Acta Univ. Sapientiae, Informatica, **1**, 1 (2009) 35–44

Automated subtree construction in a contracted abstract syntax tree

Róbert Kitlei
Faculty of Informatics,
Eötvös Loránd University,
Pázmány Péter sétány 1/c,
H–1117 Budapest, Hungary
email: kitlei@elte.hu

Abstract. Syntax trees are commonly used by compilers to represent the structure of the source code of a program, but they are not convenient enough for other tasks. One such task is refactoring, a technique to improve program code by changing its structure [7].

In this paper, we shortly describe a representation of the abstract syntax tree (AST), which is better suited for the needs of refactoring. This is achieved by contracting nodes and edges in the tree. The representation serves as the basis of the back-end of a prototype Erlang refactoring tool [8], however, it is adaptable to languages different than Erlang [2].

We introduce a method that helps us automatically generate syntactically correct subtrees. Since refactorings often have to create new parts of the tree, it is essential to make this task as convenient as possible.

1 Introduction

Syntax trees are usually created by parsers, which operate on tokens that are produced by a scanner directly from the source code, with possibly a preprocessing phase inserted. Most of the time, these syntax trees are used once – possibly the syntax tree is never constructed in its entirety, as is the

AMS 2000 subject classifications: 68P05 subtree construction
CR Categories and Descriptors: D.2.10. [**Software**]: Software engineering – *Representation*;
Key words and phrases: subtree construction

case with top-down and bottom-up parsers. However, in some cases the full syntax tree is needed. One such example is refactoring.

Refactoring is the systematic changing of source files while retaining the semantics of the code. Some refactorings, e.g. renaming a variable or a function, do not change the shape of the syntax tree, only update the information in the nodes, while others, e.g. extracting a function, do delete, move or insert new nodes or subtrees in the syntax tree. Since deletion does not pose a problem, and moving a subtree is equivalent to its removal and reinsertion, the most intriguing question of the above is the creation and insertion of new subtrees.

In addition to the above, refactorings have to gather additional information about semantic aspects of the source code as well. Since these bits of information can only be collected by visiting diverse parts of a syntax tree, syntax trees make inappropriate and inefficient representations for refactorings. A graph representation is proposed by the Erlang refactoring group at the university ELTE (Budapest, Hungary). The ELTE group proposed this representation after previous experience with refactoring [5, 8]. Details about the representation and the refactoring tool are found in [4].

The structure of the paper is as follows. In section 2, the graph representation is described to such depth as is necessary for understanding the rest of the paper. Section 3 describes a method that facilitates the creation and insertion of new subtrees. This method is the main contribution of the paper. Section 4 lists related work, and section 5 gives acknowledgements.

2 Representation structure

2.1 Node and edge contractions

ASTs built on top of source codes are typically created by compilers in compilation time. Such syntax trees are discarded after they have been used, and their construction does not involve complex traversals: they follow the construction of the tree. There are, however, applications in which the role of ASTs are augmented. In refactoring, for example, tree traversals are extensively used, because a lot of information is required that can be acquired from different locations.

In order to facilitate these traversals, a new representation of the AST was introduced, which is described in detail in [4]. Here we give an overview of the relevant parts of the representation.

ASTs inherently involve parts that are unnecessary for information collection, or are structured so that they make it more tedious. One obvious case

```
if
    X == 1 -> Y = 2;
    true   -> Y = 3
end
```

Figure 1: If clauses in Erlang.

```
to_list(Text) when is_atom(Text)    -> atom_to_list(Text);
to_list(Text) when is_integer(Text) -> integer_to_list(Text);
to_list(Text) when is_float(Text)   -> float_to_list(Text);
to_list(Text) when is_list(Text)    -> Text.
```

Figure 2: Function clauses with guards.

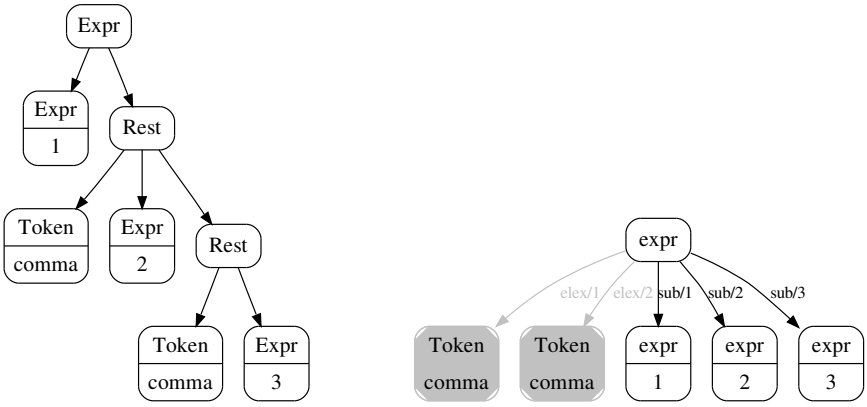
is that of chain rules: the information contained in them could be expressed as a single node, yet the traversing code has to be different for each node that occurs on the way.

Another case can be described by their functionality: the edges of the nodes can be grouped so that one traversal should follow exactly those that are in one group. To give a concrete example, clauses in Erlang have parameters, guard expressions and a body, and there are associated tokens: parentheses and an arrow. Yet the actual appearance of the clauses can be vastly different, see Figures 1 and 2. When collecting information, often either all parameters or all guard expressions are required at a time during a traversal pass, but seldom both at the same time of the traversal. Therefore, it is natural to partition the edges into groups along their uses. Since the partitions depend on the traversals used, the programmer has to decide by hand how groups should be made. This way, only as few groups have to be introduced as needed in a given application.

Another way to make the representation more compact is to contract repetitions. Repetitions are common constructs in programming languages: they are repeated uses of a rule with intercalated tokens as separators. Instead of having a slanted tree as constructed by an AST, it is more convenient for traversal purposes to represent them by a parent node with all of the repeated nodes and the intermediate tokens as its children. As a matter of fact, in the example in the above paragraph the parameters and guard expressions are

already a result of such a contraction.

Having done the above contractions has two main advantages. One is that much fewer cases have to be considered. In the case of Erlang, the grammar contained 69 nonterminals, which was reduced to three contracted node groups: forms, clauses and expressions.



(a) AST. (b) Contracted AST.

Figure 3: Repetition in the expression `1+2+3`.

A further advantage of contractions is that they enable introducing queries, which makes traversals even more effective. Queries can be further optimised by automatically adding semantic nodes and edges to the contracted AST, which make it a graph. In addition, the prototype tool also supports preprocessor constructs. Queries, semantic nodes and edges and preprocessor handling are described in detail in [4].

Since the contraction groups are different for each language (and may even differ in each application, depending on the needed level of detail), it is important that the approach should be adaptable to a wide range of grammars. For this reason, an XML representation was chosen for describing the grammar rules, the contraction groups and the edge labels. The scanner and parser are automatically generated from this file. The contracted structure is immediately constructed during parse time, and not converted from an AST.

2.2 Representation of the contracted AST

The inner nodes of the contracted AST are the contracted nodes, which also contain the originating nonterminal as information. The leaf nodes of the contracted AST are the tokens, which contain the token text and the whitespace before and after the token. The nodes are connected by labelled edges; the labels determine the contraction classes they can connect.

Contractions do not fully preserve edge ordering: order is preserved only between the edges with the same label, not between different labels. This is why the original AST cannot be restored easily: in Figure 4 it is not possible to determine whether the tokens of the clause come before, after or in between the expressions. To make it possible, more information about the structure of the contracted nodes is needed.

The lack of order between label groups is the result of using a database for storage, which is required for fast queries. However, it is expected to be a good trade-off, since the exact AST order of the nodes is seldom needed (most importantly, when reprinting the contents of the graph into a file), while it provides queries in linear time of their length. The order of the links with the same label, which is important during queries, is retained.

3 Construction of new AST subtrees

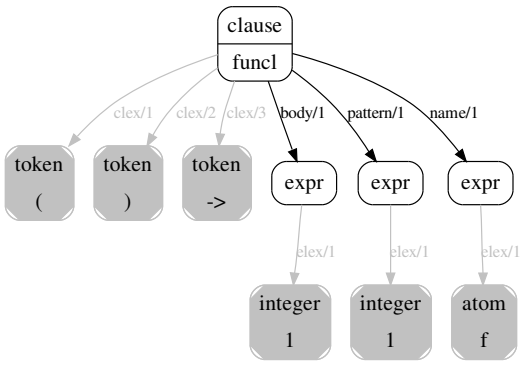
One possible solution for constructing AST subtrees would be to use the parser itself by providing the source code of the desired subtree – effectively, its front. This approach would require the user to manually fill in all the punctuation, and would require separate grammars for each nonterminal to be generated.

In this section, a different method is presented that makes constructing syntactically correct AST subtrees comfortable for the user. In section 3.1, structures are defined that describe the expected structure of a node (the node skeleton). The method itself is described in section 3.2.

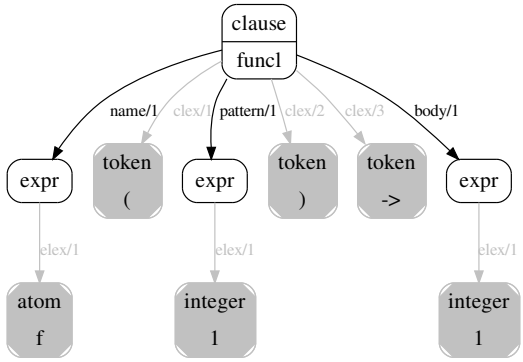
3.1 Node structure skeletons

The grammar description chosen is close to a BNF description. In it, the grammar rules are grouped by what contraction group their head belongs to. Rules, of course, may have more alternatives. The right hand sides of rules consist of a sequence of the following:

- **tokens** that contain the token node label,



(a) Part of an automatically printed contracted AST. The order of the edges between groups in unknown.



(b) The nodes rearranged in the right order. The order within the groups is retained. The tokens read: `f(1) -> 1`.

Figure 4:

- **symbols** that contain the child symbol's nonterminal and the edge label,
- **optional constructs**, sequences that either appear or not in a concrete instance and
- **repeat constructs** that contain a symbol and a token; its instances are

several (at least one) symbols with tokens intercalated.

We will such sequences node structure skeletons. Since optionals and repeats may contain one another, we shall refer to the number of contained nestings as the depth of the construct.

As an example that contains both constructs described above, let us examine the structure of lists. The structure of lists is described as follows. Lists start with an opening bracket token and end with a closing bracket token. Between them is an optional construct. The optional part consists of a repeat construct. The repeat construct uses comma tokens to separate symbols that are linked using "sub" edges from the parent node. The portion of the actual Erlang code that shows the above structure is shown in figure 5 in order to have a more concise overview.

```
[{token,"op_bracket"},
 {optional,[{repeat,"comma","sub"}]},
 {token,"cl_bracket"}];
```

Figure 5: The structure of lists as an Erlang structure used in the actual implementation. Slightly abridged.

Lists can be empty lists, or lists containing expression symbols separated by comment tokens. In the first case, the optional part is not present. In the second case, the optional is present. If there is one element in the repeat construct, there is exactly one symbol element present which denotes the expression.

$$\underbrace{[}_{\text{token}}\ \underbrace{\ }_{\text{empty optional}}\ \underbrace{]}_{\text{token}}$$

$$\underbrace{[}_{\text{token}}\ \underbrace{1}_{\text{repeat in optional}}\ \underbrace{]}_{\text{token}}$$

$$\underbrace{[}_{\text{token}}\ \underbrace{1, 2, 3, 4, 5, 6, 7, 8 + 9, f()}_{\text{repeat in optional}}\ \underbrace{]}_{\text{token}}$$

3.2 Automated subtree construction

From the XML grammar description, node structure skeletons are automatically generated for each node type.

The user has to supply two pieces of information for the node creation algorithm. One is the contents of the newly created node, which also contains its type. By supplying the type of the node, the relevant node structure skeleton can be determined. The other piece of information to be supplied is a description of the desired actual content of the new node. This parallels the structure of the skeleton in the following way.

1. Almost all tokens are automatically created; these tokens are not included in the description. Information about tokens that cannot be automatically created, e.g. names of functions, have to be present.

2. Symbol, optional and repeat descriptions are at corresponding positions with the skeleton.

3. Symbol descriptions contain the actual node to be incorporated as a child. As stated before, they are either created before or moved from a different position in the tree.

4. Repeat descriptions contain the actual symbol nodes. The tokens of repeat constructs are always autocreated, therefore they need not be specified in the description.

5. Optional descriptions are either `no_optional`, or they contain sequential content that parallels that of the optional in the skeleton.

The algorithm used to construct a contracted node processes the sequence in the skeleton along with the one in the description.

1. If the next construct is an autocreated token, it does not appear in the description, because it can be automatically created. The created token link from the parent to the depends only on the contracted type of the parent node. Note that all other constructs have to be present both in the skeleton and the description.

2. If the next construct is a (not autocreated) token or a symbol, the corresponding description item contains the node itself to be linked from the parent.

3. If the next construct is a repeat, the token nodes are autocreated and linked as above, and the symbols to be linked are listed in the repeat description. Since the representation is contracted, the insertions of the symbols and tokens below the parent do not have to be intercalated, as their connecting edges are in different label groups.

4. If the next construct is an optional, but the description contains `no_optional`, it is skipped.

5. If the next construct is an optional, and the description has actual content, the contentsof the optional skeleton and description are processed.

Subtrees can be created by repeated use of the above algorithm. When constructing a new node, previously created nodes can be used as well as nodes that were already present in the graph. Practically, for each common subtree type, a function has to be created in order to facilitate the use of the algorithm. The function itself invokes the algorithm with the contents of the created node and the appropriate parameters.

4 Related work

The design of the representation was shaped through years of experimentation and experience with refactoring functional programs. The first refactoring tools produced at ELTE [5, 8] used standard ASTs for representing the syntax. It became evident that such a representation is not convenient enough for refactoring purposes, and a new design was needed. The resulting design [4] already used the contracted graph described in section 2 as representation of the syntax tree.

The Java language tools srcML [9], JavaML [3] and JaML [1] use XML to model Java source code. XML documents can be equipped with schema information against which they can be checked. If the schema is formulated in XML itself, subtree construction algorithms similar to the one presented in this paper can be devised.

5 Acknowledgements

The refactoring group at ELTE has helped the author conceive, shape, test and improve the algorithm described in the paper.

This work was supported by GVOP-3.2.2-2004-07-0005/3.0 ELTE IKKK and Ericsson Hungary.

References

[1] G. Fischer, J. Lusiardi, and J. Wolff v. Gudenberg, Abstract syntax trees and their role in model driven software development. In *ICSEA online proceedings*. IEEE, 2007.

[2] J. Barklund and R. Virding, Erlang Reference Manual, 1999, Available from http://www.erlang.org/download/erl_spec47.ps.gz.

[3] Greg J. Badros, Javaml: a markup language for java source code. In *Proceedings of the 9th international World Wide Web conference on Computer networks: the international journal of computer and telecommunications networking*, pages 159–177. North-Holland Publishing Co. Amsterdam, The Netherlands, The Netherlands, 2000.

[4] Róbert Kitlei, László Lövei, Tamás Nagy, Zoltán Horváth, Tamás Kozsik, Preprocessor and whitespace-aware toolset for Erlang source code manipulation. Abstract submitted to the *20th International Symposium on the Implementation and Application of Functional Languages*, Hatfield UK.

[5] R. Szabó-Nacsa, P. Divinszky, and Z. Horváth, Prototype environment for refactoring Clean programs. In *The Fourth Conference of PhD Students in Computer Science (CSCS 2004)*, Szeged, Hungary, July 1–4, 2004.

[6] H. Li, S. Thompson, L. Lövei, Z. Horváth, T. Kozsik, A. Víg, and T. Nagy, Refactoring Erlang Programs. In *Proceedings of the 12th International Erlang/OTP User Conference*, November 2006.

[7] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts, *Refactoring: Improving the Design of Existing Code.* Addison-Wesley, 1999.

[8] L. Lövei, Z. Horváth, T. Kozsik, R. Király, A. Víg, and T. Nagy, Refactoring in Erlang, a Dynamic Functional Language. In *Proceedings of the 1st Workshop on Refactoring Tools*, Berlin, Germany, July 2007, pp. 45-46.

[9] Jonathan I. Maletic, Michael L. Collard, and Adrian Marcus, Source code files as structured documents. In *Proceedings of 10th IEEE International Workshop on Program Comprehension (IWPC'02)*, pp. 289–292. IEEE Computer Society Washington, DC, USA, 2002.

Received: October 1, 2008