Parsing Formal Languages Using Natural Language Parsing Techniques

Jens Nilsson* Welf Löwe* Johan Hall*† Joakim Nivre†*

*Växjö University, Sweden
†Uppsala University, Sweden

{jens.nilsson|welf.lowe|johan.hall|joakim.nivre}@vxu.se
Introduction

Why on earth should we want to apply natural language parsing techniques to programming languages?

Our motivation comes from software maintenance tasks:

- program comprehension: understand unknown source code for fixing bugs or further development
- quality assessment: judge code, e.g., in source code reviews
- reverse-engineering: reify design documents for existing code

Software maintenance is supported by numerous existing tools for program analysis

First step (usually): create syntax trees for source code files
Introduction

- No problem to create syntax trees for source code if
  - system is written in a formal language $L$
  - source code parser for $L$ exists

- More problematic if system is written in $L' \approx L$, for example:
  - incomplete programs
  - syntactically erroneous programs
  - programs written in a different dialects

- Can we use NL parsing techniques in these circumstances?
Introduction

► Three requirements on the parsing of source code:
  (i) robust: syntax tree for incorrect or incomplete input
  (ii) accurate: correct analysis for input in \( L \)
  (iii) rapid development for new languages and dialects

► Rapid development is of special interest for languages like:
  ▶ C/C++ due to numerous dialects in use
  ▶ HTML due to the fuzzy language definition
  ▶ Java coming in new versions

► Our contributions:
  1. Methodology for (i) robust and (ii) highly accurate source code parsers, (iii) rapidly developed for new language variants
  2. Evaluation on C/C++, Java, Python
Other approaches

Methods from compiler construction:
- Methods for error stabilization exist, but they throw away large parts of the source code ⇒ information loss
- More sophisticated robust approaches to avoid information loss exist, e.g., Breadth-First Parsing (Ophel, 1997), Fuzzy Parsing (Koppler, 1997), Island Grammars (Moonen, 2001), but require a lot of manual language-specific specification labor

Other NLP techniques to support software maintenance
- Extracting info from source code comments, using classical lexical analysis (Etzkornet al., 1999)
- Clone detection in source code, using vector space analysis (Marcus and Maletic, 2001; Grant and Cordy, 2009)
- Connect program documentation to source code, using latent semantic indexing (Marcus and Maletic, 2003)
Our Parser

- MaltParser (maltparser.org):
  - Data-driven parser generator for dependency analysis, recently extended to phrase structure (Hall and Nivre, 2008)
- Linear time complexity
  - Parsing units = source files (can have thousands of tokens)
- Trained on examples (no grammar)
  - Training data can be generated automatically for $L$
- Robust:
  - Produces exactly one output tree for any input


Method

Phase (i): Training

Training Program: \( T \in L \)

(a) \rightarrow Correct Syntax Trees of \( T \)

(b) \rightarrow Correct Dep. Trees of \( T \)

Parser Model

Learn

Phase (ii): Production

Program: \( P \in L' \approx L \)

(a) \rightarrow Parse

MaltParser

Parsed Dep. Trees of \( P \)

Parse

Evaluation

Correct Syntax Trees of \( P \)

Compare

Parsed Syntax Trees of \( P \)
while (count > 0) {
    stack[--count]=null;
}

Example: Training program ⇒ Syntax Trees
Example: Syntax Trees $\Rightarrow$ Dependency Trees

$\text{While} \ (\text{count} > 0) \{ \text{stack} [-- \text{count}] = \text{null} ; \}

= \text{LEFT}

$\text{While} \ (\text{identifier} > \text{IntLiteral}) \{ \text{identifier} [-- \text{identifier}] = \text{NullLiteral} ; \}

= \text{FREQ}$
Experimental Results: Java Syntax Trees

- Project: Recoder (400 source files, 92k LOC, 335k tokens)
- 80% for training, 10% for evaluation

<table>
<thead>
<tr>
<th></th>
<th>Labeled F-score</th>
<th>Unlabeled F-score</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>FREQ</td>
<td>LEFT</td>
</tr>
<tr>
<td>Unlexicalized</td>
<td>82.1</td>
<td>93.5</td>
</tr>
<tr>
<td>Lexicalized</td>
<td>89.7</td>
<td>97.7</td>
</tr>
</tbody>
</table>

- The choice of head-finding strategy is very important
  - Incremental left-to-right parser
- Names of variables, methods, classes, etc., actually contain valuable information for the parser
  - Naming conventions; certain method names, such as `equals`, are seldom used for variables
Experimental Results: Language Independence

- Labeled F-score for syntax trees using \textsc{LeFt}

<table>
<thead>
<tr>
<th></th>
<th>Java</th>
<th>Python</th>
<th>C/C++</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unlexicalized</td>
<td>93.5</td>
<td>91.5</td>
<td>95.6</td>
</tr>
<tr>
<td>Lexicalized</td>
<td>97.7</td>
<td>99.1</td>
<td>96.5</td>
</tr>
</tbody>
</table>

- C/C++ has highest labeled F-score with unlexicalized model but lowest with lexicalized model
  - C/C++ has less verbose syntax, making lexical features (names of variables, methods, etc.) less informative
Confusion Matrix: Java Dependency Trees

<table>
<thead>
<tr>
<th>Correct Label</th>
<th>Parsing Label</th>
</tr>
</thead>
<tbody>
<tr>
<td>FieldReference</td>
<td>VariableReference</td>
</tr>
<tr>
<td>VariableReference</td>
<td>FieldReference</td>
</tr>
<tr>
<td>MethodDeclaration</td>
<td>LocalVariableDeclaration</td>
</tr>
<tr>
<td>Conditional</td>
<td>FieldReference</td>
</tr>
<tr>
<td>NotEquals</td>
<td>MethodReference</td>
</tr>
<tr>
<td>Plus</td>
<td>MethodReference</td>
</tr>
<tr>
<td>Positive</td>
<td></td>
</tr>
<tr>
<td>LessThan</td>
<td>FieldReference</td>
</tr>
<tr>
<td>GreaterOrEquals</td>
<td>FieldReference</td>
</tr>
<tr>
<td>Divide</td>
<td>FieldReference</td>
</tr>
<tr>
<td>Modulo</td>
<td>FieldReference</td>
</tr>
<tr>
<td>LessOrEquals</td>
<td>FieldReference</td>
</tr>
<tr>
<td>Equals</td>
<td>NotEquals</td>
</tr>
<tr>
<td>LessOrEquals</td>
<td>Equals</td>
</tr>
<tr>
<td>NotEquals</td>
<td>Equals</td>
</tr>
</tbody>
</table>

- Confuses a reference to a class attribute with a reference that could also be a local variable or vice versa
  - Type and name analysis can easily resolve this
- MethodDeclaration not LocalVariableDeclaration:
  - Postprocessing rule:
    LocalVariableDeclaration followed by opening parenthesis (always recognized correctly) is a MethodDeclaration
- Expression level errors do not matter
Conclusions and Future Work

- Main advantage over robust approaches used so far is rapid adaptability to new languages:
  - No explicit specification using grammar and transformation rules
  - Machine learning applied to examples
  - Training data generated automatically for $L$ (requires parser)
  - Parsing extended to $L' \approx L$ (requires no parser)

- Promising results, but more work is needed:
  - From parser output to input for software analysis tools:
    - class hierarchy graphs (done for Java: 100% accurate)
    - type reference graphs (done for Java: 100% accurate)
    - call graphs
  - Apply the approach to more dialects of C/C++: analyzing correct, incomplete, and erroneous programs for standard C and its dialects