# Parsing Formal Languages using Natural Language Parsing Techniques

**Jens Nilsson**[*] **Welf Löwe**[*] **Johan Hall**[†*] **Joakim Nivre**[†*]
[*]Växjö University, School of Mathematics and Systems Engineering, Sweden
[†]Uppsala University, Department of Linguistics and Philology, Sweden
{jens.nilsson|welf.lowe|johan.hall|joakim.nivre}@vxu.se

## Abstract

Program analysis tools used in software maintenance must be robust and ought to be accurate. Many data-driven parsing approaches developed for natural languages are robust and have quite high accuracy when applied to parsing of software. We show this for the programming languages Java, C/C++, and Python. Further studies indicate that post-processing can almost completely remove the remaining errors. Finally, the training data for instantiating the generic data-driven parser can be generated automatically for formal languages, as opposed to the manually development of treebanks for natural languages. Hence, our approach could improve the robustness of software maintenance tools, probably without showing a significant negative effect on their accuracy.

## 1 Introduction

Software engineering, especially software maintenance, is supported by numerous program analysis tools. Maintenance tasks include program comprehension (understanding unknown code for fixing bugs or further development), quality assessment (judging code, e.g., in code reviews), and reverse-engineering (reifying the design documents for given source code). To extract information from the programs, the tools first parse the program code and produce an abstract syntax tree (AST) for further analysis and abstraction (Strein et al., 2007). As long as the program conforms to the syntax of a programming language, classical parsing techniques known from the field of compiler construction may be applied. This, however, cannot be assumed in general, as the programs to analyze can be incomplete, erroneous, or conform to a (yet unknown) dialect or version of the language. Despite error stabilization, classical parsers then lose a lot of information or simply break down. This is unsatisfactory for tools supporting maintenance. Therefore, quite some effort has gone into the development of robust parsers of programs for these tools (cf. our related work section 5). This effort, however, has to be repeated for every programming language.

The development of *robust* parsers is of special interest for languages like C/C++ due to their numerous dialects in use (Anderson, 2008). Also, tools for languages frequently coming in new versions, like Java, benefit from robust parsing. Finally, there are languages like HTML where existing browsers are forgiving if documents do not adhere to the formal standard with the consequence that there exist many formally erroneous documents. In such cases, robust parsing is even a prerequisite for tool-supported maintenance.

The *accuracy* of parsing is a secondary goal in the context of software maintenance. Tasks like program comprehension, quality assessment, and reverse-engineering are fuzzy by their nature. There is no well-defined notion of correctness—rather an empirical answer to the question: Did it help the software engineers in fulfilling their tasks? Moreover, the information provided to the engineers abstracts anyway from the concrete program syntax and semantics, i.e., inaccuracies in the input may disappear in the output. Finally, program analyses are often heuristics themselves, approximating computationally hard problems like pattern matching and optimal clustering.

The natural language processing (NLP) community has for many years developed parsing technology that is both completely robust and highly accurate. The present approach applies this technology to programming languages. It is robust in the sense that, for each program, the parser always gives a meaningful model even for slightly incorrect and incomplete programs. The approach is,



*Proceedings of the 11th International Conference on Parsing Technologies (IWPT)*, pages 49–60,
Paris, October 2009. 

however, not accurate to 100%, i.e., even correct programs may lead to slightly incorrect models. As we will show, it is quite accurate when applied to programming languages.

The data-driven dependency parsing approach applied here only needs correct examples of the source and the expected analysis model. Then it automatically trains and adapts a generic parser. As we will show, training data for adapting to a new programming language can even be generated automatically. Hence, the effort for creating a parser for a new programming language is quite small.

The basic idea – applying natural language parsing to programming languages – has been presented to the program maintenance community before (Nilsson et al., 2009). This paper contributes with experimental results on

1. data-driven dependency parsing of the programming languages C/C++, Java, and Python,
2. transformations between dependency structure and phrase structure adapted to programming languages,
3. generic parser model selection and its effect on parsing accuracy.

Section 2 gives an introduction to the parsing technology applied here. In section 3, the preparation of the training examples necessary is described, while section 4 presents the experimental results. Section 5 discusses related work in information extraction for software maintenance. We end with conclusions and future work in section 6.

## 2 NLP Background

Dependency structure is one way of representing the syntax of natural languages. Dependency trees form labeled, directed and rooted trees, as shown in figure 1. One essential difference compared to context-free grammar is the absence of nonterminals. Another difference is that the syntactic structure is composed of lexical tokens (also called terminals or words) linked by binary and directed relations called *dependencies*. Each token in the figure is labeled with a part-of-speech, shown at the bottom of the figure. Each dependency relation is also labeled.

The parsing algorithm used in the experiments of section 4, known as the Nivre's arc-eager algorithm (Nivre, 2003), can produce such dependency trees. It bears a resemblance to the shift-reduce parser for context-free grammars, with the most apparent difference being that terminals (not nonterminals) are pushed onto the stack. Parser configurations are represented by a stack, a list of (remaining) input tokens, and the (current) set of arcs for the dependency tree. Similar to the shift-reduce parser, the construction of syntactic structure is created by a sequence of transitions. The parser starts with an empty stack and terminates when the input queue is empty, parsing input from left to right. It has four transitions (**Left-Arc**, **Right-Arc**, **Reduce** and **Shift**), manipulating these data structures. The algorithm has a linear time complexity as it is guaranteed to terminate after at most $2n$ transitions, given that the length of the input sentence is $n$.

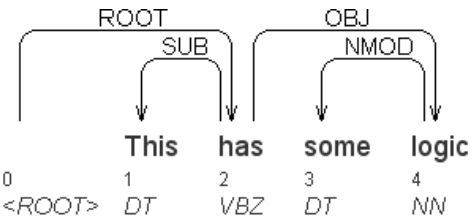


Figure 1: Sentence with a dependency tree.

In contrast to a parser guided by a grammar (e.g., ordinary shift-reduce parsing for context-free grammars), this parser is guided by a classifier induced from empirical data using machine learning (Nivre et al., 2004). Hence, the parser requires training data containing dependency trees. In other words, the parser has a training phase where the training data is used by the training module in order to learn the correct sequence of transitions. The training data can contain dependency trees for sentences of any language irrespectively of whether the language is a natural or formal one.

The training module produces the correct transition sequences using the dependency trees of the training data. These correct parser configurations and transition sequences are then provided as training data to a classifier, which predicts the correct transitions (including a dependency label for **Left-Arc**, **Right-Arc**) given parser configurations. A parser configuration contains a vast amount of information located in the data-structures. It is therefore necessary to abstract it into a set of features. Possible features are word forms and parts-

of-speech of tokens on the stack and in the list of input tokens, and dependency labels of dependency arcs created so far.

The parser produces exactly one syntactic analysis for every input, even if the input does not conform to a grammar. The price we have to pay for this robustness is that any classifier is bound to commit errors even if the input is acceptable according to a grammar.

## 3 General Approach

In section 2, we presented a parsing algorithm for producing dependency trees for natural languages. Here we will show how it can be used to produce syntactic structures for programming languages. Since the framework requires training data forming correct dependency trees, we need an approach for converting source code to dependency trees.

The general approach can be divided into two phases, training and production. In order to be able to perform both these phases in this study, we need to adapt natural language parsing to the needs of information extraction from programming language code, i.e., we need to automatically produce training data. Therefore, we apply:

(a) **Source Code ⇒ Syntax Tree**: the classical approach for generating syntax trees for correct and complete source code of a programming language.

(b) **Syntax Tree ⇒ Dependency Tree**: an approach for encoding the syntax trees as dependency trees adapted to programming languages.

(c) **Dependency Tree ⇒ Syntax Tree**: an approach to convert the dependency trees back to syntax trees.

These approaches have been accomplished as presented below. In the training phase, we need to train and adapt the generic parsing approach to a specific programming language. Therefore:

(1) **Generate training data** automatically by producing syntax trees and then dependency trees for correct programs using approaches (a) and (b).

(2) **Train** the generic parser with the training data.

This automated training phase needs to be done for every new programming language we adapt to. Finally, in the production phase, we extract the information from (not necessarily correct and complete) programs:

(3) **Parse** the new source code into dependency trees.

(4) **Convert** the dependency trees into syntax trees using approach (c).

This automated production phase needs to be executed for every project we analyze.

Steps (2) and (3) have already been discussed in section 2 for parsing natural languages. They can be generalized to parsing programming languages as described in section 3.1. Both the training phase and the production phase are complete, once the steps (a)–(c) have been accomplished. We present them in sections 3.2, 3.3, and 3.4, respectively.

### 3.1 Adapting the Input

As mentioned, the parsing algorithm described in section 2 has been developed for natural languages, which makes it necessary to resolve a number of issues that arise when the parser is adapted for source code as input. First, the parsing algorithm takes a sequence of words as input, and for simplicity, we map the tokens in a programming language to words.

One slightly more problematic issue is how to define a "sentence" in source code. A natural language text syntactically decomposes into a sequence of sentences in a relatively natural way. But is there also a natural way of splitting source code into sentences? The most apparent approach may be to define a sentence as a compilation unit, that is, a file of source code. This can however result in practical problems since a sentence in a natural language text is usually on average between 15–25 words long, partially depending on the author and the type of text. The sequence of tokens in a source file may on the other hand be much longer. Time complexity is usually in practice of less importance when the average sentence length is as low as in natural languages, but that is hardly the case when there can be several thousands tokens in a sentence to parse.

Other approaches could for instance be to let one method be a sentence. However, then we need to deal with other types of source code constructions explicitly. We have in this study for simplicity let one compilation unit be one sentence. This is possible in practice due to the linear time

complexity of the parsing algorithm of section 2, a quite unusual property compared to other NLP parsers guided by machine learning with state-of-the-art accuracy.

### 3.2 Source Code ⇒ Syntax Tree

In order to produce training data for the parser for a programming language, an analyzer that constructs syntax trees for correct and complete source code of the programming language is needed. We are in this study focusing on Java, Python and C/C++, and consequently need one such analyzer for each language. For example, figure 2 shows the concrete syntax tree of the following fragments of Java:

Example (1):

```
public String getName() {
   return name;
}
```

Example (2):

```
while (count > 0) {
   stack[--count]=null;
}
```

We also map the output of the lexical analyzer to the parts-of-speech for the words (e.g., `Identifier` for `String` and `getName`). All source code comments and indentation information (except for Python where the indentation conveys hierarchical information) have been excluded from the syntax trees. All string and character literals have also been mapped to "string" and "char", respectively. This does not entail that the approach is lossy, since all this information can be retained in a post-processing step, if necessary. As pointed out by, for instance, Collard et al. (2003), comments and indentation may among other things be of interest when trying to understand source code.

### 3.3 Syntax Tree ⇒ Dependency Tree

Here we will discuss the conversion of syntax trees into dependency trees. We use a method that has been successfully applied for natural languages for converting syntax trees into a *convertible* dependency tree that makes it possible to perform the inverse conversion, meaning that information about the syntax tree is saved in complex arc labels (Hall and Nivre, 2008). We also present results in section 4 using the dependency trees that cannot be used for the inverse conversion, which we call *non-convertible* dependency trees.

The conversion is performed in a two-step approach. First, the algorithm traverses the syntax tree from the root and identifies the head-child and the terminal head for all nonterminals in a recursive depth-first search. To identify the head-child for each nonterminal, the algorithm uses heuristics called *head-finding rules*, inspired by, for instance, Magerman (1995). Three head-finding strategies have been investigated. For each nonterminal:

1. FREQ: Let the element with the most frequently occurring name be the head, but exclude the token ';' as a potential head. If two tokens have the same frequency, let the leftmost occurring element be the head.
2. LEFT: let the leftmost terminal in the entire subtree of the nonterminal be the head of all other elements.
3. RIGHT: let the rightmost terminal in the entire subtree of the nonterminal be the head of all other elements.

The dependency trees in figures 3 and 4 use LEFT and FREQ. LEFT and RIGHT induce that all arcs are pointing to the right and left, respectively. The head-finding rules for FREQ are automatically created by counting the children's names for each distinct non-terminal name in the syntax trees of the training data. The priority list is then compiled by ordering the elements by descending frequency for each distinct non-terminal name. For instance, given that the syntax trees are grammatically correct, every non-terminal `While` will contain the tokens `(`, `)` and `while`. These tokens have thus the highest priority, and `while` therefore becomes the head in the lower dependency tree of figure 4. This is the same as choosing the left-most mandatory element for each left-hand side in the grammar. An interesting observation is that binary operators and the copy assignment operator become the heads of their operands for FREQ, which is the case for < and = in figure 4. Note also that the element names of terminals act as part-of-speech tags, e.g., the part-of-speech for `String` is `Identifier`.

In the second step, a dependency tree is created according to the identified terminal heads. The arcs in the convertible dependency tree are labeled with complex arc labels, where each complex arc label consists of two sublabels:

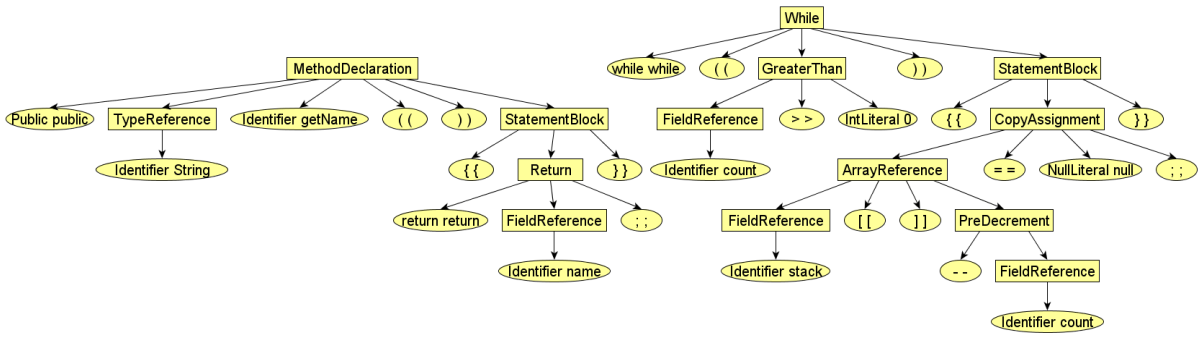


Figure 2: Syntax trees for examples (1) and (2).

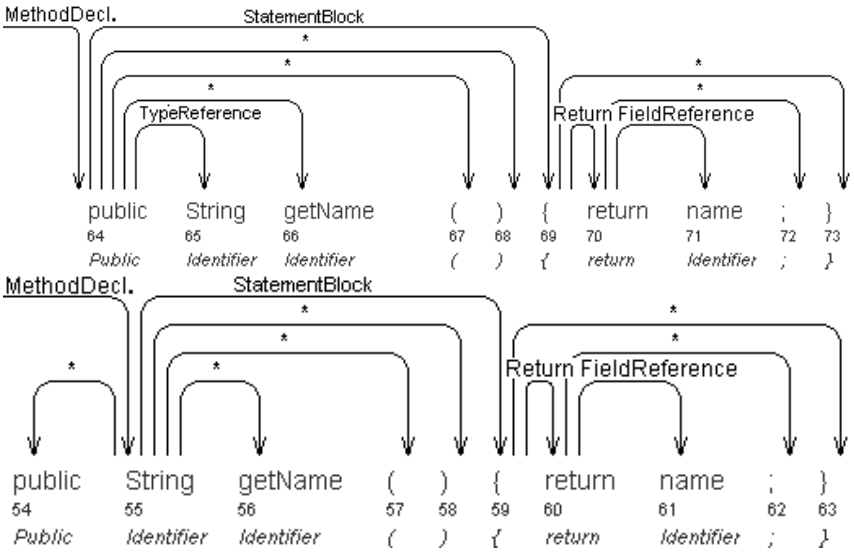


Figure 3: Non-convertible dependency trees for example (1) using LEFT (upper) and FREQ (lower).

1. Encode the dependent spine, i.e., the sequence of nonterminal labels from the dependent terminal to the highest nonterminal where the dependent terminal is the terminal head; "|" separates the nonterminal labels,

2. Encode the attachment point in the head spine, a non-negative integer value $a$, which means that the dependent spine is attached $a$ steps up in the head spine.

By encoding the arc labels with these two sublabels, it is possible to perform the inverse conversion, (see subsection 3.4).

The non-convertible dependency labels allow us to reduce the complexity of the arc labels, making the learning problem simpler due to fewer distinct arc labels. This may result in a higher accuracy during parsing and can be used as input for further processing directly without taking the detour back to syntax trees. This can be motivated by the fact that all information in the syntax trees is usually not needed anyway in many reverse engineering tasks, but labels indicating method calls and declarations – the most important information for most program comprehension tasks – are preserved. This is exemplified by the fact that both dependency structures in figure 3 contain the label `MethodsDecl`.. We thus believe that all the necessary information is also captured in this less informative dependency tree. Each dependency label is the highest nonterminal name of the spine, that is, the single nonterminal name that is closest to its head. The non-convertible dependency label also excludes the attachment point value, making the learning problem even simpler. Figures 3 and 4 show the non-convertible dependency labels of the syntax trees (or phrase structure trees) in the same figures, where each label contains just a single nonterminal name of the original syntax trees.

### 3.4 Dependency Tree ⇒ Syntax Tree

The inverse conversion is a bottom-up and top-down process on the convertible dependency tree

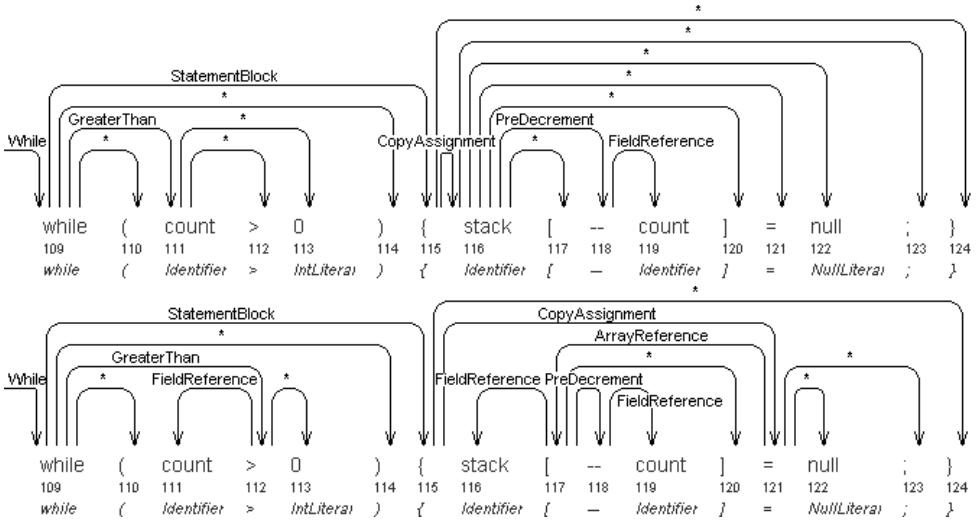


Figure 4: Non-convertible dependency trees for example (2) using LEFT (upper) and FREQ (lower).

(must contain complex arc labels). First, the algorithm visits every terminal in the convertible dependency tree and restores the spines of nonterminals with labels for each terminal using the information in the first sublabel of the incoming arc. Thus, the bottom-up process results in a spine of zero or more arcs from each terminal to the highest nonterminal of which the terminal is the terminal head. Secondly, the spines are weaved together according to the arcs of the dependency tree. This is achieved by traversing the dependency tree recursively from the root using a pre-order depth-first search, where the dependent spine is attached to its head spine or to the root of the syntax tree. The attachment point $a$, given by the second sublabel, specifies the number of nonterminals between the terminal head and the attachment nonterminal.

## 4 Experiments

We will in this section present parsing experiments and evaluate the accuracy of the syntax trees produced by the parser. As mentioned in section 2, the parsing algorithm is robust in the sense that it always produces a syntactic analysis no matter the input, but it can commit errors even for correct input. This section investigates the accuracy for correct input, when varying feature set, head-finding rules and language. We begin with the experimental setup.

### 4.1 Experimental Setup

The open-source software MaltParser (maltparser.org) (Nivre et al., 2006) is used in the experiments. It contains an implementation of the parsing algorithm, as well as an implementation of the conversion strategy from syntax trees to dependency trees and back, presented in subsections 3.3 and 3.4. It comes with the machine learner LIBSVM (Chang and Lin, 2001), producing the most accurate results for parsing natural languages compared to other evaluated machine learners (Hall et al., 2006). LIBSVM requires training data. The source files of the following projects have been converted into dependency trees:

- For Java: Recoder 0.83 (Gutzmann et al., 2007), using all source files in the directory "src" (having 400 source files with 92k LOC and 335k tokens).
- For C/C++: Elsa 2005.08.22b (McPeak, 2005), where 1389 source files were used, including the 978 C/C++ benchmark files in the distribution (thus comprising 1389 source files with 265k LOC and 691k tokens).
- For Python: Natural Language Toolkit 0.9.5 (Bird et al., 2008), where all source files in the directory "nltk" were used (having 160 source files with 65k LOC and 280k tokens).

To construct the syntax tree for the source code file of Recoder, we have used Recoder. It creates an abstract syntax tree for a source file, but we are currently interested in the concrete syntax tree with all the original tokens. In this first conversion step, the tokens of the syntax trees are thus retained. For example, the syntax trees in figure 2 are generated by Recoder.

The same strategy was adopted for Elsa with the difference that CDT 4.0.3, a plug-in to the Eclipse IDE to produce syntax trees for source code of C/C++, was used for producing the abstract syntax trees.[1] It produces abstract syntax trees just like Recoder, so the concrete syntax trees have also been created by retaining the tokens.

The Python 2.5 interpreter is actually shipped with an analyzer that produces concrete syntax trees (using the Python imports `from _ast import PyCF_ONLY_AST` and `import parser`), which we have utilized for the Python project above. Hence, no additional processing is needed in order prepare the concrete syntax trees as training data.

For the experiments, the source files have been divided into a training set $T$ and a development test set $D$, where the former comprises 80% of the dependency trees and the latter 10%. The remaining 10% ($E$) has been left untouched for later use. The source files have been ordered alphabetically by the file names including the path. The dependency trees have then been distributed into the data sets in a pseudo-randomized way. Every tenth dependency tree starting at index 9 (i.e. dependency trees 9, 19, 29, ...) will belong to $D$, and every tenth dependency trees starting at index 0 to $E$. The remaining trees constitute the training set $T$.

### 4.2 Metrics

The standard evaluation metric for parse trees for natural languages based on context-free grammar is F-score, the harmonic mean of precision and recall. F-score compares constituents – defined by triples $\langle i, j, XP \rangle$ spanning between terminals $i$ and $j$ – derived from the test data with those derived from the parser. A constituent in the parser output matches a constituent in the test data when they span over the same terminals in the input string. Recall is the ratio of matched constituents over all constituents in the test data. Precision is the ratio of matched constituents over all constituents found by the parser. F-score comes in two versions, one unlabeled ($F_U$) and one labeled ($F_L$), where each correct constituent in the latter also must have the correct nonterminal name (i.e., $XP$). The metric is implemented in Evalb (Collins and Sekine, 2008).

[1] It is worth noting that CDT failed to produce syntax trees for 2.2% of these source files, which were consequently excluded from the experiments. This again indicates the difficult of parsing C/C++ due to its different dialects.

| | $F_L$ | | | $F_U$ | | |
|---|---|---|---|---|---|---|
| | FR | LE | RI | FR | LE | RI |
| **UL** | 82.1 | 93.5 | 74.6 | 92.3 | 97.9 | 90.6 |
| **L** | 89.7 | 97.7 | 80.8 | 95.8 | 99.3 | 92.1 |

Table 1: F-score for various parser models and head-finding rules for Java, where FR = FREQ, LE = LEFT and RI = RIGHT.

The standard evaluation metric measuring accuracy for dependency parsing for natural language is, on the other hand, labeled ($AS_L$) and unlabeled ($AS_U$) attachment score. $AS_U$ is the ratio of tokens attached to its correct head. $AS_L$ is the same as $AS_U$ with the additional requirement that the dependency label should be correct as well.

### 4.3 Results

This section presents the parsing results. The first experiment was conducted for Java, using the inverse transformation back to syntax trees. Two feature models are evaluated, one unlexicalized feature sets (**UL**) containing 13 parts-of-speech and 4 dependency label features, and one lexicalized feature sets (**L**) containing all these 17 features and 13 additional word form features, developed by manual feature optimization. Table 1 compares these two feature sets, as well as the different head-finding rules discussed previously.

The figures give a clear answer to the question whether lexical information is beneficial or not. Every figure in the row **L** is higher than its corresponding figure in the row **UL**. This means that names of variables, methods, classes, etc., actually contain valuable information for the classifier. This is in contrast to ordinary syntactic parsing using a grammar of programming languages where all names are mapped to the same value (e.g. Identifier), and, e.g., integer constants to IntLiteral, before the parse. One potential contributing factor of the difference is the naming conventions that programmers normally follow. For example, naming classes, class attributes and local variables, etc. using typical methods names, such as `equals` in Java, is usually avoided by programmers.

It is just as clear that the choice of head-finding strategy is very important. For both $F_L$ and $F_U$, the best choice is with a wide margin LEFT, followed by FREQ. RIGHT is consequently the least accurate one. A higher amount of arcs pointing to the right seems to be beneficial for the strategy of

| | $AS_L$ | | | $AS_U$ | | |
|---|---|---|---|---|---|---|
| | FR | LE | RI | FR | LE | RI |
| *CO* | 87.6 | 96.6 | 86.6 | 90.9 | 98.2 | 90.7 |
| *NC* | 91.0 | 99.1 | 89.5 | 92.1 | 99.7 | 90.7 |

Table 2: Attachment score for Java and the lexical feature set, where *CO* = convertible and *NC* = non-convertible dependency trees.

| | Python | | C/C++ | |
|---|---|---|---|---|
| | $F_L$ | $F_U$ | $F_L$ | $F_U$ |
| **UL** | 91.5 | 92.1 | 95.6 | 96.4 |
| **L** | 99.1 | 99.2 | 96.5 | 96.9 |

Table 3: F-score for various parser models and head-finding rules LEFT for Python and C/C++.

parsing from left to right.

Table 1 can be compared to the accuracy on the parser output before conversion from dependency trees to syntax trees. This is shown in the first row (*CO*) of table 2, where all information in the complex dependency label is concatenated and placed in the dependency label. The relationships between the head-finding strategies remain the same, but it is worth noting that the accuracies for FREQ and RIGHT are closer to each other, entailing a more difficult conversion to syntax trees for the latter. The first row can also be compared to the second row (*NC*) in the same table, showing the accuracies when training and parsing with non-convertible dependency trees. One observation is that each figure in *NC* is higher than its corresponding figure in *CO* (even $AS_U$ for RIGHT with more decimals), probably attributed to the lower burden on the parser. Both $AS_U$ and $AS_L$ are above 99% for the non-convertible dependency trees using LEFT.

We can see that choosing an appropriate representation of syntactic structure to be used during parsing is just as important for programming languages as for natural languages, when using data-driven natural language parsers (Bikel, 2004).

The parser output in table 1 can more easily be used as input to existing program comprehension tools, normally requiring abstract syntax trees. However, the highly accurate output for LEFT using non-convertible dependency trees could be worth using instead, but it requires some additional processing.

In order to investigate the language independence of our approach, table 3 contains the corresponding figures as in table 1 for Python and C/C++, restricted to LEFT, which is the best head-finding strategy for these languages as well. Again, each lexicalized feature set (**L**) outperforms its corresponding unlexicalized feature set (**UL**). Python has higher $F_L$ and virtually the same $F_U$ as Java, whereas C/C++ has the lowest accuracies for **L**. However, the **UL** figures are not far behind the **L** figures for C/C++, and C/C++ has in fact higher $F_L$ for **UL** compared to Java and Python. These results can maybe be explained by the fact that C/C++ has less verbose syntax than both Java and Python, making the lexical features less informative.

The $F_L$ figures for Java, Python and C/C++ using LEFT can also be compared to the corresponding figures in Nilsson et al. (2009). They use the same data sets but a slightly different head-finding strategy. Instead of selecting the leftmost element (terminal or non-terminal) as in LEFT, they always select the leftmost terminal, resulting in $F_L$=99.5 for Java, $F_L$=98.3 for Python and $F_L$=96.5 for C/C++. That is, our results are slightly lower for Java, higher for Python, and slightly higher for C/C++. The same holds for $F_U$ as well. That is, having only arcs pointing to the right results in high accuracy for all languages (which is the case for `Left` described in section 3), but small deviations from this head-finding strategy can in fact be beneficial for some languages.

We are not aware of any similar studies for programming languages[2] so we compare the results to natural language parsing. First, the figures in table 2 for dependency structure are better than figures reported for natural languages. Some natural languages are easier to parse than others, and the parsing results of the CoNLL shared task 2007 (Nivre et al., 2007) for dependency structure indicate that English and Catalan are relatively easy, with $AS_L$ around 88-89% and $AS_U$ around 90-94% for the best dependency parsers.

Secondly, compared to parsing German with phrase structure with the same approach as here, with $F_U$ = 81.4 and $F_L$ = 78.7%, and Swedish, with $F_U$ = 76.8 and $F_L$ = 74.0 (Hall and Nivre,

[2]A comparative experiment using another data-driven NLP parser for context-free grammar could be of theoretical interest. However, fast parsing time is important in program comprehension tasks, and data-driven NLP parsers for context-free grammar have worse than a linear time complexity. As, e.g., the Java project has 838 tokens per source file, linear time complexity is a prerequisite in practice.

| | Correct Label | Parsing Label |
|---|---|---|
| 66 | FieldReference | VariableReference |
| 25 | VariableReference | FieldReference |
| 12 | MethodDeclaration | LocalVariableDeclaration |
| 9 | Conditional | FieldReference |
| 5 | NotEquals | MethodReference |
| 4 | Plus | MethodReference |
| 4 | Positive | * |
| 4 | LessThan | FieldReference |
| 4 | GreaterOrEquals | FieldReference |
| 4 | Divide | FieldReference |
| 4 | Modulo | FieldReference |
| 4 | LessOrEquals | FieldReference |
| 3 | Equals | NotEquals |
| 3 | LessOrEquals | Equals |
| 3 | NotEquals | Equals |

Table 4: Confusion matrix for Java using non-convertible dependency trees with LEFT, ordered by descending frequency.

2008), the figures reported in tables 1 and 3 are also much better. It is however worth noting that natural languages are more complex and less regular compared to programming languages. Although it remains to be shown, we conjecture that these figures are sufficiently high for a large number of program comprehension tasks.

### 4.4 Error Analysis

This subsection will study the result for Java with non-convertible dependency trees (*NC*) and LEFT, in order to get a deeper insight into the types of errors that the parser commits. Specifically, the labeling mistakes caused by the parser are investigated here. This is done by producing a confusion matrix based on the dependency labels. That is, how often does a parser confuse label $X$ with label $Y$. This is shown in table 4 for the 15 most common errors.

The two most frequent errors show that the parser confuses *FieldReference* and *VariableReference*. A *FieldReference* refers to a class attribute whereas a *VariableReference* could refer to either an attribute or a local variable. The parser mixes a reference to a class attribute with a reference that could also be a local variable or vice versa. The error is understandable, since the parser obviously has no knowledge about where the variables are declared. This is an error that type and name analysis can easily resolve. On the use-occurrence of a name (reference), analysis looks up for both possible define-occurrences of the name (declaration), first a *LocalVariableDeclaration* and then a *FieldDeclaration*. It uses the one that is found first.

Another type of confusion involves declarations, where a *MethodDeclaration* is misinterpreted as a *LocalVariableDeclaration*. This type of error can be resolved by a simple post-processing step: a *LocalVariableDeclaration* followed by opening parenthesis (always recognized correctly) is a *MethodDeclaration*.

Errors that involve binary operators, e.g., *Conditional*, *NotEqual*, *Plus*, are at rank 4 and below in the list of the most frequent errors. They are most likely a result of the incremental left-to-right parsing strategy. The whole expression should be labeled in accordance with its binary operator (see `count > 0` in figure 4 for LEFT), but is incorrectly labeled as either *MethodReference*, *FieldReference* or some other operator instead. The references actually occur in the left-hand side subexpression of the binary operators. This means that subexpressions and bracketing were recognized correctly, but the type of the top expression node was mixed up. Extending the lookahead in the list of remaining input tokens, making it possible for the classifier in the parser to look at even more yet unparsed tokens, might be one possible solution. However, these errors are by and large relatively harmless anyway. Hence, no correction is in practice needed.

Figure 5 displays some typical mistakes for the example program fragment

```
return (fw.unitIndex == unitIndex &&
          fw.unitIndex.equals(unitList));
```

The parser mixes up a *ParenthesizedExpression* with a *Conditional*, a boolean *ParenthesizedExpression* only occurring in conditional statements and expressions. Then it incorrectly assigns the label *Equals* to the arc between the first left parenthesis and the first `fw` instead of the correct label *LogicalAnd*. It mixes up the type of the whole expression, an *Equals*- (i.e., ==) is taken for an *LogicalAnd*-expression (i.e., `&&`). Finally, the two *FieldReference*s are taken as more general *VariableReference*s, which is corrigible as discussed.

In addition to a possible error correction in a post-processing step, the parsing errors could disappear due to the abstraction of subsequent analyses as commonly used in software maintenance tools. For instance, without any error correction, the type reference graphs of our test program, the correct one and the one constructed using the not quite correct parsing results, are identical.

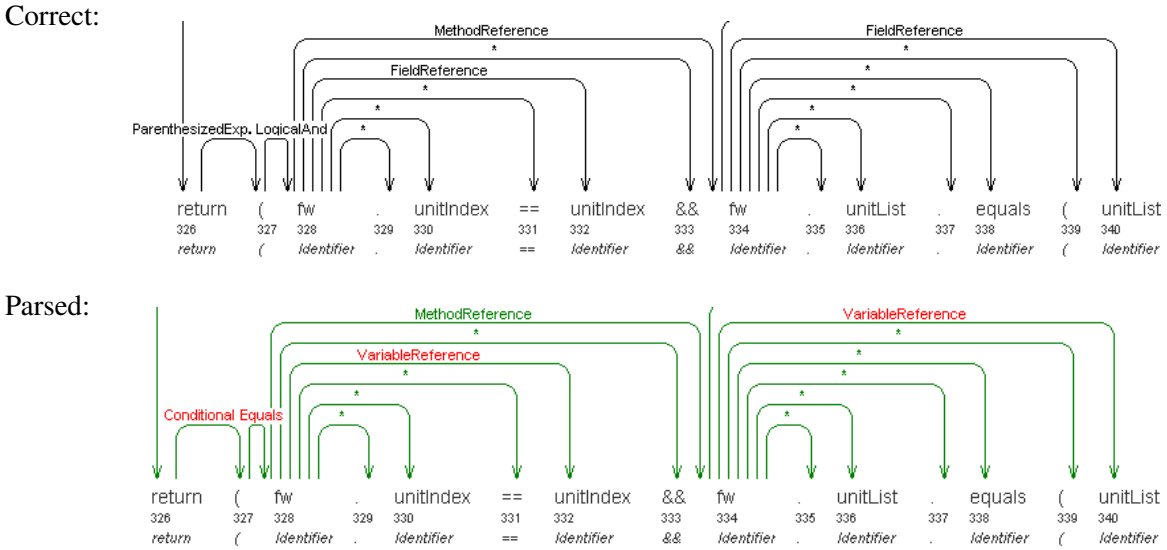


Figure 5: Typical errors for LEFT using by non-convertible dependency trees.

## 5 Related Work

Classical parsers for formal languages have been known for many years. They (conventionally) accept a context-free language defined by a context-free grammar. For each program, the parsers produce a phrase structure referred to as an abstract syntax tree (AST) which is also defined by a context-free language. Parsers including error stabilization and AST-constructors can be generated from context-free grammars for parsers (Kastens et al., 2007). A parser for a new language still requires the development of a complex specification. Moreover, error stabilization often throws away large parts of the source – it is robust but does not care about maximizing accuracy.

Breadth-First Parsing (Ophel, 1997) was designed to provide better error stabilization than traditional parsers and parser generators. It uses a two phase approach: the first phase identifies high-level entities – the second phase parses the structure with these entities as root nonterminals (axioms).

Fuzzy Parsing (Koppler, 1997) was designed to efficiently develop parsers by performing the analysis on selected parts of the source instead of the whole input. It is specified by a set of (sub)grammars each with their own axioms. The actual approach is then similar to Breadth-First Parsing: it scans for instances of the axioms and then parses according to the grammar. It makes parsing more robust in the sense that it ignores source fragments – including missing parts, errors and deviations therein – that subsequent analyses abstract from anyway. A prominent tool using the fuzzy parsing approach for information extraction in reverse-engineering tools is Sniff (Bischofberger, 1992) for analyzing C++ code.

Island grammars (Moonen, 2001) generalize on Fuzzy Parsing. Parsing is controlled by two grammar levels (island and sea) where the sea-level is used when no island-level production applies. The island-level corresponds to the sub-grammars of fuzzy parsing. Island grammars have been applied in reverse-engineering, specifically, to bank software (Moonen, 2002).

Syntactic approximation based on lexical analysis was developed with the same motivation as our work: when maintenance tools need syntactic information but the documents could not be parsed for some reason, hierarchies of regular expression analyses could be used to approximate the information with high accuracy (Murphy and Notkin, 1995; Cox and Clarke, 2003). Their information extraction approach is characterized as "lightweight" in the sense that it requires little specification effort.

A similar robust and light-weight approach for information extraction constructs XML formats (JavaML and srcML) from C/C++/Java programs first, before further processing with XML tools like Xpath (Badros, 2000; Collard et al., 2003). It combines lexical and context free analyses. Lexical pattern matching is also used in combination with context free parsing in order to extract facts from semi-structured specific comments and con-

figuration specifications in frameworks (Knodel and Pinzger, 2003).

TXL is a rule-based language defining information extraction and transformation rules for formal languages (Cordy et al., 1991). It makes it possible to incrementally extend the rule base and to adapt to language dialects and extensions. As the rules are context-sensitive, TXL goes beyond the lexical and context-free approaches discussed before.

The fundamental difference of our approach compared to lexical, context-free, and context-sensitive approaches (and combinations thereof) is that we use *automated* machine learning instead of *manual* specification for defining and adapting the information extraction.

General NLP techniques have been applied for extracting facts from general source code comments to support software maintenance (Etzkorn et al., 1999). Comments are extracted from source code using classical lexical analysis; additional information is extracted (and then added) with classical compiler front-end technology.

NLP has also been applied to other information extraction tasks in software maintenance to analyze unstructured or very large information sources, e.g., for analyzing requirement specifications (Sawyer et al., 2002), in clone detection (Marcus and Maletic, 2001; Grant and Cordy, 2009), and to connect program documentation to source code (Marcus and Maletic, 2003).

## 6 Conclusions and Future Work

In this paper, we applied natural language parsing techniques to programming languages. One advantage is that it offers *robustness*, since it always produces some output even if the input is incorrect or incomplete. Completely correct analysis can, however, not be guaranteed even for correct input. However, the experiments showed that accuracy is in fact close to 100%.

In contrast to robust information extractors used so far for formal languages, the approach presented here is rapidly adaptable to new languages. We *automatically generate* the language specific information extractor using machine learning and training of a generic parsing, instead of *explicitly specifying* the information extractor using grammar and transformation rules. Also the training data can be generated automatically. This could increase the development efficiency of parsers, since no language specification has to be provided, only examples.

Regarding accuracy, the experiments showed that selecting the syntactic base representation used by the parser internally has a major impact. Incorporating, for instance, class, method and variable names in the set of features of the parser improves the accuracy more than expected. The detailed error analysis showed that many errors committed by the parser are forgivable, as they are anyway abstracted in later processing phases. Other errors are easily corrigible. We can also see that the best results presented here are much higher than the best parsing results for natural languages.

Besides efficient information extractor development, efficient parsing itself is important. Applied to programs which can easily contain several million lines of code, a parser with more than linear time complexity is not acceptable. The data-driven parser utilized here has linear parsing time.

These results are only the first (promising) step towards natural language parsing leveraging information extraction for software maintenance. However, the only way to really evaluate the usefulness of the approach is to use its output as input to client analyses, e.g., software measurement and architecture recovery, which we plan to do in the future. Another direction for future work is to apply the approach to more dialects of C/C++, such as analyzing correct, incomplete, and erroneous programs for both standard C and its dialects.

## References


Paul Anderson. 2008. 90 % Perspiration: Engineering Static Analysis Techniques for Industrial Applications. In *Proceedings of the 8th IEEE International Working Conference on Source Code Analysis and Manipulation*, pages 3–12.

Greg J. Badros. 2000. 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.

Daniel M. Bikel. 2004. Intricacies of Collins' Parsing Model. *Computational Linguistics*, 30(4):479–511.

Steven Bird, Edward Loper, and Ewan Klein. 2008. Natural Language Toolkit (NLTK) 0.9.5. http://nltk.org/.

Walter R. Bischofberger. 1992. Sniff: A Pragmatic Approach to a C++ Programming Environment. In *USENIX C++ Conference*, pages 67–82.

Chih-Chung Chang and Chih-Jen Lin. 2001. LIBSVM: A Library for Support Vector Machines.

Michael L. Collard, Huzefa H. Kagdi, and Jonathan I. Maletic. 2003. An XML-Based Lightweight C++ Fact Extractor. In *11th IEEE International Workshop on Program Comprehension*, pages 134–143.

Michael Collins and Satoshi Sekine. 2008. Evalb. http://nlp.cs.nyu.edu/evalb/.

James R. Cordy, Charles D. Halpern-Hamu, and Eric Promislow. 1991. TXL: a Rapid Prototyping System for Programming Language Dialects. *Computer Languages*, 16(1):97–107.

Anthony Cox and Charles L. A. Clarke. 2003. Syntactic Approximation Using Iterative Lexical Analysis. In *Proceedings of the 11th IEEE International Workshop on Program Comprehension*, pages 154–163.

Letha H. Etzkorn, Lisa L. Bowen, and Carl G. Davis. 1999. An Approach to Program Understanding by Natural Language Understanding. *Natural Language Engineering*, 5(3):219–236.

Scott Grant and James R. Cordy. 2009. Vector Space Analysis of Software Clones. In *Proceedings of the IEEE 17th International Conference on Program Comprehension*, pages 233–237.

Tobias Gutzmann, Dirk Heuzeroth, and Mircea Trifu. 2007. Recoder 0.83. http://recoder.sourceforge.net/.

Johan Hall and Joakim Nivre. 2008. Parsing Discontinuous Phrase Structure with Grammatical Functions. In *Proceedings of GoTAL*, pages 169–180.

Johan Hall, Joakim Nivre, and Jens Nilsson. 2006. Discriminative Classifiers for Deterministic Dependency Parsing. In *Proceedings of COLING-ACL*, pages 316–323.

Uwe Kastens, Anthony M. Sloane, and William M. Waite. 2007. *Generating Software from Specifications*. Jones and Bartlett Publishers.

Jens Knodel and Martin Pinzger. 2003. Improving Fact Extraction of Framework-Based Software Systems. In *Proceedings of 10th Working Conference on Reverse Engineering*, pages 186–195.

Rainer Koppler. 1997. A Systematic Approach to Fuzzy Parsing. *Software - Practice and Experience*, 27(6):637–649.

David M. Magerman. 1995. Statistical Decision-tree Models for Parsing. In *Proceedings of ACL*, pages 276–283.

Andrian Marcus and Jonathan I. Maletic. 2001. Identification of High-Level Concept Clones in Source Code. In *Proceedings of the 16th IEEE international conference on Automated software engineering*, page 107.

Andrian Marcus and Jonathan I. Maletic. 2003. Recovering Documentation-to-Source-Code Traceability Links using Latent Semantic Indexing. In *Proceedings of the 25th International Conference on Software Engineering*, pages 125–135.

Scott McPeak. 2005. Elsa: The Elkhound-based C/C++ Parser. http://www.cs.berkeley.edu/∼smcpeak.

Leon Moonen. 2001. Generating Robust Parsers using Island Grammars. In *Proceedings of the 8th Working Conference on Reverse Engineering*, pages 13–22.

Leon Moonen. 2002. Lightweight Impact Analysis using Island Grammars. In *Proceedings of the 10th International Workshop on Program Comprehension*, pages 219–228.

Gail C. Murphy and David Notkin. 1995. Lightweight Source Model Extraction. *SIGSOFT Software Engineering Notes*, 20(4):116–127.

Jens Nilsson, Welf Löwe, Johan Hall, and Joakim Nivre. 2009. Natural Language Parsing for Fact Extraction from Source Code. In *Proceedings of 17th IEEE International Conference on Program Comprehension*, pages 223–227.

Joakim Nivre, Johan Hall, and Jens Nilsson. 2004. Memory-based Dependency Parsing. In *Proceedings of CoNLL*, pages 49–56.

Joakim Nivre, Johan Hall, and Jens Nilsson. 2006. MaltParser: A Data-Driven Parser-Generator for Dependency Parsing. In *Proceedings of LREC*, pages 2216–2219.

Joakim Nivre, Johan Hall, Sanda Kübler, Ryan McDonald, Jens Nilsson, Sebastian Riedel, and Deniz Yuret. 2007. The CoNLL 2007 Shared Task on Dependency Parsing. In *Proceedings of CoNLL/ACL*, pages 915–932.

Joakim Nivre. 2003. An Efficient Algorithm for Projective Dependency Parsing. In *Proceedings of IWPT*, pages 149–160.

John Ophel. 1997. Breadth-First Parsing. citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.50.3035.

Pete Sawyer, Paul Rayson, and Roger Garside. 2002. REVERE: Support for Requirements Synthesis from Documents. *Information Systems Frontiers*, 4(11):343–353.

Dennis Strein, Rüdiger Lincke, Jonas Lundberg, and Welf Löwe. 2007. An Extensible Meta-Model for Program Analysis. *IEEE Transactions on Software Engineering*, 33(9):592–607.