convergence process requires such a tool aid in the process.



```
data SomeType = foo(list[SomeType] someList) |
                bar(set[int] someSet);
```

Figure 2.5: An example of a data type with two constructors in Rascal. The data type *SomeType* contains the constructor *foo* (consisting of a *list* of *SomeType* named *someList*) and *bar* (consisting of a *set* of *int* named *someSet*).



```
data Grammar = \grammar(
                        set[Symbol] starts,
                        map[Symbol sort, Production def] rules
                        );
```

Figure 2.6: The *grammar* data type. The name of the constructor starts with a backslash to prevent ambiguity.

2.3 Rascal

Rascal is a domain specific language for source code analysis and manipulation, also know as meta-programming. Part of Rascal's feature set is its own grammar formalism to define grammars. The diff algorithm needs to be able to calculate the differences between two grammars described in this notation.

2.3.1 Data types

Besides the normal data type like *bool*, *int* and *str*(ing), Rascal has built-in support and syntax for several collection types, including *list*s, *set*s and *map*s. Each of these collection types have their own syntax to instantiate instances. Rascal also has a data type to represent tree structures, the *node* type. Nodes are untyped data structures which contain a name and a list of children. *Constructor*s can define typed subtypes of *node*s and can be used to define problem-specific data types. See figure 2.5 for an example.

2.3.2 Grammars in Rascal

Grammars in Rascal are represented using several different data types consisting of dozens of constructors. Each of these data types are build especially to contain parts of the abstract syntax tree of a grammar.

In these data types, the type of collection (*list*, *set* or *map*) defines the orderliness of the elements. *List*s are used to defined ordered elements and are allowed to contain multiple copies of the same element. *Sets* are used when the elements are unordered and their elements need to be unique within the set. *Map*s are used to store mappings between a

```
data Production
  = \choice(Symbol def, set[Production] alternatives)
  | \priority(Symbol def, list[Production] choices)
  | \associativity(Symbol def, Associativity \assoc,
                    set[Production] alternatives)
  | \others(Symbol def)
  | \reference(Symbol def, str cons)
  ;
```

Figure 2.7: The *Production* data type.

key and a value. The mappings are stored unordered and require the key to be unique within the map.

The *Grammar* data type

The *Grammar* data type (see figure 2.6) serves as the root of an abstract syntax tree describing a context-free grammar. It consists of a *set* of *Symbol* representing the starting rules (*starts*) and a *map* (*rules*) defining the productions. The map contains mappings of a key (*sort*) with a value (*def*). The key contains a *Symbol* containing the name of the production, the value contains a *Production* containing a tree representing the production.

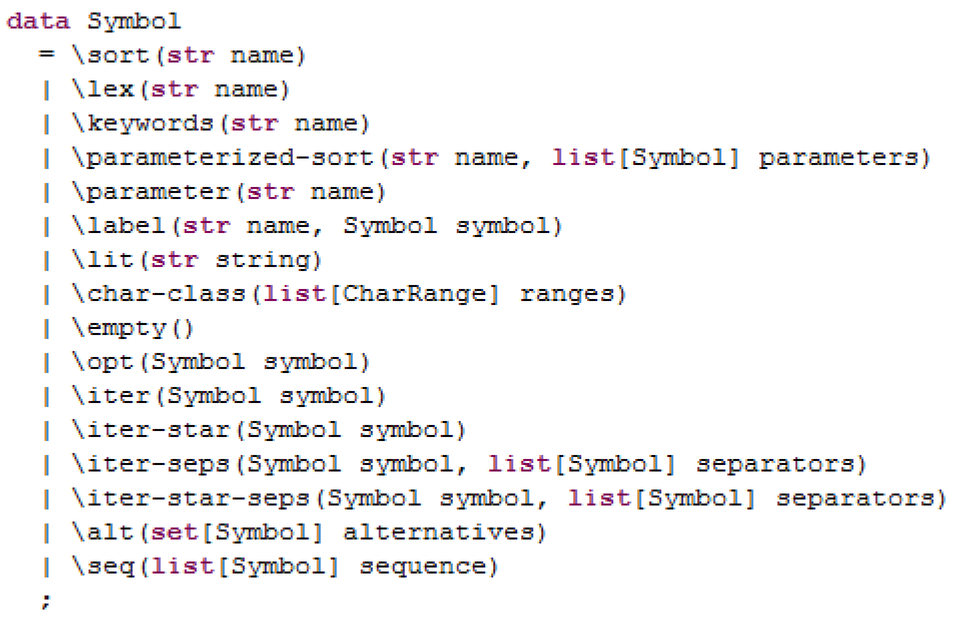
The *Production* data type

The *Production* data type is used as base type for productions (see figure 2.7). Of these five constructors, the following three are the most commonly used:

choice Defines a set of unordered alternatives.

priority Defines an ordered choices. Unlike *choice*, the order is important. It is used when multiple productions are defined and they should be tried from left to right, for instance when associativity is involved.

associativity Defines the associativity of the alternatives. The *Associativity* data type defines which associativity to use for the alternatives, *left* or *assoc* for left to right, *right* for right to left or *non-assoc* when related occurrences of alternatives are not allowed.



```
data Symbol
  = \sort(str name)
  | \lex(str name)
  | \keywords(str name)
  | \parameterized-sort(str name, list[Symbol] parameters)
  | \parameter(str name)
  | \label(str name, Symbol symbol)
  | \lit(str string)
  | \char-class(list[CharRange] ranges)
  | \empty()
  | \opt(Symbol symbol)
  | \iter(Symbol symbol)
  | \iter-star(Symbol symbol)
  | \iter-seps(Symbol symbol, list[Symbol] separators)
  | \iter-star-seps(Symbol symbol, list[Symbol] separators)
  | \alt(set[Symbol] alternatives)
  | \seq(list[Symbol] sequence)
  ;
```

Figure 2.8: The *Symbol* data type.

The *Symbol* data type

The *Symbol* data type is used to describe parts of the productions (see figure 2.8). Here is an overview of the most frequently used *Symbol* constructors:

sort A user-defined non-terminal.

lex A lexical terminal.

lit Corresponds to a terminal symbol in EBNF.

keywords A terminal used to define keywords.

parameterized-sort Defines a parameterized sort symbol. The *parameters* contain a list of *parameter* constructors, which define the names of the parameters. This way a template can be used as a symbol, like generics in Java.

parameter Contains the name of a parameters for the *parameterized-sort* constructor.

label Applies a label to a symbol.

char-class Defines an character range. *ranges* contains a list of *CharRange*, which defines the begin and end character of the range.

opt Corresponds to an option ([...]) in EBNF.

iter An one-or-more repetition. Corresponds to {...} in EBNF.

iter-star A zero-or-more repetition (an optional *iter* symbol).

iter-seps An one-or-more repetition, which need to be separated by the *separators* symbols.

iter-star-seps A zero-or-more repetition, which need to be separated by the *separators* symbols.

alt Defines an unordered set of alternatives. Corresponds to a pipe in EBNF.

seq Defines an ordered sequence of symbols. Every symbols has to match and be in the same order.

2.4 Importance of orderliness

As can be seen in section 2.3.2, all constructors which contain a collection of elements define their orderliness by their data type. Unordered collections are wrapped in a *set* (or *map*), ordered collections are stored in *list*s. Although the orderliness of elements can be ignored by converting all collections to ordered collections, it will add unnecessary information to the grammar. Since the order is not important for unordered collections, adding information to order them can lead to extra edit operations. These extra edit operations can lead to incorrect choices by ordered diff algorithms, since these extra edit operations increase the edit distance between two elements.

By treating all collections in a grammar as unordered, all order information will be destroyed. Unordered diff algorithms will ignore the order of collections in collections where the order of the elements is important. This can lead to cheaper edit scripts, but can reorder collections into an incorrect order by mutating elements into elements in another position.

Since neither scenario is desirable, both unordered and ordered diff algorithm are not usable for context-free grammars.

Chapter 3

Differencing context-free grammars

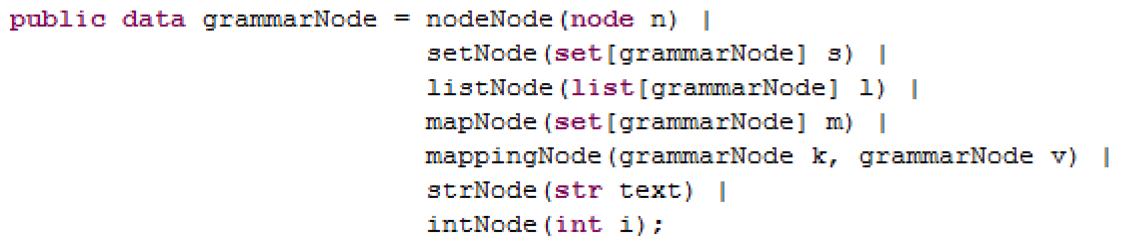
It has been established earlier that differencing grammars can benefit from algorithms that can handle both ordered and unordered elements. Since no such algorithm exists, it needs to developed. The effort of developing *cfgDiff* is discussed in this chapter.

The chapter starts off with a description of the pseudo code which is used to present parts of the algorithm. After that the preprocessing actions which are applied to the grammar are described. Following this is an overview of the supported edit operations and their cost model. Finally the different functions of the algorithm are described.

3.1 Pseudo code

Spread throughout this chapter are pseudo code of parts of the algorithm. This pseudo code has an Rascal-like syntax. To aid in readability and briefness, parts of the algorithm, such as optimizations and special cases, may be omitted. For example, *diffList* and *diffSet* return the cost of the edit script besides the edit script itself. This way the already calculated cost does not have to be recalculated when the cost of the edit script is required by any of its calling functions. Another example are various lists of *grammarNode*s. Most of these lists are lists of their id's in actual code. This way *grammarNode*s can be compared by their id's when the list gets modified, which is faster and more precise in Rascal[1]. To prevent extra code for looking up the *grammarNode* associated to an id, this detail was omitted.

[1]When comparing two objects for equality, Rascal doesn't check the annotations for equality. This introduces a side effect in our situation when two identical subtrees are in the same list. When either of the subtrees is removed from the list, the leftmost subtree of the two will be removed.



```
public data grammarNode = nodeNode(node n) |
                          setNode(set[grammarNode] s) |
                          listNode(list[grammarNode] l) |
                          mapNode(set[grammarNode] m) |
                          mappingNode(grammarNode k, grammarNode v) |
                          strNode(str text) |
                          intNode(int i);
```

Figure 3.1: The *grammarNode* data type and its constructors.

3.2 Preprocessing

The algorithm starts off by preprocessing the two grammars.

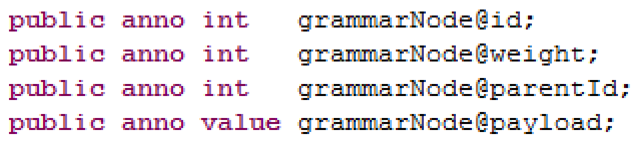
3.2.1 Transforming grammar data types to general data types

Grammars in Rascal can be build up using dozens of different constructors. Since the constructor names contain information about the action to apply on their children, the constructors can change from one into the other between two versions of a grammar while leaving their children untouched. For example, when changing an 'one or more' repetition on a subproduction to a 'zero or more' repetition, the *iter* constructor in the grammar is changed into a *iter-star* constructor. Although these transformations are simple, we need *cfgDiff* to be able to detect these kind of transformations easily without the need to define lists of possible transformations. To be able to do so, *cfgDiff* transforms all constructors into untyped *node*s. This way, all constructors can be treated the in same way.

Because constructors are treated generically, the children of constructors have to be treated generically too. This way the constructor name and children will not be seen as a entity and can be dealt with on an individual basis. By dividing the children *cfgDiff* will be able to do more advanced transformations, such as inserting elements into and deleting elements from the children. This way we can transform a *iter* constructor into an *iter-seps*[2] constructor by changing the name of the node and inserting a list of separator symbols.

To be able to divide the children, we need to transform the children into a list of type unsafe containers. Each of these types, *node*s, *list*s, *set*s, *map*s, *str*ings and *int*egers will be wrapped in their own container constructor of the *grammarNode* data type (see figure 3.1). Because of this, *cfgDiff* will work on *grammarNode*s instead of the *Grammar* data type.

[2]A 'zero or more' repetition with a list of separator symbols to serve as separation between the elements.



```
public anno int   grammarNode@id;
public anno int   grammarNode@weight;
public anno int   grammarNode@parentId;
public anno value grammarNode@payload;
```

Figure 3.2: The annotations of a *grammarNode*.

Maps

We deviate a bit from this strategy when it comes to *map*s. Maps consist of an unordered set of keys. Each key has its own value assigned to it. Because of this, we can represent a map as a *set* of *mapping*s.

Each *mapping* contains a key and value. Although these can be represented using a *node*, the key and value never switch places in a context-free grammar. Because of this, we represent a *mapping* with its own *grammarNode* type, *mappingNode*. This causes *cfgDiff* to distinguish between a node and a mapping, preventing the unnecessary comparison of between a key and a value.

3.2.2 Applying meta data

Once an element has been transformed into a *grammarNode* data type, meta data can be added to each *grammarNode*. The meta data consists of four pieces of information, stored on the *grammarNode* using annotations (see figure 3.2). These annotations and their contents are:

id A unique identifier.

weight The weight of the *grammarNode* (including the weights of its children).

parentId The identifier of the element's parent.

payload The subtree of the grammar represented by this *grammarNode* and its children.

The id contains a serial number, which should be unique in the grammar. The method of generating the id and order of assignment is irrelevant to the algorithm. The weight annotation contains the weight of the *grammarNode* (see 3.2.4). In some parts of the algorithm access to the id of the parent is needed to generate an edit operation. Since Rascal does not support back references, the id of the parent is stored instead of a reference. Because all elements of a grammar are wrapped in *grammarNode*s, the size of the tree increases. To speed up equality checks, the subtree of the grammar represented by this *grammarNode* is stored in payload. This way we can check for equality on this annotation instead of the *grammarNode* itself.

3.2.3 Hashing

X-Diff[20] hashes all XML elements in both trees. This way the hashes instead of the elements themselves can be checked for equality, which decreases run times. Although X-Diff works on XML in an unordered only model, the same principle can be applied to context-free grammars.

Although hashing is possible in Rascal, implementing the system made the algorithm run slower than without hashing. Because of this, the algorithm does not utilize hashing to improve equality checking speed.

3.2.4 The weight model

For each element in a grammar, its weight can be calculated by determining its data type and the combined weight of its children. The weight can be calculated for each type by calculating the following:

nodeNode One plus the sum of the weights of all children.

setNode One plus the sum of the weights of all elements.

listNode One plus the sum of the weights of all elements.

mapNode One plus the sum of the weights of all its *mappingNode*s.

mappingNode The weight of the key plus the weight of the value.

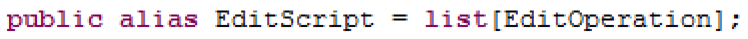
stringNode One.

integerNode One.

These weights are used by *cfgDiff* to calculate the cost of an edit operation. The weight of an element increases in value anytime the element gets larger. This is because edit operations which modify larger subtrees should get a higher cost than edit operations affecting smaller subtrees[3]. For example, removing and inserting the whole grammar tree should have a higher cost than removing and inserting a subtree of the grammar. This is caused by weights, the weight of the whole grammar is larger than the weight of any of its subtrees on their own.

Most other algorithms use a function which determine the weight of a string based on its length. This makes sense if the amount of difference between two strings make a semantical difference in the structure. However, in Rascal's grammars strings are used to identify symbols. This means that the length or amount of difference between two strings does not make a difference, the only thing which counts is whether or not the

[3]This is not true for move and substitute operations, they have a fixed costs.



```
public alias EditScript = list[EditOperation];
```

Figure 3.3: The *EditScript* type. It aliases *EditScript* as a list of *EditOperation*s.

strings match. If they do not match, they point to different symbols. Because of this, the length of a string has no influence on its weight and is fixed to a weight of one. Integers work the same way, because only the fact that they differ or not counts. Therefore the weight of any integer is also fixed to one.

For sets, lists and maps the sum of the weight of all children is taken and incremented with one to represent the set, list or map itself. Nodes however are treated differently. Although nodes are essentially a list with a string describing what action the node represents, the weight still is one plus the weight of the children[4]. In the weight calculation, the string representing the name of the node and the 'list' which contain the children do not count as weight. This means that an empty node (a node without any children) has a weight of one.

3.3 Edit scripts

When two grammars are differenced, *cfgDiff* produces an edit script. The edit script describes a possible list of mutation, edit operations, which can be performed on the original grammar to produce the modified grammar. If both grammars are the same, an empty edit script is generated.

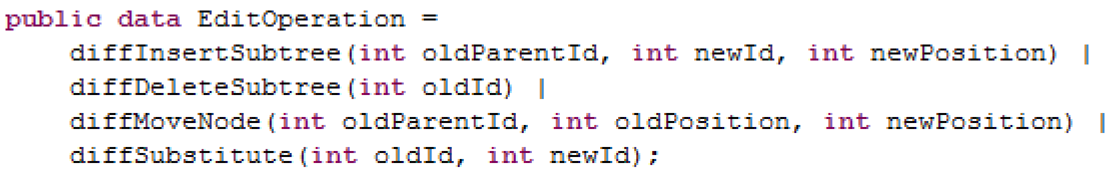
The list of edit operations are ordered. Rearranging the order can result in a different grammar or an invalid operation. For instance, if we take the following list:

A B C D

And reorder it using the following move operations:

1. Move the element at position 3 to position 1.
2. Move the element at position 2 to position 0.

Move operation 1 will rearrange the list to *A D B C* and operation 2 rearranges the list to *B A D C.* If the order of the move operations were reversed, the edit script will mutate the list into a different result. Move operation 2 would rearrange the list to *C A B D* and operation 1 rearranges the list to *C D A B.*



```
public data EditOperation =
    diffInsertSubtree(int oldParentId, int newId, int newPosition) |
    diffDeleteSubtree(int oldId) |
    diffMoveNode(int oldParentId, int oldPosition, int newPosition) |
    diffSubstitute(int oldId, int newId);
```

Figure 3.4: The *EditOperation* type and its constructors.

Element type	Insert element into	Delete element from	Substitute	Move
node	X	X	Name only	Children only
set	X	X		
list	X	X		X
map	X	X		
string			X	
integer			X	

Table 3.1: Element types and their supported edit operations.

3.3.1 Supported edit operations

cfgDiff supports four different edit operations: insert, delete, substitute and move (see figure 3.4 and table 3.1). The edit operations consist of identifiers referencing to the elements in question. This way any post processing done by future work can easily reference to the elements and their surroundings, opening up possibilities to better describe grammar changes.

Insert

The insert operation inserts an element, identified by *newId*, at a specific position in the element identified by *oldParentId*. If the parent element is ordered, the element is inserted at position *newPosition*, shifting the element currently at that position (if any) and any subsequent elements to the right. It is not possible to insert any element as a parent of an existing element.

Delete

The delete operation is the reverse of the insert operation. It deletes the element identified by *oldId* from its parent. When the element is deleted, it shifts all of the parents elements which are currently to right of that position (if any) to the left. It is not possible to delete an element while keeping a part of its subtree in its place with this operation.

[4]As opposed to one for the node itself plus one for the name string plus one for the list containing the children plus the weight of the children.

Substitute

Substitute replaces the element identifier by *oldId* with the element identifier by *newId*. If the element is a *node*, only the name of the *node* is replaced.

Move

The move operation moves an element on *oldParentId* to another position on the same element. An element can not be moved from one element onto another. The operation deleted the element at position *oldPosition* and then inserts it back at position *newPosition*. Because *set*s they are unordered, move operations are not supported on them.

3.3.2 Cost model

For each edit operation a cost can be calculated based on the weight of the element which is mutated. The cost model is deterministic, it always results in the same cost if the same edit operation, element and grammars[5] are provided.

Insert

Insert operations have a cost equal to the weight of the subtree they insert. A different approach would be to provide discounts on cost for larger subtrees. However, this would favor mutate unchanged elements around the subtree to benefit from the discount on a larger inserted subtree. Because this behavior is not wanted, linear cost model for inserts is implemented.

Delete

Delete operations have the same cost model as insert operations. This is because delete operations are the reverse of an insert operation and thus should have the same cost model.

Substitute

Substitutes always have a cost of one. Since substitutes can only be applied to strings, integers and the labels of nodes (which are a string), the cost is equivalent to the weight of those types. Because the amount of difference does not matter, only the fact that

[5]And if the identifiers are assigned in the same manner.

the value has changed, a fixed cost model was chosen. The cost of one was chosen because this is the weight of any of the possible elements that can be substituted. Also, a substitute operation should be cheaper than deleting and inserting the node, which has a combined cost of 2 for both strings and integers.

Move

Moves too have a fixed cost of one. Although moves can be applied to any type of element, they only change the ordering of the children of an element. Because of this a fixed cost model was chosen. Also, a move operation should be cheaper than deleting and inserting the element, which has a cost of at least 2. Furthermore a move operation should also be preferred over swapping the values of both elements with two substitutes, if that is possible, which has a cost of 2.

I will illustrate the choices in the cost model with a complex scenario. Consider the following ordered *list*s of strings:

["A", "B"] and ["B", "C"]

When the first list gets mutated into the second list, there are three possible scenario's[6]. The first scenario is moving "A" after "B" and substituting "A" for "C". The second scenario deleted "A" and inserts "C" after "B". The third scenario substitutes "B" with "C" and "A" with "B". Because all three scenario's are as likely to happen in real life scenario's, all edit scripts should result in an equal cost. Under the current cost model, this is true.

3.4 The *cfgDiff* algorithm

After the supported edit operations and their costs have been defined, we can dive deeper into *cfgDiff*.

3.4.1 Early rejection

When diffing two grammars, *cfgDiff* will calculate the edit distance between a lot of elements. A lot of those calculations will be unnecessary, because changing the original element into the new element will not make any sense in a context-free grammar. For example, changing an element containing an integer into an element containing a set is something that should not happen. If the parent element would allow a set of alternatives at that position in the grammar, the integer would be contained into a set already. Because the elements are not compatible with each other, matching the subtrees of those

[6]For simplicity, duplicate and more complex edit scripts are omitted as they are already represented or irrelevant for this example.

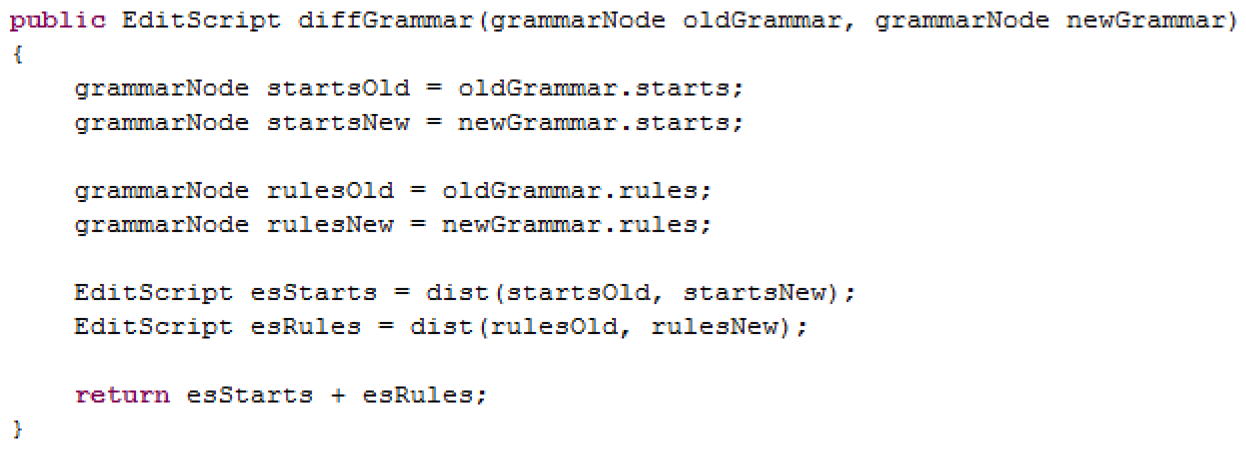
elements, if they have any, against each other would make no sense either since the result of those matches are going to be rejected anyway. To avoid having to calculate those unnecessary calculations, an early rejection system has been build into the algorithm.

The early rejection system is a simple system. Because different types of elements probably will not match, a comparison between two different kinds of elements will always be rejected. This way we can compare the constructor names of two *grammarNode*s and reject them if they does not match. Although the comparison is valid in some locations of the grammar, they just can not be transformed into each other, an edit script has to be returned anyway. Because of this the worst possible edit script will be returned, deleting the original element and inserting the new element.

3.4.2 Bottom up or top down

Most algorithms found during my literary study traverse the tree bottom up [26, 4, 20]. When a bottom up approach is used, the algorithm starts at the leafs. When traversing up the tree the leafs combine with their parents and their sibblings, growing into bigger subtrees. This way the edit distance between the elements of the subtree can be calculated incrementally by using the already calculated edit distance between their children.

Although the bottom up approach seems to be favored by existing literature, if also tested *cfgDiff* with a top down approach. Implementing this approach gave a significant speed improvement over the bottom up approach. This can result from two factors. First of all, when calculating the edit distance between two nodes using the top down approach, the children of both nodes can be compared with each other before calculating the edit distance between each pair of nodes. Because comparing nodes is extremely cheap compared to calculating the edit distance between two nodes, the matching space for the edit distance function can be reduced significantly with a minimal investment. Secondly, the bottom up approach was used by an early paper [26] and two papers presenting a XML diff algorithm [4, 20]. Since most XML documents have a more fixed hierarchy than grammars, the depth of the leave nodes in XML documents will not vary much between relevant leaf nodes. This means matching or related subtrees can be found more effectively in XML documents using a bottom up approach. Although grammars also have a fixed hierarchy describing which nodes can be the children of which node, the structure of a grammar describes a large part of the grammar (for instance a node describing whether or not the content should be repeated once or more). In XML documents the XML structure describes how the information is structured in the document, for example that this node contains the name of an article. This means early rejection based on the signature of a node, like [20] uses, is counter productive for *cfgDiff* because it incorrectly rejects valid matches, resulting in a larger than necessary edit script. However, this causes that fewer comparisons can be rejected early, increasing the number of edit distances that need to be calculated and slowing down the algorithm.



```
public EditScript diffGrammar(grammarNode oldGrammar, grammarNode newGrammar)
{
    grammarNode startsOld = oldGrammar.starts;
    grammarNode startsNew = newGrammar.starts;

    grammarNode rulesOld = oldGrammar.rules;
    grammarNode rulesNew = newGrammar.rules;

    EditScript esStarts = dist(startsOld, startsNew);
    EditScript esRules = dist(rulesOld, rulesNew);

    return esStarts + esRules;
}
```

Figure 3.5: Pseudo code of the *diffGrammar* function.

3.4.3 *diffGrammar*: diffing two grammars

The *diffGrammar* function is used to diff two grammars (see figure 3.5). Since the root of a grammar is a constructor, the preprocessing will transform it into a nodeNode. Although *dist* can effectively do this itself, it will compare both children against each other if both the starting rules and productions are changed. To eliminate this, the *diffGrammar* function splits up the two children and pulls them separately through *dist*. Since the algorithm effectively removes matching elements from sets and maps, no further actions are necessary to reduce the number of starting rules and productions.

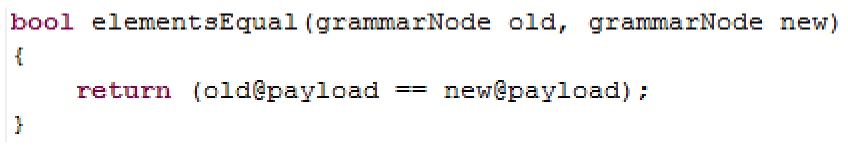
3.4.4 *dist*: calculate the difference between two elements

The *dist* function is used to calculate the edit distance between two elements (see figure 3.6). First, the function checks if both elements are equal. If so, the function terminates with an empty edit script. If not, the function checks whether both element types are compatible. If the element types are not compatible, the function returns an edit script deleting the old element and inserting the new element. This serves as early rejection (see 3.4.1).

For each *grammarNode* type, *dist* uses specific calculations to calculate the edit script between two elements. For *int*egers and *str*ings, a substitute is generated. *node*s generate a substitute operation if the name does not match plus the edit script for mutating its children, which are treated as a *list*. For *list*s and *set*s, *diffList* and *diffSet* will be called to calculate the edit script. In a *map*, the *mapping*s will be differenced as if they are part of a *set*. Their *mapping*s are divided into the key and value are pulled through *dist* separately. Their edit scripts will be combined to form the edit script of the *mapping*.

```
EditScript dist(grammarNode old, grammarNode new)
{
    if (elementsEqual(old, new))
        return []; // Empty edit script

    if (getName(old) != getName(new)) // Check if the types match
        return [diffDeleteSubtree(old@id),
                diffInsertSubtree(old@parentId, new@id, 0)];

    switch (getName(old))
    {
        case "nodeNode":
        {
            EditScript diff = [];

            if (getName(old.n) != getName(new.n))
                diff += diffSubstitute(old@id, new@id);

            return diff + diffList(old.n, new.n);
        }
        case "listNode":
            return diffList(old.l, new.l);
        case "setNode":
            return diffSet(old.s, new.s);
        case "strNode":
            // They don't match since elementsEqual failed
            return [diffSubstitute(old@id, new@id)];
        case "intNode":
            // They don't match since elementsEqual failed
            return [diffSubstitute(old@id, new@id)];
        case "mapNode":
            return diffSet(old.m, new.m);
        case "mappingNode":
            return dist(old.k, new.k) + dist(old.v, new.v);
    }
}
```

Figure 3.6: Pseudo code of the *dist* function.



```
bool elementsEqual(grammarNode old, grammarNode new)
{
    return (old@payload == new@payload);
}
```

Figure 3.7: Pseudo code of the *elementsEqual* function.

3.4.5 *elementsEqual*: comparing two elements for equality

The *elementsEqual* function checks if two elements are equal to each other (see figure 3.7). Because the original grammar is packed into *grammarNode*s, we can compare the value of the *payload* annotation for equality. This way we can check if the subtrees represented by both elements match.

3.4.6 *findBestChanges*: finding the best combination of changes

The *findBestChanges* function tries all combinations of inserting, deleting and changing the elements of one list into another list (see figure 3.8). Although the function works on two lists (to allow copies of elements to exist in the lists as opposed to sets), the elements are treated as unordered to reduce the complexity of the function. This way it can handle both ordered and unordered lists. If the list was ordered, the calling function can order the results afterwards To further reduce the complexity even further, the change of one element into another also counts as the change of deleting one element and inserting the other element.

Because the *findBestChanges* function needs to try all possible combinations, it has a very high complexity. To make matters worse, the edit distance between every pair of elements from both lists need to calculated[7]. Replacing this function with a function where a strategy picks which element to change into another element first has been tried, but were unable to return the optimal result every time.

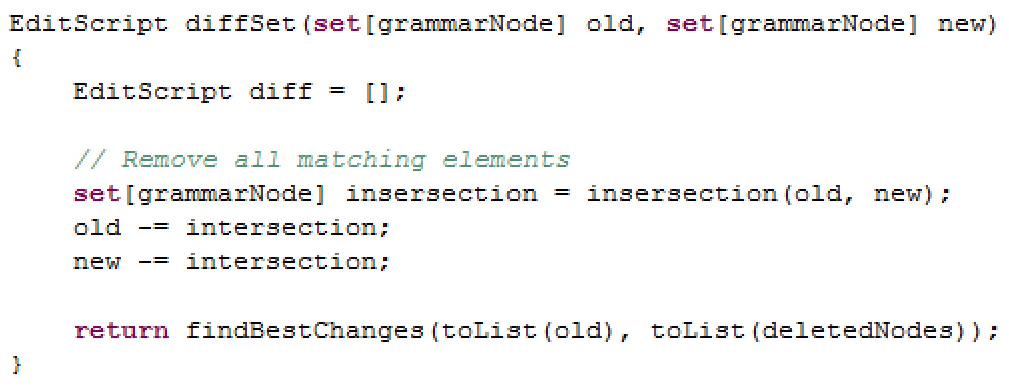
The function first checks if either of the two incoming lists are empty. If this is the case the function returns a diff script deleting or inserting all elements from the nonempty list. Since the function is used recursively, this serves as a termination. The first element of the list containing the fewest elements is taken and compared against every element of the other list. In each iteration the functions calls itself with both lists minus the elements taken from both lists in this iteration, resulting in an edit script. Then the function retrieves the edit distance between the pair of elements. Both edit scripts are added together and stored. When all elements from the longest list are iterated through, the cheapest edit scripts are chosen and returned.

The function returns a list of edit scripts, all of which are the cheapest edit script.

[7]Although omitted in the pseudo code, a cache is used to prevent the calculation of the distance between a pair of elements multiple times

```
list[tuple[EditScript, map[int, int]]]
    findBestChanges(list[grammarNode] old, list[grammarNode] new)
{
    if (size(new) > 0)
        return [<[diffInsertSubtree(new[i]@parentId,
                new[i]@id, idx) | idx <- index(new)], ()>];
    if (size(old) > 0)
        return [<[diffDeleteSubtree(d@id) | d <- old], ()>];

    list[tuple[EditScript, map[int, int]]] options = [];

    if (size(new) < size(old))
    {
        grammarNode i = new[0];
        list[grammarNode] restI = [x | x <- new, x@id != i@id];
        for (d <- old)
        {
            tuple[int, list[tuple[EditScript, map[int, int]]]]
                rest = <0, []>;

            if (size(old) > 1 || size(new) > 1)
                rest = findBestChanges([x | x <- old, x@id != d@id], restI);
            else
                rest = <0, [<[], ()>]>;

            for (r <- rest)
            {
                EditScript editDiff = dist(d, i);
                EditScript worstDiff = [diffDeleteSubtree(d@id),
                        diffInsertSubtree(i@parentId, i@id,
                            findPosition(new, i))];

                if (getCost(worstDiff) <= getCost(editDiff))
                    options += <r[0] + worstDiff, r[1]>;
                if (getCost(worstDiff) >= getCost(editDiff))
                    options += <r[0] + editDiff, r[1] + (d@id : i@id)>;
            }
        }
    }
    else
    {
        ; // Same as above, but take one old and loop through new...
    }

    map[int, list[tuple[EditScript, map[int, int]]]] costOptions = ();
    for (option <- options)
    {
        costOptions[getCost(option)] += option;
    }
    return costOptions[min(domain(costOptions))];
}
```

Figure 3.8: The *findBestChanges* function.



```
EditScript diffSet(set[grammarNode] old, set[grammarNode] new)
{
    EditScript diff = [];

    // Remove all matching elements
    set[grammarNode] insersection = insersection(old, new);
    old -= intersection;
    new -= intersection;

    return findBestChanges(toList(old), toList(deletedNodes));
}
```

Figure 3.9: Pseudo code of the *diffSet* function.

Because the function does not know where moves will be required and how the picking of elements to pair up affects the number of required moves, it can not pick one. To make sure that the calling function can reorder the element pairs using the least amount of moves possible, all possibilities are returned. This way the calling function calculate move edit operation for all these possibilities and pick the cheapest one. For example, the *findBestChanges* function retrieves the following input lists:

["A", "B"] and ["C", "D"]

The cheapest edit script between the two lists involves two substitutes. However, because *findBestChanges* works unordered to reduce the complexity of the function, it has two valid edit scripts which can modify the first list into the second. The first possibility substitutes "A" for "C" and "B" for "D", which keeps the list in the same order as a side effect. The second possibility substitutes "A" for "D" and "B" for "C", which reorders the list and will require one move operation later on. Because *findBestChanges* works unordered, it has no preference for either of the two edit scripts since they both are the cheapest of all the enumerated possibilities. However, if the second option was an ordered list, the calling function needs to add an extra move operation to rearrange the list. To prevent this *findBestChanges* will return a list of possible edit scripts and lets the calling function pick the best option from those.

3.4.7 *diffSet*: calculate the difference between two sets

The *diffSet* function diffs two sets (see figure 3.9). It results in a edit script for mutating the original set into the new set.

Reducing complexity

When the edit distance between two sets need to be calculated, the algorithm tries to reduce the matching space between the two sets first. Since checking for equality is very cheap compared to calculating the edit distance between two elements, the algorithm tries to reduce the matching space by removing matchings elements.

When two elements are equal, the edit distance between the two is zero. No cheaper combination with either of the elements in it can be made with any other elements from either set, so letting *findBestChanges* process those combinations is a waste of time. Therefore elements that exist in both sets get removed.

To find the matching elements, the algorithm calculates the intersection between the two sets. Every element contained in the intersection gets removed from the old and new sets.

Finding the best combination of changes

Once all matching elements have been eliminated from both sets, the *findBestChanges* will do the rest of the work by finding the best way to change the original elements into the new elements.

3.4.8 *diffList*: calculate the difference between two lists

The *diffList* function differences two lists (see figure 3.10[8]). It results in an edit script for mutating the original list into the new list. Unlike *diffSet*, *diffList* respects the orderliness of the lists by adding move edit operations to the edit script if necessary.

Reducing complexity

Like the *diffSet* function, the *diffList* function first tries to reduce the complexity by finding matching pairs (see figure 3.11). At first, it tries to remove elements from the front and back of the list by checking for their equivalence. If they are equivalent, the elements can be removed from the list. Since these elements are in the front and/or back of the list, removing them from both lists will not make any difference on the reordering process.

Because *diffList* works on *list*s instead of *set*s, an element can occur as multiple copies in a *list*, for example when the name of a non-terminal appears multiple times within a

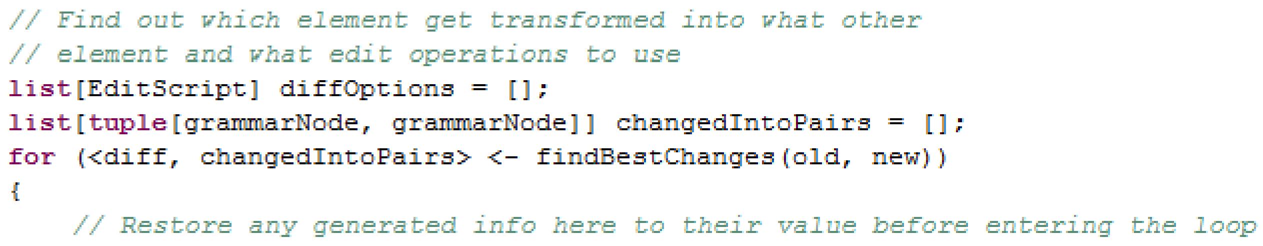
[8]Parts of this figure are reused in the subsequent subsections to point at certain parts of the pseudo code.

```
EditScript diffList(list[grammarNode] old, list[grammarNode] new)
{
    list[grammarNode] oldCopy = old;
    list[grammarNode] newCopy = new;

    // Check if either list is empty, if so we can return early
    if (size(old) == 0)
        return [diffInsertSubtree(oldParentId, new[idx]@id, idx)
             | idx <- index(new)];
    if (size(new) == 0)
        return [diffDeleteSubtree(x@id) | x <- old];

    // Find all matching pairs
    set[tuple[grammarNode, grammarNode]] matchingPairs =
        {<o, n> | o <- old, n <- new, elementsEqual(o, n)};

    // Check which matching pairs we should use to form
    // the longest common subsequence of them
    <lcsMatchingPairs, lcsNonMatchingPairs> =
        lcsMatchingPairs(old, new, matchingPairs);

    // Remove those matching pairs from the lists
    old -= [o | <o, n> <- lcsMatchingPairs];
    new -= [n | <o, n> <- lcsMatchingPairs];

    // Find out which element get transformed into what other
    // element and what edit operations to use
    list[EditScript] diffOptions = [];
    list[tuple[grammarNode, grammarNode]] changedIntoPairs = [];
    for (<diff, changedIntoPairs> <- findBestChanges(old, new))
    {
        // Restore any generated info here to their value before entering the loop

        // Find out the pairs to use from all pairs to form the
        // longest common subsequence of them
        <lcsPairs, movePairs> =
            lcsMatchingPairs(old, new, matchingPairs + changedIntoPairs);

        // Make lists which contain items that are either inserted or deleted
        old -= [o | <o, n> <- (lcsPairs + movePairs)];
        new -= [n | <o, n> <- (lcsPairs + movePairs)];

        // Delete all non matching old items
        diff += [diffDeleteSubtree(o@id) | o <- old];

        // Make a list of all old id's without the deleted ones
        list[int] oldIds = [o@id | o <- (oldCopy - old)];

        // Transform the oldIds list to contain their paired new id's
        map[int, int] lookup =
            (o@id : n@id | <o, n> <- (lcsPairs + movePairs));
        list[int] newIds = [lookup[id] | id <- oldIds];
        // Make a list of id's from the new list in that order
        list[int] toOrder = [n@id | n <- newCopy];

        // Make a list of new id's that aren't moved
        set[int] unmovedIds = {n@id | <o, n> <- lcsPairs};

        // Move all movePairs to their new positions
        for (<x, newId> <- movePairs)
        {
            // Find the from location and remove the element
            int fromPosition = findPosition(newIds, newId);
            newIds = remove(newIds, fromPosition);

            int toPosition = 0;
            // Take the toOrder list and take the reversed order
            // of all elements to the left of newId
            for (curId <-
                reverse(slice(toOrder, 0, findPosition(toOrder, newId))))
            {
                // Check if that element has been unmoved
                if (curId in unmovedIds)
                {
                    toPosition = findPosition(newIds, curId)+1;
                    unmovedIds += curId;
                    break;
                }
            }

            // Insert the element at the new location
            newIds = insertAt(newIds, toPosition, newId);

            // Add the move edit operation
            diff += diffMoveNode(old@parentId, fromPos, toPos);
        }

        // Insert all non matching new items
        diff += [diffInsertSubtree(n@id, findPosition(newCopy, n@id))
             | n <- newCopy, n in new];

        diffOptions += diff;
    }

    map[int, EditScript] costOptions = ();
    for (diff <- diffOptions)
    {
        costOptions[getCost(diff)] += diff;
    }
    return costOptions[min(domain(costOptions))][0];
}
```

Figure 3.10: The pseudo code of the *diffList* function.

```
EditScript diffList(list[grammarNode] old, list[grammarNode] new)
{
    list[grammarNode] oldCopy = old;
    list[grammarNode] newCopy = new;

    // Check if either list is empty, if so we can return early
    if (size(old) == 0)
        return [diffInsertSubtree(oldParentId, new[idx]@id, idx)
            | idx <- index(new)];
    if (size(new) == 0)
        return [diffDeleteSubtree(x@id) | x <- old];

    // Find all matching pairs
    set[tuple[grammarNode, grammarNode]] matchingPairs =
        {<o, n> | o <- old, n <- new, elementsEqual(o, n)};

    // Check which matching pairs we should use to form
    // the longest common subsequence of them
    <lcsMatchingPairs, lcsNonMatchingPairs> =
        lcsMatchingPairs(old, new, matchingPairs);

    // Remove those matching pairs from the lists
    old -= [o | <o, n> <- lcsMatchingPairs];
    new -= [n | <o, n> <- lcsMatchingPairs];
```

Figure 3.11: Reducing the complexity in the *diffList* function.



```
// Find out which element get transformed into what other
// element and what edit operations to use
list[EditScript] diffOptions = [];
list[tuple[grammarNode, grammarNode]] changedIntoPairs = [];
for (<diff, changedIntoPairs> <- findBestChanges(old, new))
{
    // Restore any generated info here to their value before entering the loop
```

Figure 3.12: Finding the best combination of changes in the *diffList* function.

production. Because of this, neither of the elements from matching pairs are removed from the list at this time. Instead they are marked as a matching pair of elements.

Because an element can occur as multiple copies within a list and may not appear as frequent in the other list, we need to find the best copy for those cases to minimize the number of moves. An algorithm solves this problem by selecting which combination of matching pairs form the longest sequence of matching pairs while maintaining the order of the lists. All matching pairs which are not in this sequence are unmarked, all elements in the sequence are removed from their respective list. This reduces the list length and produces pairs of elements which are equivalent and may require a move later on (because edit operations will make non matching element equivalent later on, moves will be calculated later).

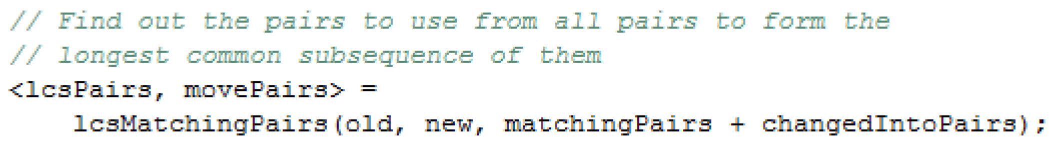
The *lcsMatchingPairs* function is an altered version of the longest common subsequence[18] and finds the longest sequence of matching pairs. Because *cfgDiff* already checked for equality, the algorithm can check the set of matching pairs to see if the elements are equal. The function returns two sets of matching pairs. The first set contains the longest sequence of matching pairs that preserve the order of the original lists. The second set contains all matching pairs which does not contain any elements of the first set, making it the set of all matching pairs not in the longest sequence.

Finding the best combination of changes

Once the matching space has been reduced, the *findBestChanges* function will try all combinations of changing a list into the other list to find the cheapest combinations of changes (see figure 3.12). Besides each edit script, the *findBestChanges* function also provides a list of pairs of elements which are mutated from one into the other[9].

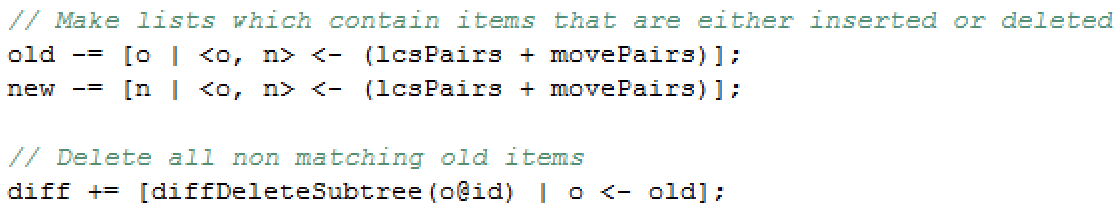
Since *findBestChanges* returns a list of the cheapest edit scripts (see 3.4.6), the *diffList* functions enters a loop here. For every possibility of *findBestChanges*, it will generate the full edit script including moves. This way *cfgDiff* can pick the cheapest edit script at the end of the function.

[9]Pairs where one elements gets deleted and the other inserted and unmatched elements are removed from this list since they does not require a move



```
// Find out the pairs to use from all pairs to form the
// longest common subsequence of them
<lcsPairs, movePairs> =
    lcsMatchingPairs(old, new, matchingPairs + changedIntoPairs);
```

Figure 3.13: Finding which pairs require a move in the *diffList* function.



```
// Make lists which contain items that are either inserted or deleted
old -= [o | <o, n> <- (lcsPairs + movePairs)];
new -= [n | <o, n> <- (lcsPairs + movePairs)];

// Delete all non matching old items
diff += [diffDeleteSubtree(o@id) | o <- old];
```

Figure 3.14: Deleting elements in the *diffList* function.

Finding which pairs require a move

The *lcsMatchingPairs* function can then be used to find out which of the matching pairs require a move (see figure 3.13). This way we know which pairs form the smallest set of pairs that require a move to end up in the correct order.

Delete unmatched elements

Once we know which elements form pairs, we can figure out which elements are not in any pair (see figure 3.14). These elements will either be inserted of deleted, depending on which list they originate from. To make the rearrangement of elements that will happen later easier, *cfgDiff* also deletes all elements that need to be deleted here. All elements that need to be inserted, will be inserted at a later stage.

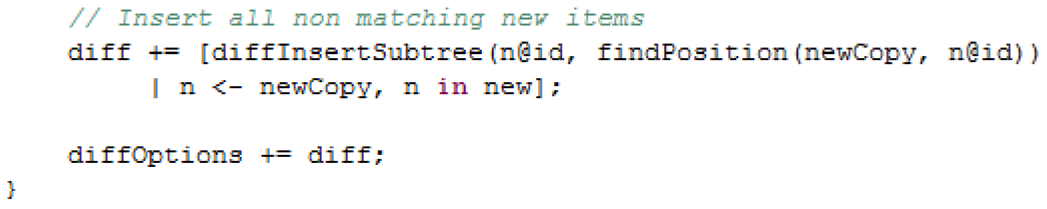
Finding out where to move elements to

The algorithm knows which pairs require a move, but need to figure out where to move the element to (see figure 3.15). Because elements will be moved around in the list, simply taking their position will not result in the correct move operations everytime. However, a sequence of steps can be executed to figure out which position the element has at this point and to which position the element has to be moved to:

- Take the list of old elements and remove all deleted elements.
- Transform the list to their matched new elements (*newIds*).
- Make a list in the order we want to reorder the element to (*toOrder*).

```
// Make a list of all old id's without the deleted ones
list[int] oldIds = [o@id | o <- (oldCopy - old)];

// Transform the oldIds list to contain their paired new id's
map[int, int] lookup =
    (o@id : n@id | <o, n> <- (lcsPairs + movePairs));
list[int] newIds = [lookup[id] | id <- oldIds];

// Make a list of id's from the new list in that order
list[int] toOrder = [n@id | n <- newCopy];

// Make a list of new id's that aren't moved
set[int] unmovedIds = {n@id | <o, n> <- lcsPairs};

// Move all movePairs to their new positions
for (<x, newId> <- movePairs)
{
    // Find the from location and remove the element
    int fromPosition = findPosition(newIds, newId);
    newIds = remove(newIds, fromPosition);

    int toPosition = 0;
    // Take the toOrder list and take the reversed order
    // of all elements to the left of newId
    for (curId <-
        reverse(slice(toOrder, 0, findPosition(toOrder, newId))))
    {
        // Check if that element has been unmoved
        if (curId in unmovedIds)
        {
            toPosition = findPosition(newIds, curId)+1;
            unmovedIds += curId;
            break;
        }
    }

    // Insert the element at the new location
    newIds = insertAt(newIds, toPosition, newId);

    // Add the move edit operation
    diff += diffMoveNode(old@parentId, fromPos, toPos);
}
```

Figure 3.15: Moving elements in the *diffList* function.



```
    // Insert all non matching new items
    diff += [diffInsertSubtree(n@id, findPosition(newCopy, n@id))
         | n <- newCopy, n in new];

    diffOptions += diff;
}
```

Figure 3.16: Inserting elements in the *diffList* function.

- Make a list of new elements that does not require a move (*unmovedIds*).
- For each pair which needs a move, do the following:
  - Take the current position of the element in *newIds*. This is the *fromPosition*.
  - Set the *toPosition* to zero, if it is not found we want to move the element to the beginning of the list.
  - Loop through all if the id's in *toOrder* that are left of curId. Do this from right to left.
  - If that id is found in *unmovedIds*, set it as *toPosition*, add it to *unmovedIds* and break from this loop.
  - Add the move operation to the diff script.
  - Update the *newIds* list to reflect the move.

Insert unmatched items

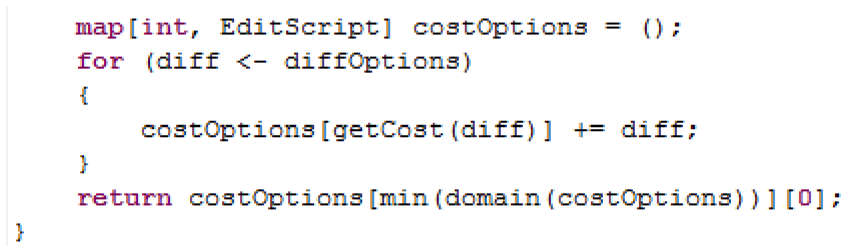
Once all elements have been deleted and moved, it leaves us with all elements that need to be inserted (see figure 3.16). We do this from left to right to simplify the process of finding the position to insert at. To do this we loop through the original new list. For each element we check if that element is in the insert list. If so, we add an insert operation to the diff script.

At this point the edit script is complete for this incomplete edit script that *findBestChanges* provided. We add this edit script to a list and try the next incomplete edit script.

Picking the cheapest edit script

At this point, all possibilities of *findBestChanges* have been completed and collected in a list (see figure 3.17). We find the cheapest edit script[10] and return it.

[10]If multiple edit scripts share the same cost, we can pick any. Which one does not matter at this point since the changing of a list into another list does not affect its surrounding elements.



```
    map[int, EditScript] costOptions = ();
    for (diff <- diffOptions)
    {
        costOptions[getCost(diff)] += diff;
    }
    return costOptions[min(domain(costOptions))][0];
}
```

Figure 3.17: Picking the cheapest edit script in the *diffList* function.

Chapter 4

Results

After the algorithm has been presented, we can test the algorithm. The algorithm has been tested for correctness of the edit script, run time and shortness of the edit script. After these tests, the *cfgDiff* algorithm will be applied to a real life grammar and its output[1] will be compared against other diff algorithms.

4.1 Test set generation

The algorithm is tested by generating random trees and random changes. Because *cfgDiff* works on a type unsafe model, trees are generated instead of grammars. This way the size of the tree is more manageable because we are not bound by predetermined data types, which have a mandatory minimum number of children. This results in less variance between the weights of the generated trees. Furthermore, this simplifies the generation of trees and will never result in invalid data types when the tree is modified.

The weight of the generated tree is determined by an input variable, *treeSize*. An algorithm generates a random root node and up to five children[2]. For each of these children, a random element is chosen and allotted a part of the *treeSize* budget which will be the resulting weight of the child.

Each of these children will be a node, list or set. The generation between these types has been weighted to generated a node 4 out of 6 times, a list 1 out of 6 times and a set 1 out of 6 times. This way, the tree will be a better reflection of the contents of a grammar tree, which are node heavy. Each of these types will be filled with their own children, generated recursively, if the remaining weight budget for the child allows it.

[1]Because of recent changes in Rascal's grammar data types, the data types in the output of *cfgDiff* may not match those described in section 2.3.

[2]This limit is put into place to prevent the generation of a large number of elements, which does not occur often in real life grammar scenario's, which will slow down *cfgDiff* down dramatically.

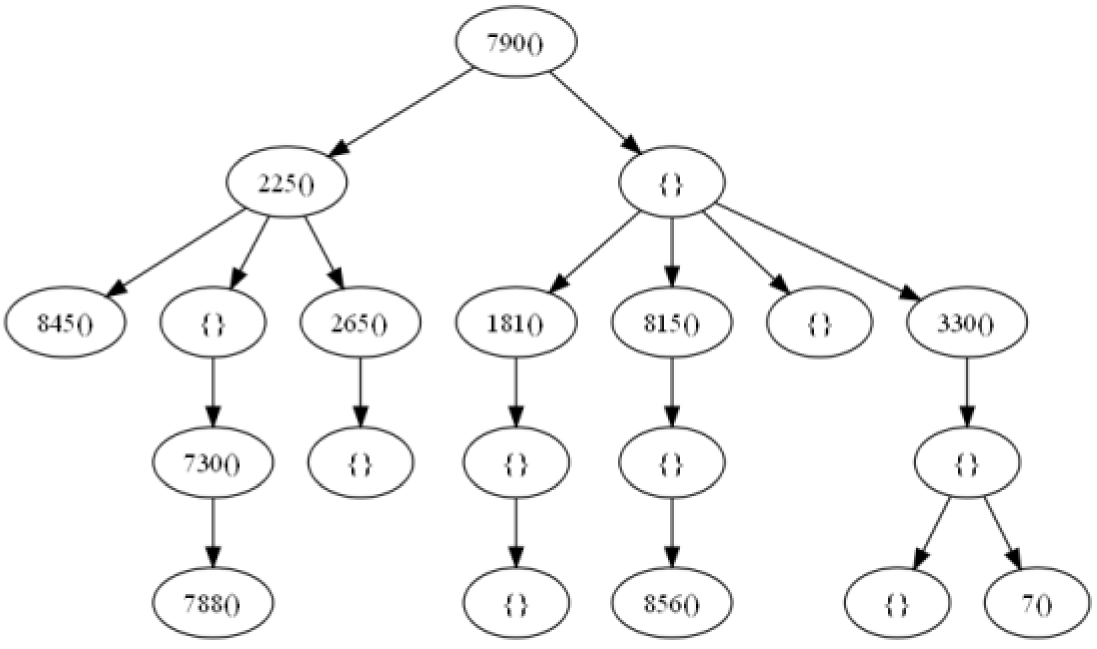


Figure 4.1: The original generated tree. Nodes containing {} are a set, their children are the elements of the set. A number followed by () denotes a node. The number is the name of the node and are a strings. Numbers where used in test set generation because they are easier to distinguish than random texts.

Once the original tree has been generated, an altered tree has to be generated. To do this, an algorithm generates up to *treeSize* random changes and applies it to a copy of the original tree. Although the changes are not recorded[3], the edit cost of these changes are computed and saved.

Figure 4.1 provides an example of an original tree. Figure 4.2 shows what the tree looks like after random modifications have been applied to that tree. Figure 4.3 shows the output of *cfgDiff* when ran on the two grammars.

4.2 Test run with small tree sizes

The first test run focused on small tree sizes. Because the trees remain small, large numbers of samples can be completed relatively quickly and would result in small data sets if an error occurred, making them easier to diagnose. This test run contained 10.000 samples, were all of the generated original trees has a weight of eleven. To generate an altered tree, between zero and ten random edit operations (insert, delete, substitute and

[3] *cfgDiff* may generate another or more efficient edit script, making the comparison of the contents of an edit script almost impossible to do.

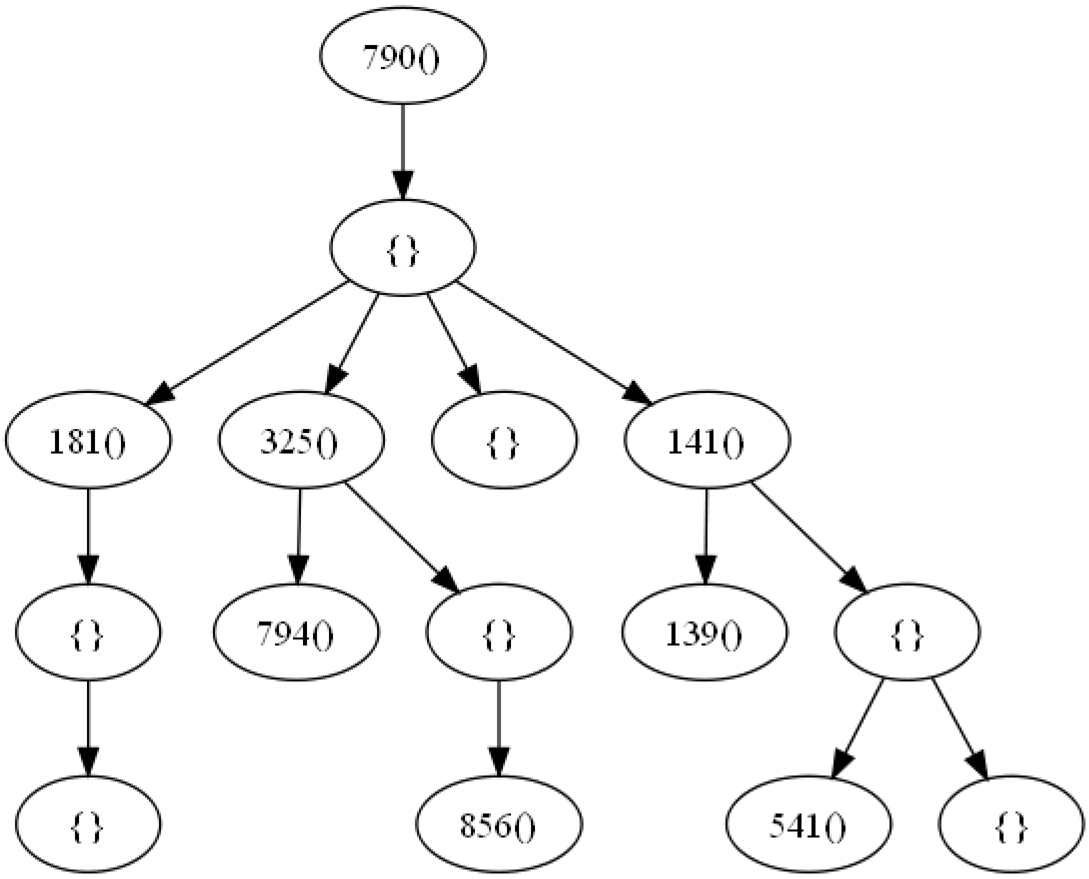
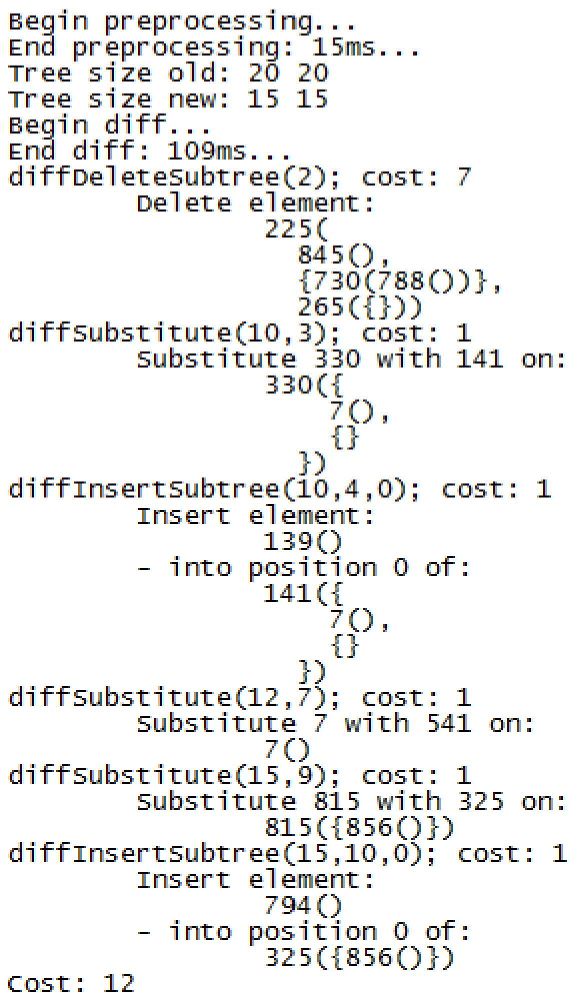


Figure 4.2: The modified tree. See figure 4.1 for the original tree and a legend.



```
Begin preprocessing...
End preprocessing: 15ms...
Tree size old: 20 20
Tree size new: 15 15
Begin diff...
End diff: 109ms...
diffDeleteSubtree(2); cost: 7
        Delete element:
                225(
                  845(),
                  {730(788())},
                  265({}))
diffSubstitute(10,3); cost: 1
        Substitute 330 with 141 on:
                330({
                    7(),
                    {}
                  })
diffInsertSubtree(10,4,0); cost: 1
        Insert element:
                139()
        - into position 0 of:
                141({
                    7(),
                    {}
                  })
diffSubstitute(12,7); cost: 1
        Substitute 7 with 541 on:
                7()
diffSubstitute(15,9); cost: 1
        Substitute 815 with 325 on:
                815({856()})
diffInsertSubtree(15,10,0); cost: 1
        Insert element:
                794()
        - into position 0 of:
                325({856()})
Cost: 12
```

Figure 4.3: The difference between the trees of figure 4.1 and figure 4.2.

move) were applied to the original tree.

4.2.1 Correctness of the edit script

When applied to the original tree, all edit script resulted in the altered tree. This shows that for this test set, all edit scripts are correct.

4.2.2 Run time

The run time of each run of *cfgDiff* was measured using *cpuTime* (exclusive preprocessing time). The 10.000 samples took 328.4 seconds to difference, resulting in an average run time of 32.8 milliseconds[4]. The longest run time in this test run was 577.2 milliseconds. The longest run times were the result of a lot of insert operations, which caused a larger than usual altered tree. Preprocessing took 33.2 seconds, an average of 3.3 milliseconds per sample.

4.2.3 Shortness of the edit script

In three cases the edit cost was larger than the cost of the generated changes (see section 4.5 for an explanation), all of which were 1 point more costlier than the generated changes.

4.3 Test run with larger tree sizes

Once we know that small tree sizes result in correct edit script, we can increase the size of the trees. This way we can see how the algorithm performs when larger trees have to be differenced. This is important because Grammars in Rascal tend to be represented by very large trees.

This test run started with a *treeSize* of 10[5] With this *treeSize*, 10 random tests will be generated and differenced. When the runs are completed, the *treeSize* is increased with 10 and another 10 tests will be ran. This goes on until 1.000 tests have been completed or an error has been found.

[4]The resolution of *cpuTime* was limited to 15,6 milliseconds on the test machine, which may affect the preciseness of the results

[5]Because the tree generation creates a root node with 10 weight points worth of children, the weight of the total tree is *treeSize* + 1.

	1-2	2-3	3-4	4-5	1-5
insert	12	31	6	0	46
delete	31	0	308	0	336
substitute	1	2	10	0	16
move	0	0	0	0	0
total	44	33	324	0	398

Table 4.1: The edit cost for each edit operation between the versions of Derric's grammar.

4.3.1 Correctness of the edit script

As with the small tree sizes test run, all edit scripts resulted in the altered tree, showing that for this test set, all edit scripts are correct.

4.3.2 Run time

The 1.000 samples took 12.796 seconds to difference, resulting in an average run time of 12.8 seconds. The longest run time in this test run was 185.8 seconds. Preprocessing took 109.9 seconds, an average of 109.9 milliseconds per sample.

4.3.3 Shortness of the edit script

As opposed to the small tree size test, every cost of the calculated edit script were smaller than or equal to the cost of the generated changes.

4.4 Application of *cfgDiff* to a real life grammar

Until now, all presented result are based on generated data. To see how *cfgDiff* performs on context-free grammars, real grammars need to be used as input for *cfgDiff*. Since not many grammars written in Rascal have undergone an evolution, pickings for test subjects are scarce.

I've picked the grammar of Derric[16] as test subject for *cfgDiff*. Its grammar evolution includes five versions in Rascal syntax, before that it was written in SDF. See table 4.1 and figure 4.6 for the results. The edit costs of each diff may seem high in comparison to the textual difference between the source grammars. This is because Rascal transforms the grammar radically when the *Grammar* data type is made.

To make a comparison with an existing tree diff algorithm, all of Derric's grammars have been transformed into an XML representation. The names of constructors are transformed into uppercase to be able to identify which XML elements are nodes. The

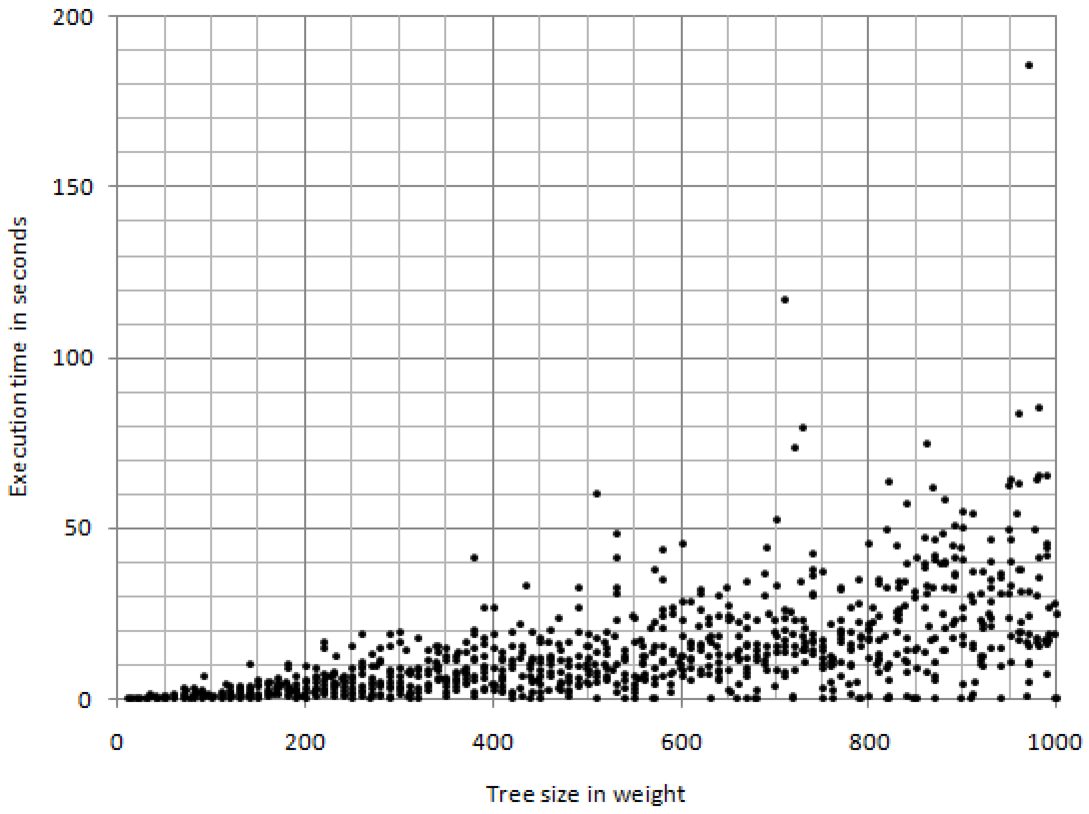


Figure 4.4: The results of the large tree sizes test run. The weight of the tree is the highest weight of either (original or altered) tree. The plot cloud becomes less dense after a weight of 510 because no larger original trees where generated.

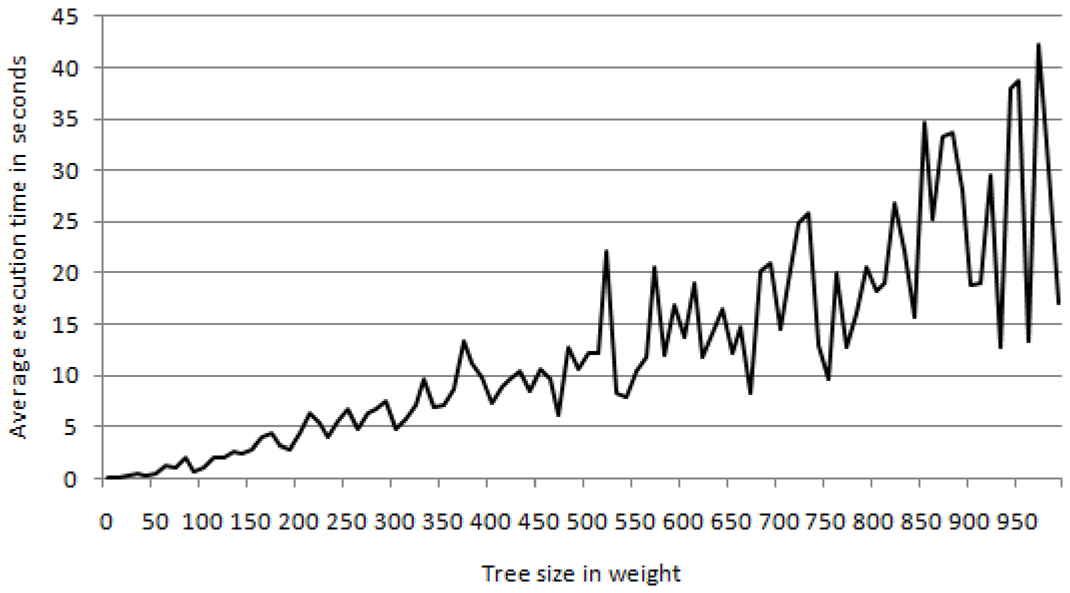


Figure 4.5: The average run times of the large tree sizes test run. The weight of the tree is grouped into the nearest group of ten. High variance in average execution time is caused by an average of 10 samples per data point.

names of the constructors children are displayed in lowercasing and are always the children of the constructors XML element. The XML documents where differenced using DeltaXML[6], an ordered tree diff algorithm. The textual differences between the source grammars are also provided.

Since the Derric grammar generates large trees (between 5487 and 5904 elements), unchanged parts of the tree are omitted.

[6]http://www.deltaxml.com

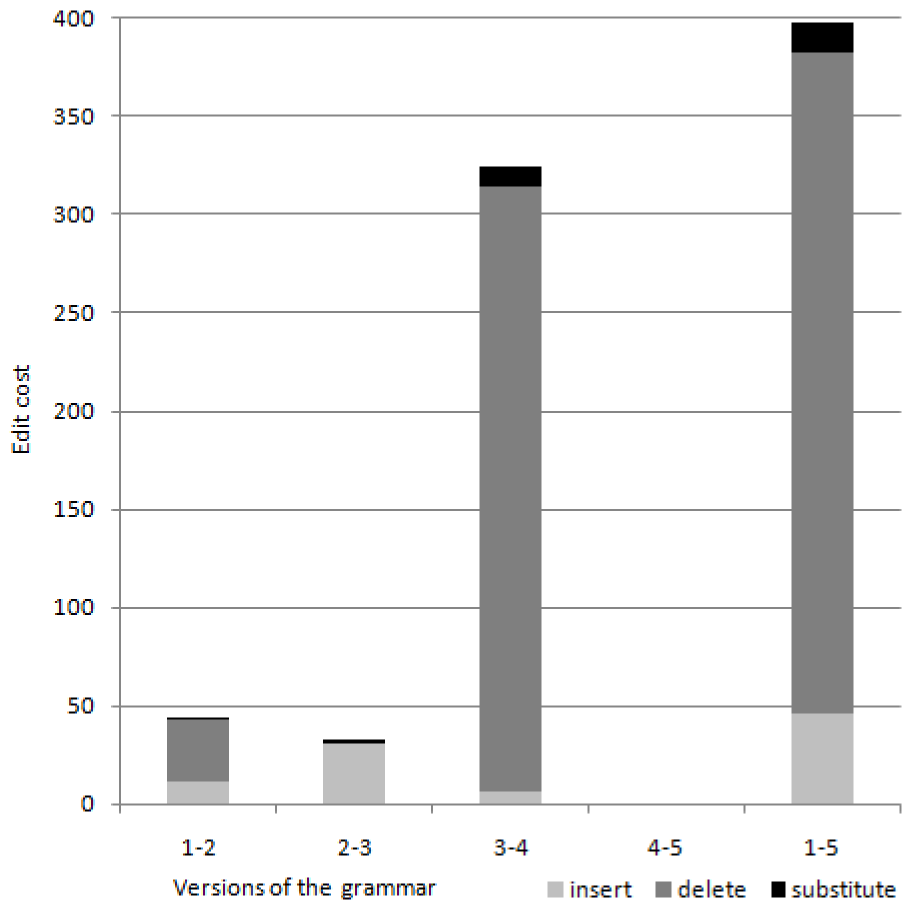
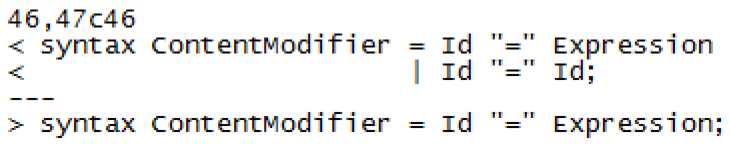


Figure 4.6: The edit cost for each edit operation between the versions of Derric's grammar.



```
46,47c46
< syntax ContentModifier = Id "=" Expression
<                        | Id "=" Id;
---
> syntax ContentModifier = Id "=" Expression;
```

Figure 4.7: The textual difference between version 1 and 2 of Derric's grammar.

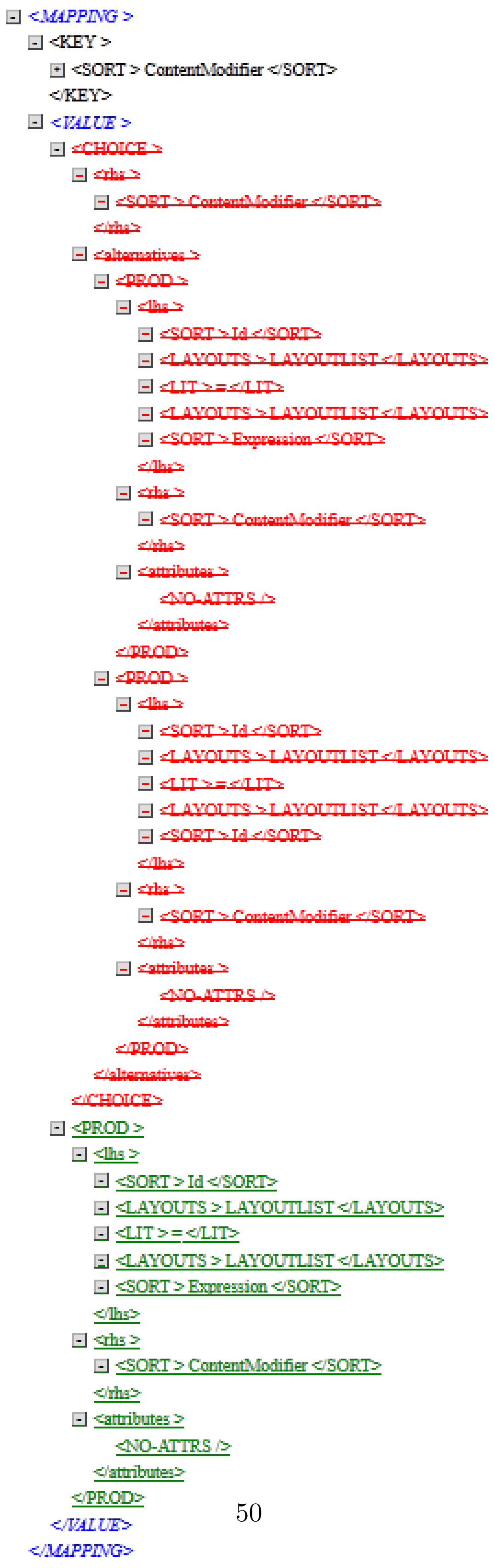
4.4.1 Versions 1 and 2

The evolution from version 1 to 2 involves the modification of the non-terminal *ContentModifier* (see figure 4.7). It removes the option *Id "=" Id*, since this was already covered in the non-terminal *Expression*. Preprocessing the two grammars took 1.372 seconds and calculating the difference between them took 0.187 seconds.

In XML, the difference is the replacement of the value of the *ContentModifier* mapping (see figure 4.8).

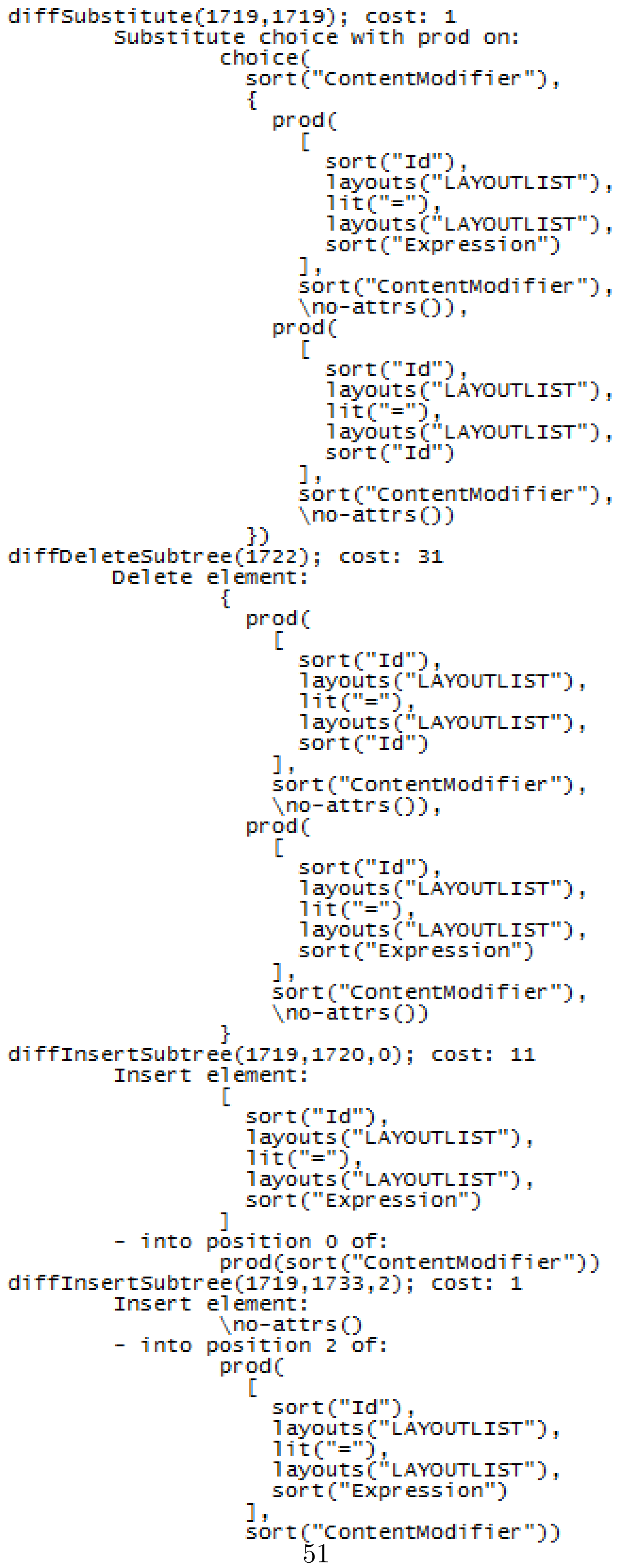
cfgDiff replaces the value of the *ContentModifier* mapping as well (see figure 4.9). However, it replaces only parts of it, it leaves the *rhs* child intact. First it substitutes the name of the *CHOICE* constructor with *PROD*[7]. Then it removes the *alternatives* child from the *PROD* constructor. Then it inserts the *lhs* child subtree as the first child of the *PROD* constructor, shifting the already existing child to the second position. Finally it inserts the *attributes* child subtree as the third child of the *PROD* constructor.

[7]Because of this change, the names of the children will automatically change and a third empty child will be added



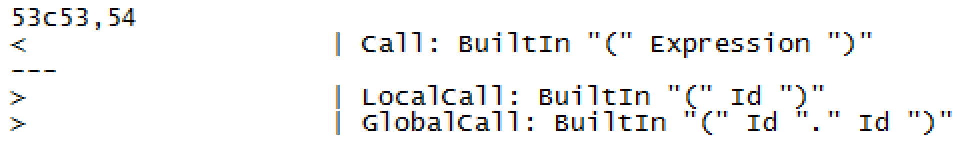
- *<MAPPING >*
  - <KEY >
    - <SORT > ContentModifier </SORT>
  - </KEY>
  - *<VALUE >*
    - ~~<CHOICE >~~
      - ~~<rhs >~~
        - ~~<SORT > ContentModifier </SORT>~~
      - ~~</rhs>~~
      - ~~<alternatives >~~
        - ~~<PROD >~~
          - ~~<lhs >~~
            - ~~<SORT > Id </SORT>~~
            - ~~<LAYOUTS > LAYOUTLIST </LAYOUTS>~~
            - ~~<LIT > = </LIT>~~
            - ~~<LAYOUTS > LAYOUTLIST </LAYOUTS>~~
            - ~~<SORT > Expression </SORT>~~
          - ~~</lhs>~~
          - ~~<rhs >~~
            - ~~<SORT > ContentModifier </SORT>~~
          - ~~</rhs>~~
          - ~~<attributes >~~
            - ~~<NO-ATTRS />~~
          - ~~</attributes>~~
        - ~~</PROD>~~
        - ~~<PROD >~~
          - ~~<lhs >~~
            - ~~<SORT > Id </SORT>~~
            - ~~<LAYOUTS > LAYOUTLIST </LAYOUTS>~~
            - ~~<LIT > = </LIT>~~
            - ~~<LAYOUTS > LAYOUTLIST </LAYOUTS>~~
            - ~~<SORT > Id </SORT>~~
          - ~~</lhs>~~
          - ~~<rhs >~~
            - ~~<SORT > ContentModifier </SORT>~~
          - ~~</rhs>~~
          - ~~<attributes >~~
            - ~~<NO-ATTRS />~~
          - ~~</attributes>~~
        - ~~</PROD>~~
      - ~~</alternatives>~~
    - ~~</CHOICE>~~
    - <PROD >
      - <lhs >
        - <SORT > Id </SORT>
        - <LAYOUTS > LAYOUTLIST </LAYOUTS>
        - <LIT > = </LIT>
        - <LAYOUTS > LAYOUTLIST </LAYOUTS>
        - <SORT > Expression </SORT>
      - </lhs>
      - <rhs >
        - <SORT > ContentModifier </SORT>
      - </rhs>
      - <attributes >
        - <NO-ATTRS />
      - </attributes>
    - </PROD>
  - *</VALUE>*
- *</MAPPING>*

Figure 4.8: The difference between the XML representations of version 1 and 2 of Derric's grammar.



```
diffSubstitute(1719,1719); cost: 1
        Substitute choice with prod on:
                choice(
                  sort("ContentModifier"),
                  {
                    prod(
                      [
                        sort("Id"),
                        layouts("LAYOUTLIST"),
                        lit("="),
                        layouts("LAYOUTLIST"),
                        sort("Expression")
                      ],
                      sort("ContentModifier"),
                      \no-attrs()),
                    prod(
                      [
                        sort("Id"),
                        layouts("LAYOUTLIST"),
                        lit("="),
                        layouts("LAYOUTLIST"),
                        sort("Id")
                      ],
                      sort("ContentModifier"),
                      \no-attrs())
                  })
diffDeleteSubtree(1722); cost: 31
        Delete element:
                {
                  prod(
                    [
                      sort("Id"),
                      layouts("LAYOUTLIST"),
                      lit("="),
                      layouts("LAYOUTLIST"),
                      sort("Id")
                    ],
                    sort("ContentModifier"),
                    \no-attrs()),
                  prod(
                    [
                      sort("Id"),
                      layouts("LAYOUTLIST"),
                      lit("="),
                      layouts("LAYOUTLIST"),
                      sort("Expression")
                    ],
                    sort("ContentModifier"),
                    \no-attrs())
                }
diffInsertSubtree(1719,1720,0); cost: 11
        Insert element:
                [
                  sort("Id"),
                  layouts("LAYOUTLIST"),
                  lit("="),
                  layouts("LAYOUTLIST"),
                  sort("Expression")
                ]
        - into position 0 of:
                prod(sort("ContentModifier"))
diffInsertSubtree(1719,1733,2); cost: 1
        Insert element:
                \no-attrs()
        - into position 2 of:
                prod(
                  [
                    sort("Id"),
                    layouts("LAYOUTLIST"),
                    lit("="),
                    layouts("LAYOUTLIST"),
                    sort("Expression")
                  ],
                  sort("ContentModifier"))
```

Figure 4.9: The output of *cfgDiff* on version 1 and 2 of Derric's grammar.



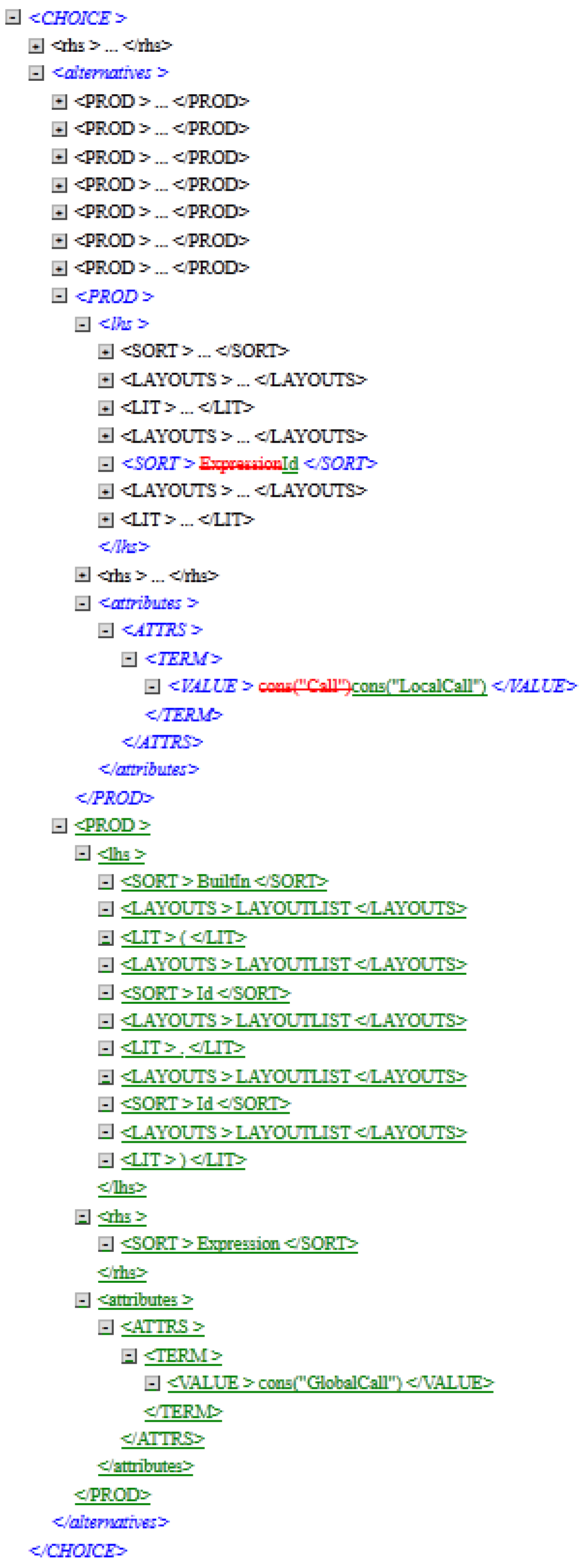
```
53c53,54
<                    | Call: BuiltIn "(" Expression ")"
---
>                    | LocalCall: BuiltIn "(" Id ")"
>                    | GlobalCall: BuiltIn "(" Id "." Id ")"
```

Figure 4.10: The textual difference between version 2 and 3 of Derric's grammar.

4.4.2 Versions 2 and 3

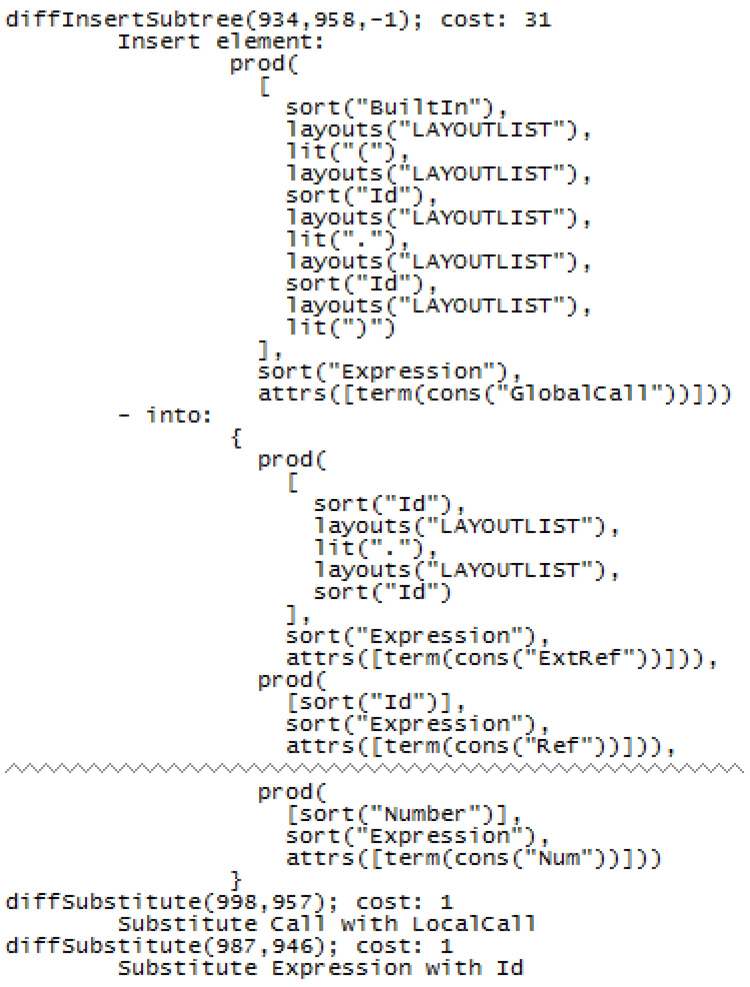
The evolution from version 1 to 2 involves a change made to function calls (see figure 4.10). It changes the syntax of the parameter to only allow an *Id* or *Id "." Id* and labels them differently. Preprocessing the two grammars took 1.326 seconds and calculating the difference between them took 0.421 seconds.

In XML, the difference is the two text replacements and the insertion of the *PROD* constructor (see figure 4.11). *cfgDiff* mimics these changes (see figure 4.12).



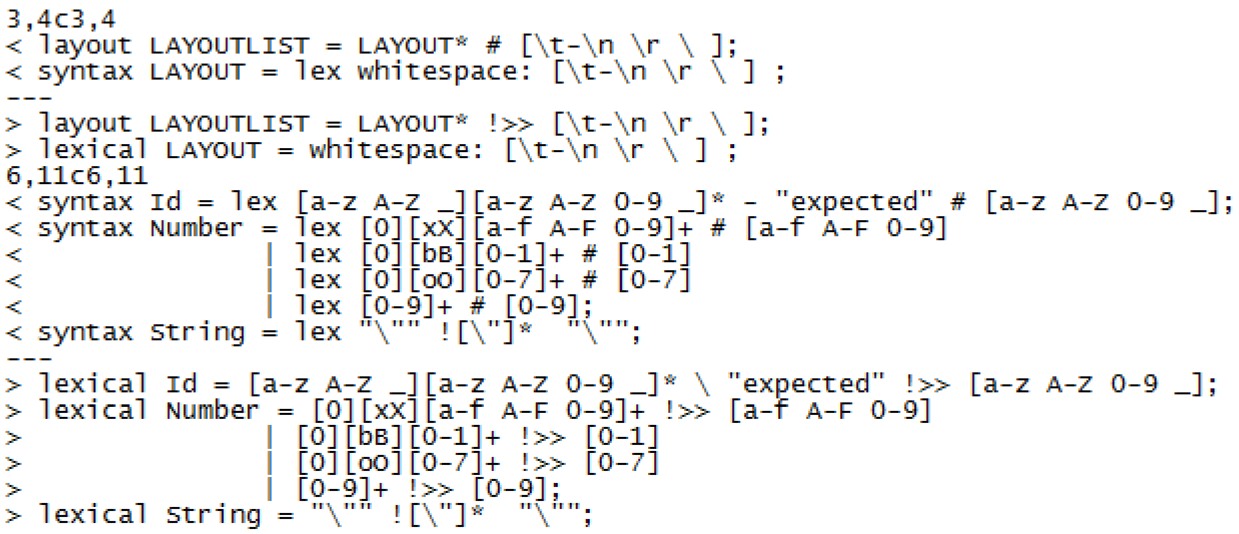
```
<CHOICE >
  <rhs > ... </rhs>
  <alternatives >
    <PROD > ... </PROD>
    <PROD > ... </PROD>
    <PROD > ... </PROD>
    <PROD > ... </PROD>
    <PROD > ... </PROD>
    <PROD > ... </PROD>
    <PROD > ... </PROD>
    <PROD >
      <lhs >
        <SORT > ... </SORT>
        <LAYOUTS > ... </LAYOUTS>
        <LIT > ... </LIT>
        <LAYOUTS > ... </LAYOUTS>
        <SORT > ~~Expression~~Id </SORT>
        <LAYOUTS > ... </LAYOUTS>
        <LIT > ... </LIT>
      </lhs>
      <rhs > ... </rhs>
      <attributes >
        <ATTRS >
          <TERM >
            <VALUE > ~~cons("Call")~~cons("LocalCall") </VALUE>
          </TERM>
        </ATTRS>
      </attributes>
    </PROD>
    <PROD >
      <lhs >
        <SORT > BuiltIn </SORT>
        <LAYOUTS > LAYOUTLIST </LAYOUTS>
        <LIT > ( </LIT>
        <LAYOUTS > LAYOUTLIST </LAYOUTS>
        <SORT > Id </SORT>
        <LAYOUTS > LAYOUTLIST </LAYOUTS>
        <LIT > . </LIT>
        <LAYOUTS > LAYOUTLIST </LAYOUTS>
        <SORT > Id </SORT>
        <LAYOUTS > LAYOUTLIST </LAYOUTS>
        <LIT > ) </LIT>
      </lhs>
      <rhs >
        <SORT > Expression </SORT>
      </rhs>
      <attributes >
        <ATTRS >
          <TERM >
            <VALUE > cons("GlobalCall") </VALUE>
          </TERM>
        </ATTRS>
      </attributes>
    </PROD>
  </alternatives>
</CHOICE>
```

Figure 4.11: The difference between the XML representations of version 2 and 3 of Derric's grammar.



```
diffInsertSubtree(934,958,-1); cost: 31
        Insert element:
                prod(
                  [
                    sort("BuiltIn"),
                    layouts("LAYOUTLIST"),
                    lit("("),
                    layouts("LAYOUTLIST"),
                    sort("Id"),
                    layouts("LAYOUTLIST"),
                    lit("."),
                    layouts("LAYOUTLIST"),
                    sort("Id"),
                    layouts("LAYOUTLIST"),
                    lit(")")
                  ],
                  sort("Expression"),
                  attrs([term(cons("GlobalCall"))]))
        - into:
                {
                  prod(
                    [
                      sort("Id"),
                      layouts("LAYOUTLIST"),
                      lit("."),
                      layouts("LAYOUTLIST"),
                      sort("Id")
                    ],
                    sort("Expression"),
                    attrs([term(cons("ExtRef"))])),
                  prod(
                    [sort("Id")],
                    sort("Expression"),
                    attrs([term(cons("Ref"))])),

                  prod(
                    [sort("Number")],
                    sort("Expression"),
                    attrs([term(cons("Num"))]))
                }
diffSubstitute(998,957); cost: 1
        Substitute Call with LocalCall
diffSubstitute(987,946); cost: 1
        Substitute Expression with Id
```

Figure 4.12: The output of *cfgDiff* on version 2 and 3 of Derric's grammar.



```
3,4c3,4
< layout LAYOUTLIST = LAYOUT* # [\t-\n \r \ ];
< syntax LAYOUT = lex whitespace: [\t-\n \r \ ] ;
---
> layout LAYOUTLIST = LAYOUT* !>> [\t-\n \r \ ];
> lexical LAYOUT = whitespace: [\t-\n \r \ ] ;
6,11c6,11
< syntax Id = lex [a-z A-Z _][a-z A-Z 0-9 _]* - "expected" # [a-z A-Z 0-9 _];
< syntax Number = lex [0][xX][a-f A-F 0-9]+ # [a-f A-F 0-9]
<                | lex [0][bB][0-1]+ # [0-1]
<                | lex [0][oO][0-7]+ # [0-7]
<                | lex [0-9]+ # [0-9];
< syntax String = lex "\"" ![\"]*  "\"";
---
> lexical Id = [a-z A-Z _][a-z A-Z 0-9 _]* \ "expected" !>> [a-z A-Z 0-9 _];
> lexical Number = [0][xX][a-f A-F 0-9]+ !>> [a-f A-F 0-9]
>                | [0][bB][0-1]+ !>> [0-1]
>                | [0][oO][0-7]+ !>> [0-7]
>                | [0-9]+ !>> [0-9];
> lexical String = "\"" ![\"]*  "\"";
```

Figure 4.13: The textual difference between version 3 and 4 of Derric's grammar.

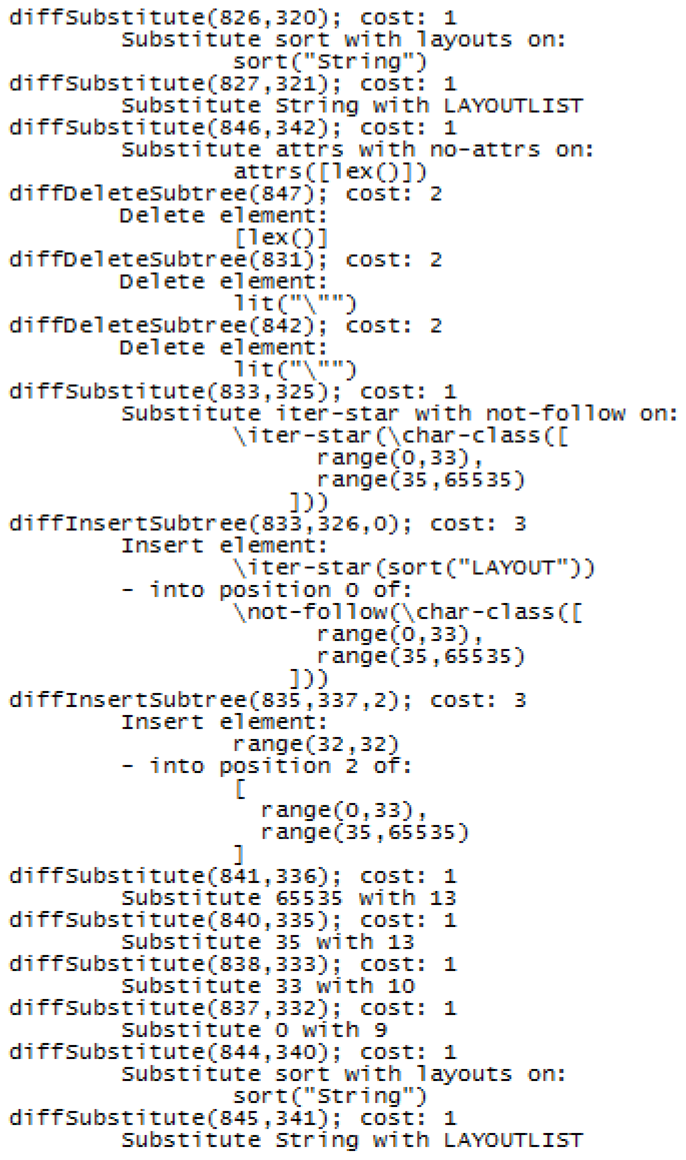
4.4.3 Versions 3 and 4

The evolution from version 3 to 4 refactors terminals (see figure 4.13). Until now, all terminals were incorrectly declared using *syntax*. Rascal normally uses *lexical* to define terminal symbols. Also the way the follow restrictions were written has been changed. Preprocessing the two grammars took 1.263 seconds and calculating the difference between them took 1.138 seconds.

Because of the extreme length of the output of DeltaXML, its changes are omitted for briefness. DeltaXML deleted 7 nodes in the XML structure and inserted 1 node. *cfgDiff*'s changes mimic the changes DeltaXML detected, but uses 8 deletes, 10 substitutes and 2 insertions in its edit script (see figure 4.14).

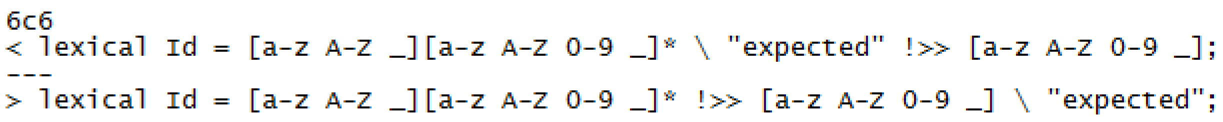
```
diffDeleteSubtree(849); cost: 73
        Delete element:
                <sort("Id"),{diff(
                    sort("Id"),
                    restrict(
                      sort("Id"),
                      prod(
                        [
                          \char-class([
                              range(65,90),
                              range(95,95),
                              range(97,122)
                            ]),
                          \iter-star(\char-class([
                                range(48,57),
                                range(65,90),
                                range(95,95),
                                range(97,122)
                              ]))
                        ],
                        sort("Id"),
                        attrs([lex()])),
                      {prod(
                          [\char-class([
                                range(48,57),
                                range(65,90),
                                range(95,95),
                                range(97,122)
                              ])],
                          restricted(sort("Id")),
                          \no-attrs())}),
                    {prod(
                        [lit("expected")],
                        sort("Id"),
                        attrs([reject()]))})}>
diffDeleteSubtree(2496); cost: 15
        Delete element:
                <lit("\""),{prod(
                    [\char-class([range(34,34)])],
                    lit("\""),
                    attrs([literal()]))}>
diffDeleteSubtree(319); cost: 32
        Delete element:
                <layouts("LAYOUTLIST"),{restrict(
                    layouts("LAYOUTLIST"),
                    prod(
                      [\iter-star(sort("LAYOUT"))],
                      layouts("LAYOUTLIST"),
                      \no-attrs()),
                    {prod(
                        [\char-class([
                              range(9,10),
                              range(13,13),
                              range(32,32)
                            ])],
                        restricted(layouts("LAYOUTLIST")),
                        \no-attrs())})}>
diffDeleteSubtree(707); cost: 24
        Delete element:
                <sort("LAYOUT"),{prod(
                    [\char-class([
                          range(9,10),
                          range(13,13),
                          range(32,32)
                        ])],
                    sort("LAYOUT"),
                    attrs([
                        term(cons("whitespace")),
                        lex()
                      ]))}>
```

```
diffDeleteSubtree(2290); cost: 158
        Delete element:
                <sort("Number"),{restrict(
                    sort("Number"),
                    choice(
                      sort("Number"),
                      {
                        prod(
                          [
                            \char-class([range(48,48)]),
                            \char-class([
                                range(79,79),
                                range(111,111)
                              ]),
                            iter(\char-class([range(48,55)]))
                          ],
                          sort("Number"),
                          attrs([lex()])),
                        prod(
                          [iter(\char-class([range(48,57)]))],
                          sort("Number"),
                          attrs([lex()])),
                        prod(
                          [
                            \char-class([range(48,48)]),
                            \char-class([
                                range(66,66),
                                range(98,98)
                              ]),
                            iter(\char-class([range(48,49)]))
                          ],
                          sort("Number"),
                          attrs([lex()])),
                        prod(
                          [
                            \char-class([range(48,48)]),
                            \char-class([
                                range(88,88),
                                range(120,120)
                              ]),
                            iter(\char-class([
                                  range(48,57),
                                  range(65,70),
                                  range(97,102)
                                ]))
                          ],
                          sort("Number"),
                          attrs([lex()]))
                      }),
                    {
                      prod(
                        [\char-class([range(48,57)])],
                        restricted(sort("Number")),
                        \no-attrs()),
                      prod(
                        [\char-class([range(48,55)])],
                        restricted(sort("Number")),
                        \no-attrs()),
                      prod(
                        [\char-class([range(48,49)])],
                        restricted(sort("Number")),
                        \no-attrs()),
                      prod(
                        [\char-class([
                              range(48,57),
                              range(65,70),
                              range(97,102)
                            ])],
                        restricted(sort("Number")),
                        \no-attrs())
                    })}>
```



```
diffSubstitute(826,320); cost: 1
        Substitute sort with layouts on:
                sort("String")
diffSubstitute(827,321); cost: 1
        Substitute String with LAYOUTLIST
diffSubstitute(846,342); cost: 1
        Substitute attrs with no-attrs on:
                attrs([lex()])
diffDeleteSubtree(847); cost: 2
        Delete element:
                [lex()]
diffDeleteSubtree(831); cost: 2
        Delete element:
                lit("\"")
diffDeleteSubtree(842); cost: 2
        Delete element:
                lit("\"")
diffSubstitute(833,325); cost: 1
        Substitute iter-star with not-follow on:
                \iter-star(\char-class([
                      range(0,33),
                      range(35,65535)
                    ]))
diffInsertSubtree(833,326,0); cost: 3
        Insert element:
                \iter-star(sort("LAYOUT"))
        - into position 0 of:
                \not-follow(\char-class([
                      range(0,33),
                      range(35,65535)
                    ]))
diffInsertSubtree(835,337,2); cost: 3
        Insert element:
                range(32,32)
        - into position 2 of:
                [
                  range(0,33),
                  range(35,65535)
                ]
diffSubstitute(841,336); cost: 1
        Substitute 65535 with 13
diffSubstitute(840,335); cost: 1
        Substitute 35 with 13
diffSubstitute(838,333); cost: 1
        Substitute 33 with 10
diffSubstitute(837,332); cost: 1
        Substitute 0 with 9
diffSubstitute(844,340); cost: 1
        Substitute sort with layouts on:
                sort("String")
diffSubstitute(845,341); cost: 1
        Substitute String with LAYOUTLIST
```

Figure 4.14: The output of *cfgDiff* on version 3 and 4 of Derric's grammar.



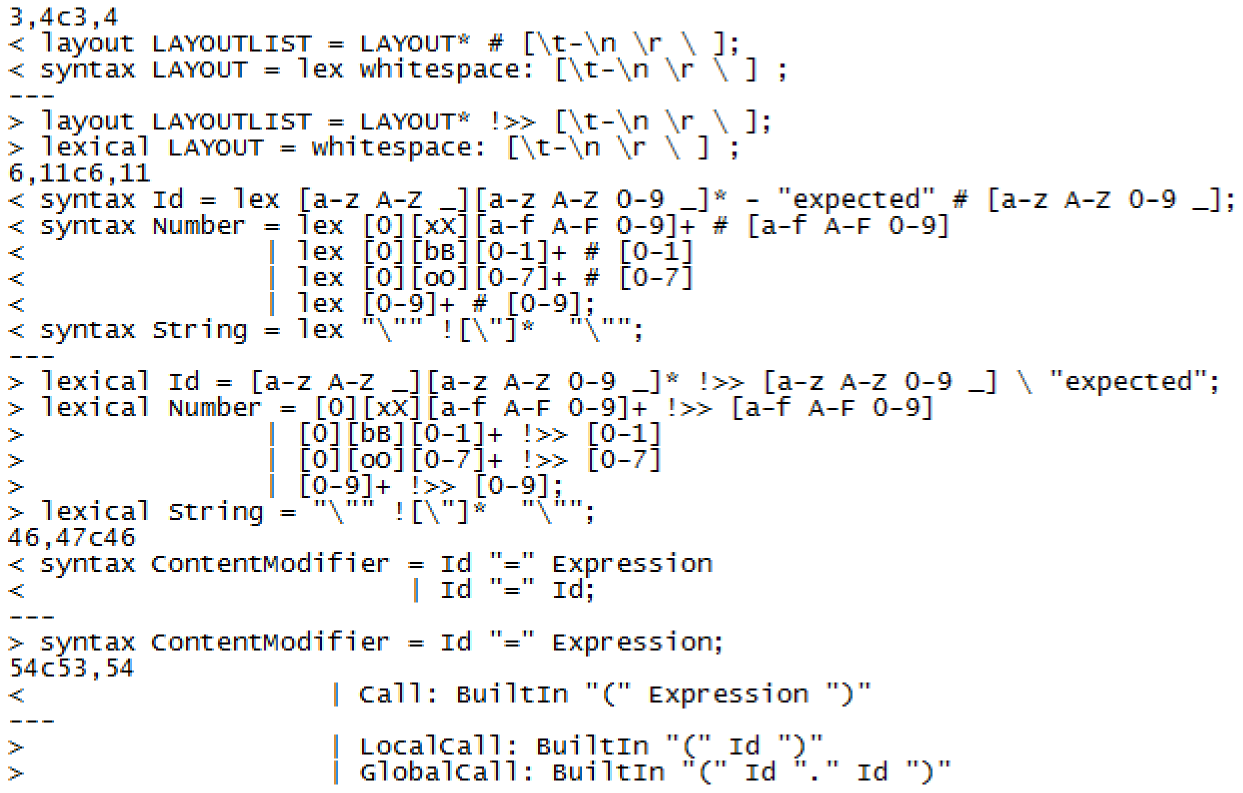
```
6c6
< lexical Id = [a-z A-Z _][a-z A-Z 0-9 _]* \ "expected" !>> [a-z A-Z 0-9 _];
---
> lexical Id = [a-z A-Z _][a-z A-Z 0-9 _]* !>> [a-z A-Z 0-9 _] \ "expected";
```

Figure 4.15: The textual difference between version 4 and 5 of Derric's grammar.

4.4.4 Versions 4 and 5

The evolution from version 4 to 5 reorders the follow restrictions on the terminal *Id* (see figure 4.15). However, this has no influence on the contents of the *Grammar* data type. Therefore, *cfgDiff* did not detect any changes. Preprocessing the two grammars took 1.232 seconds and calculating the difference between them took less than 0.016 seconds.

Because the ordering in the *Grammar* data type isn't changed, DeltaXML did not detect any changes either.



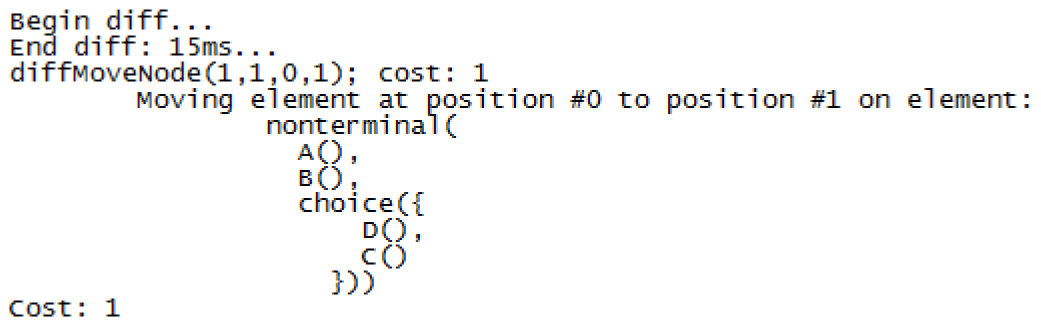
```
3,4c3,4
< layout LAYOUTLIST = LAYOUT* # [\t-\n \r \ ];
< syntax LAYOUT = lex whitespace: [\t-\n \r \ ] ;
---
> layout LAYOUTLIST = LAYOUT* !>> [\t-\n \r \ ];
> lexical LAYOUT = whitespace: [\t-\n \r \ ] ;
6,11c6,11
< syntax Id = lex [a-z A-Z _][a-z A-Z 0-9 _]* - "expected" # [a-z A-Z 0-9 _];
< syntax Number = lex [0][xX][a-f A-F 0-9]+ # [a-f A-F 0-9]
<               | lex [0][bB][0-1]+ # [0-1]
<               | lex [0][oO][0-7]+ # [0-7]
<               | lex [0-9]+ # [0-9];
< syntax String = lex "\"" ![\"]*  "\"";
---
> lexical Id = [a-z A-Z _][a-z A-Z 0-9 _]* !>> [a-z A-Z 0-9 _] \ "expected";
> lexical Number = [0][xX][a-f A-F 0-9]+ !>> [a-f A-F 0-9]
>                | [0][bB][0-1]+ !>> [0-1]
>                | [0][oO][0-7]+ !>> [0-7]
>                | [0-9]+ !>> [0-9];
> lexical String = "\"" ![\"]*  "\"";
46,47c46
< syntax ContentModifier = Id "=" Expression
<                        | Id "=" Id;
---
> syntax ContentModifier = Id "=" Expression;
54c53,54
<                    | Call: BuiltIn "(" Expression ")"
---
>                    | LocalCall: BuiltIn "(" Id ")"
>                    | GlobalCall: BuiltIn "(" Id "." Id ")"
```

Figure 4.16: The textual difference between version 1 and 5 of Derric's grammar.

4.4.5 Versions 1 and 5

For completeness I've included the changes between the first and last version of Derric's grammar (see figure 4.16). Because they include all the changes described in the previous sections, they will not be described again here. Preprocessing the two grammars took 1.341 seconds and calculating the difference between them took 35.131 seconds.

Because of the extreme length of the changes between the processed grammars, the output of both DeltaXML and *cfgDiff* are omitted. DeltaXML deleted 7 nodes in the XML structure, substituted 2 (2 deletes and 2 insertions), and inserted 3 node. *cfgDiff* needed 10 deletes, 16 substitutes and 8 insertions for its edit script.



```
Begin diff...
End diff: 15ms...
diffMoveNode(1,1,0,1); cost: 1
        Moving element at position #0 to position #1 on element:
                nonterminal(
                  A(),
                  B(),
                  choice({
                      D(),
                      C()
                    }))
Cost: 1
```

Figure 4.17: The output of *cfgDiff* on the example from section 2.1.2.

4.5 Shortness of the edit script

In some cases the edit script has a higher cost than the generated changes. This is caused by an optimization in the *diffList* function. An algorithm is used to find which identical elements should be matched up in pairs. Sometimes a copy of an element exists in a list and has an identical element in the other list, for example:

["X", "B", "B"] and ["Z", "B", "C"]

The algorithm can pick either "B" from the first list and pair it with the "B" from the second list. However, because one "B" can be substituted for a "C", which "B" is picked is important for the cost of the edit script. If the first "B" of the first list gets paired up with the "B" of the second list, the second "B" can be substituted with a "C", resulting in one substitute and one more for changing "X" into "Z". If the second "B" of the first list gets paired up with the "B" of the second list, the first "B" will be substituted with "C" and moved to the last position[8], resulting in one substitute, one move and one more for changing "X" into "Z".

Because the algorithm that picks the pairs has no knowledge of the actions that happen later on with the picked pairs, the algorithm sometimes picks the wrong pair of items, resulting in an extra move operation. Because the *diffList* algorithm tries to remove identical elements from the beginning and ending of a list, these cases only happen when the elements are placed between nonidentical elements.

4.6 Output of example from section 2.1.2

Section 2.1.2 provided an example to show the shortcomings in existing diff algorithms. Since *cfgDiff* is designed for context-free grammars, its output (see figure 4.17) should show improvements over the existing work.

[8]Or deleting the first "B" and inserting "C", which has the same cost as one substitute and one move.

cfgDiff only recognizes the only semantical difference between the two parts of the grammar, the reordering of A and B. The reordering of the unordered C and D is ignored. This deviates from the results of other diff algorithms. The text-based diff algorithm and ordered tree diff algorithms also recognized the reordering of C and D, which is semantically irrelevant. Unordered tree diff algorithms does not see any changes at all. They ignore the reordering of A and B, which is relevant in a context-free grammar.

Chapter 5

Evaluation

This chapter analyses the *cfgDiff* algorithm by comparing it to X-Diff[20].

5.1 Differences with X-Diff

cfgDiff has been based on X-Diff[20]. Although X-Diff is comparable to *cfgDiff*, it differences XML documents with an ordered model, the development of *cfgDiff* changed the algorithm drastically.

5.1.1 Edit operations

X-Diff lacks a move operation, although it works on an ordered model. Instead, insert and delete operations are used. This can result in costlier edit scripts, but helps decreases run times.

Furthermore, X-Diff only supports the substitute operation on leaf nodes (text nodes or attribute nodes in XML documents). Since XML elements should not change from one type into another, only leaf nodes can be substituted.

5.1.2 Different algorithm to find the best set of changes

Because *cfgDiff* supports move operations, which X-Diff does not, another algorithm needed to be developed to find the best set of changes between two collections. X-Diff uses the minimum cost maximum flow[**?**] algorithm to find the best set of changes. Because *cfgDiff* has to move element pairs which are not in the flow, the algorithm has to take these changes into account as well. Therefore the minimum cost maximum flow algorithm wouldn't suffice in this situation and the *findBestChanges* algorithm has been introduced.

5.1.3 Early rejection

Because substitution is not supported on XML elements, the early rejection model rejects any comparisons based on the mismatching names of XML elements. *cfgDiff* deviates from this model to support substitution on constructors. This feature is implemented because the constructors in grammars contain information about how their children need to be handled; XML elements only describe what information they contain.

5.1.4 Hashing

X-Diff uses hashing to speed up element equality checks. Because hashing slowed *cfgDiff* down too much, hashing is not implemented in the final version of *cfgDiff*.

5.1.5 Bottom up vs. top down

X-Diff uses a bottom up approach. The first version of *cfgDiff* used the same approach, but was significantly slower than its top down counterpart. The difference in efficiency can be explained by difference in the early rejection models and the data they work on. The names of XML elements in XML documents are diverse. X-Diff can effectively reject many element comparisons because of this. Because *cfgDiff* supports substitute operations on *node*s, this early rejection rule can not be implemented in *cfgDiff*. Therefore, no *node* to *node* comparisons can be rejected. To make matters worse, grammars mainly consist of *node*s.

5.2 Trade offs

5.2.1 Untyped data types

cfgDiff treats all constructors as untyped nodes. This way all constructors are allowed to be transformed in any other constructor. The algorithm can include a list of valid transformations to restrict the transformations and serve as rejection rules, decreasing the run time of the algorithm. However, this would require time to describe valid transformations and maintain this list when Rascal's grammar data types are updated.

Because developing and maintaining such a list is labor intensive, the algorithm was developed to work on untyped data types. As a side effect parts of the algorithm can be reused to difference other data structures as well.

5.2.2 Eliminating matching elements

The *diffList* function removes matching elements to reduce the matching space and decrease the run time. However, this may result in additional move operations (see section 4.5). Because the conditions on which this occurs are rare and the savings on elements where subtrees match are large, this trade off was implemented.

5.2.3 Describing edit operations in tree edit operations

Due to time constraints on this research project, a feature to transform tree edit operations into more descriptive grammar edit operations has not been implemented. The amount of time for research and testing this feature would not fit in the tight schedule of this project.

5.3 Weaknesses

5.3.1 Possibly long run times on large lists

The *findBestChanges* function tries all possible mutations in order to find the cheapest edit script. This is necessarily to allow move edit operations. If move edit operations were not be supported, a faster algorithm could be used instead. Because *findBestChanges* was to enumerates all possible mutations, it can result in long run times when a grammar contains large lists.

5.3.2 Inability to detect elements inserted within the tree

Some modifications to grammars can result in elements that are inserted within the tree, as opposed to inserting them as leaves. Because *cfgDiff* does not support this operation, modifications like these can result in the deletion and insertion of whole subtrees. Implementing this feature would severely increase run times.

5.3.3 Inability to differences in grammar edit operations

The edit scripts *cfgDiff* produces consists of tree edit operations. The usability and user friendliness of the tool would benefit from edit scripts which contain grammar edit operations, since they are more descriptive. Also, a tool which pinpoints the original sections of the grammars which are associated with each edit operation would benefit the user experience.

Chapter 6

Conclusion

The *cfgDiff* algorithm is a tree differencing algorithm which is developed for context-free grammars. Although it lacks the transformation of tree edit operations into grammar edit operations, it is a improvement over existing alternatives. It can handle both ordered and unordered collections within the same tree, something that happens in most grammars. Because of this feature, *cfgDiff* is able to detect changes without the unnecessary or missing changes ordered and unordered tree diff algorithm detect.

6.1 Threats to validity

The algorithm has been tested on random trees due to the complexity of generating valid grammars and the time it would take to generate and validate such test cases. Therefore it is possible that the algorithm may function differently on grammars than it did on random trees.

Also, some statistics presented in this thesis may be off due to small sample sizes, as in figure 4.5. Furthermore, the statistics may be less precise because of the low resolution of the *cpuTime* function. Its resolution is 15.6 milliseconds on the test machine, which may affect some statistics.

6.2 Future work

The points described in section 5.3 provide improvements that can be made to the algorithm. Addressing these points may result in an improved version of a function or a totally different implementation.

Furthermore, *cfgDiff* provides an edit script containing tree edit operations. Work can be done to transform the tree edit operations into grammar edit operations, for example 'substitute non-terminal X'. This way, the edit script becomes more readable because it provides a context in which the change has been detected. These grammar edit operations may even be transformed into higher level grammar edit operations, such as the inlining of a non-terminal.

Bibliography

[1] Canfora, G., Cerulo, L., and Di Penta, M. Tracking your changes: A language-independent approach. *IEEE Softw. 26* (January 2009), 50–57.

[2] Chawathe, S. S., Rajaraman, A., Garcia-Molina, H., and Widom, J. Change detection in hierarchically structured information. *SIGMOD Rec. 25* (June 1996), 493–504.

[3] Chen, Y. F., Douglis, F., Huang, H., and Vo, K. P. TopBlend: An efficient implementation of HtmlDiff in Java. In *Proceedings of the WebNet2000 Conference* (Nov. 2000).

[4] Cobena, G., Abiteboul, S., and Marian, A. Detecting changes in XML documents. In *In ICDE* (2001), pp. 41–52.

[5] Dulucq, S., and Touzet, H. Analysis of tree edit distance algorithms. In *Proceedings of the 14th annual conference on Combinatorial pattern matching* (Berlin, Heidelberg, 2003), CPM'03, Springer-Verlag, pp. 83–95.

[6] Garcia-Molina, H., and Labio, W. J. Efficient snapshot differential algorithms for data warehousing. Tech. rep., Stanford, CA, USA, 1996.

[7] Girschick, M., and Darmstadt, T. Difference detection and visualization in UML class diagrams. Tech. rep., TU Darmstadt, 2006.

[8] Hirschberg, D. S. A linear space algorithm for computing maximal common subsequences. *Commun. ACM 18* (June 1975), 341–343.

[9] Jiang, T., Wang, L., and Zhang, K. Alignment of trees - an alternative to tree edit. In *Proceedings of the 5th Annual Symposium on Combinatorial Pattern Matching* (London, UK, 1994), CPM '94, Springer-Verlag, pp. 75–86.

[10] Klint, P., Lämmel, R., and Verhoef, C. Toward an engineering discipline for grammarware. *ACM Trans. Softw. Eng. Methodol. 14* (July 2005), 331–380.

[11] Kuhn, H. The Hungarian method for the assignment problem. *Naval research logistics quarterly 2*, 1-2 (1955), 83–97.

[12] Masek, W. J., and Paterson, M. A faster algorithm computing string edit distances. *J. Comput. Syst. Sci. 20*, 1 (1980), 18–31.

[13] Myers, E. W. An O(ND) difference algorithm and its variations. *Algorithmica 1* (1986), 251–266.

[14] Sankoff, D. Matching sequences under deletion-insertion constraints. *Proceedings of the Natural Academy of Sciences of the U.S.A. 69* (1972), 4–6.

[15] Selkow, S. M. The tree-to-tree editing problem. *Inf. Process. Lett. 6*, 6 (1977), 184–186.

[16] van den Bos, J., and van der Storm, T. Bringing domain-specific languages to digital forensics. In *Proceedings of the 33rd International Conference on Software Engineering (ICSE'11)* (2011), ACM. Software Engineering in Practice.

[17] van der Storm, T. Original project description, 2010.

[18] Wagner, R. A., and Fischer, M. J. The string-to-string correction problem. *J. ACM 21* (January 1974), 168–173.

[19] Wang, J. T. L., Zhang, K., Jeong, K., and Shasha, D. A system for approximate tree matching. *IEEE Trans. on Knowl. and Data Eng. 6* (August 1994), 559–571.

[20] Wang, Y., DeWitt, D. J., and yi Cai, J. X-Diff: An effective change detection algorithm for XML documents. In *ICDE* (2003), pp. 519–530.

[21] Williams, C. C., and Spacco, J. W. Branching and merging in the repository. In *Proceedings of the 2008 international working conference on Mining software repositories* (New York, NY, USA, 2008), MSR '08, ACM, pp. 19–22.

[22] Xing, Z., and Stroulia, E. UMLDiff: an algorithm for object-oriented design differencing. In *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering* (New York, NY, USA, 2005), ASE '05, ACM, pp. 54–65.

[23] Xing, Z., and Stroulia, E. Differencing logical UML models. *Automated Software Engg. 14* (June 2007), 215–259.

[24] Yang, W. Identifying syntactic differences between two programs. *Softw. Pract. Exper. 21* (June 1991), 739–755.

[25] Zaytsev, V. *Recovery, Convergence and Documentation of Languages.* PhD thesis, Vrije Universiteit, Amsterdam, The Netherlands, October 2010.

[26] Zhang, K., and Shasha, D. Simple fast algorithms for the editing distance between trees and related problems. *SIAM J. Comput. 18* (December 1989), 1245–1262.