Generating Rewritable Abstract Syntax Trees

A Foundation for the Rapid Development of Source Code Transformation Tools

Jeffrey L. Overbey and Ralph E. Johnson

University of Illinois at Urbana-Champaign, Urbana IL 61801, USA,
{overbey2,johnson}@cs.uiuc.edu

Abstract. Building a production-quality refactoring engine or similar source code transformation tool traditionally requires a large amount of hand-written, language-specific support code. We describe a system which reduces this overhead by allowing both a parser and a *fully rewritable* AST to be generated automatically from an *annotated* grammar, requiring little or no additional hand-written code. The rewritable AST is ideal for implementing program transformations that preserve the formatting of the original sources, including spacing and comments, and the system can be augmented to allow transformation of C-preprocessed sources even when the target language is not C or C++. Moreover, the AST design is fully customizable, allowing it to resemble a hand-coded tree. The amount of required annotation is typically quite small, and the annotated grammar is often an order of magnitude smaller than the generated code.

1 Introduction

To many programmers, the *Refactor › Extract Method* menu item is "just another feature" of an IDE, one with the same visual precedence as *Edit › Copy* or *File › Print*. However, to the IDE developer, these features are not at all comparable: Building a refactoring tool is a substantial development effort. Moreover, it requires a substantial amount of infrastructure, most of which is usually written by hand.

The central data structure in a refactoring engine is usually a *rewritable AST,* that is, an abstract syntax tree whose nodes can be added to, rearranged, removed, and replaced *to perform textual transformations on the underlying source code.* Nearly all of a refactoring engine's components depend on the AST, so it must be implemented before any significant program transformations or analyses. And, unfortunately, it is a component that is often very time-consuming to code by hand.

This paper describes a system that generates an abstract syntax tree from a grammar. The grammar can be *annotated* to customize the AST design, and then it is used to generate AST node classes as well as a parser which constructs ASTs comprised of these nodes. The generated ASTs are *rewritable* in the sense that refactorings and other source code transformations can be coded by restructuring the AST, and the revised source code can be emitted, preserving all of the user's formatting, including spacing and comments; there is no need to develop a prettyprinter. The system can even be extended to handle preprocessed code. Moreover, while the generated AST nodes

are usually comparable to hand-coded AST nodes, they can be customized, replaced, or intermixed with hand-coded AST nodes. This AST generation system has been implemented in a tool called Ludwig and has been used to generate rewritable ASTs for several projects, most notably Photran, an open source refactoring tool for Fortran.

The remainder of this paper is organized around our three major contributions. §2 describes how to annotate a parsing grammar to produce a satisfactory AST. Although many AST generators already exist, our system of annotations is new and is fundamentally different from (and arguably more concise than) all of the existing AST specification languages. §3 describes how to augment an AST to allow both rewriting and faithful reproduction of the original source code. Although it is fairly straightforward, this process ("concretization") also appears to be new. §4 discusses modifications necessary to support preprocessed source text. This is the first work, to our knowledge, that addresses supporting C preprocessor directives in arbitrary languages; here, our major contribution is in how we handle conditional compilation. §5 gives empirical results on the ASTs we generate; our work is compared and contrasted with related work in §6, and the paper concludes in §7.

2 Abstract Syntax Annotations

The grammar supplied to a parser generator defines the *concrete* syntax of the language, which is almost always different from an ideal *abstract* syntax. For example, consider the following grammar for a language where a program is simply a list of statements, and a statement is either an if-statement, an unless-statement, or a print-statement.

‹*program*›	::=	‹*program*› ‹*stmt*› | ‹*stmt*›	(1)
‹*stmt*›	::=	‹*if-stmt*› | ‹*unless-stmt*› | ‹*print-stmt*›	(2)
‹*if-stmt*›	::=	IF ‹*expr*› THEN ‹*stmt*› ENDIF	(3)
	|	IF ‹*expr*› THEN ‹*stmt*› ELSE ‹*stmt*› ENDIF	(4)
‹*unless-stmt*›	::=	UNLESS ‹*expr*› ‹*stmt*›	(5)
‹*print-stmt*›	::=	PRINT ‹*expr*›	(6)
	|	PRINT ‹*expr*› TO STDERR	(7)
‹*expr*›	::=	TRUE | FALSE | LITERAL-STRING	(8)

There are several ways to generate an AST directly from the grammar. Clearly, such an AST will not have an ideal design, but it is useful as a starting point; in this section, we will define six annotations which allow us to refine the AST's design. The AST nodes we generate will be classes comprised of public fields, although adapting our technique to generate properly-encapsulated classes or even non-object-oriented ASTs (e.g., using structs and unions in C or algebraic data types in ML) is straightforward.

One obvious method for generating an AST directly from the grammar is to

- generate one AST node class for each nonterminal, where
- this class contains one field for each symbol on the right-hand side of one of that nonterminal's productions.

For example, the AST nodes corresponding to ‹*print-stmt*› and ‹*stmt*› would be the following, where *Token* is the name of a class representing tokens returned by a lexical analyzer.

```
class PrintStmtNode {
  public Token print;
  public ExprNode expr;
  public Token to;
  public Token stderr;
}
```

```
class StmtNode {
  public IfStmtNode ifStmt;
  public UnlessStmtNode unlessStmt;
  public PrintStmtNode printStmt;
}
```

When these classes are instantiated in an AST, irrelevant fields will be set to `null`.

2.1 Annotation 1: Omission

This method for generating ASTs has several things wrong with it. One of the most obvious is that keywords like IF and PRINT are almost never included in an AST. We will indicate that these tokens can be *omitted* or *elided* by ~~striking out~~ their symbols in the grammar.[1] For example,

⟨*print-stmt*⟩ ::= ~~PRINT~~ ⟨*expr*⟩
| ~~PRINT~~ ⟨*expr*⟩ ~~TO~~ STDERR

would generate a *PrintStmtNode* with only two fields: `expr` and `stderr`.

Generalizing this slightly, we will establish the following rules.

- If a symbol on the right-hand side of a production is annotated for omission, no field is generated for that symbol.
- If a nonterminal on the left-hand side of a production is annotated for omission, no AST node class is generated for that nonterminal.[2]

2.2 Annotation 2: Labeling

Determining the names of AST classes and their fields from the names of nonterminal and terminal symbols in the grammar is often sufficient, but sometimes these names need to be customized. This can be done by *explicitly labeling* symbols in the grammar and interpreting these labels as follows.

- Labeling the nonterminal on the left-hand side of a production determines the name of the AST node class to generate.
- Labeling a nonterminal or terminal symbol on the right-hand side of a production determines the name of the field to which that symbol corresponds.

The idea of labeling symbols is simple, yet it is extremely powerful, having several uses and implications.

---

[1] In our implementation, we represent the same in ASCII by prefixing the symbol with a hyphen and a colon, as in `-:print`.

[2] The ability to omit entire AST node classes is useful when only a partial AST is desired. For example, the Eclipse JDT [10] and CDT [9] both contain a "lightweight" AST which describes high-level organizational structures (classes, methods, etc.) but not statements or expressions.

Labeling to Distinguish Fields Labeling is the only annotation that is strictly required, at least in certain cases. *IfStmtNode* is one example. The problem is production (4):

$$\langle \textit{if-stmt} \rangle ::= \textsc{if}\ \langle \textit{expr} \rangle\ \textsc{then}\ \langle \textit{stmt} \rangle\ \textsc{else}\ \langle \textit{stmt} \rangle\ \textsc{endif}$$

Since the symbol ‹*stmt*› appears twice in the same production, we need *two* fields so that the node can have separate fields for the then-statement and the else-statement. We will accomplish this by adding distinctive labels to these symbols in the grammar, rewriting productions (3) and (4) to generate distinct fields for the two occurrences of ‹*stmt*›.

$$\begin{aligned}\langle \textit{if-stmt} \rangle ::= {} & \cancel{\textsc{if}}\ \langle \textit{expr} \rangle\ \cancel{\textsc{then}}\ \overbrace{\langle \textit{stmt} \rangle}^{\text{thenStmt}}\ \cancel{\textsc{endif}} \\ | {} & \cancel{\textsc{if}}\ \langle \textit{expr} \rangle\ \cancel{\textsc{then}}\ \overbrace{\langle \textit{stmt} \rangle}^{\text{thenStmt}}\ \cancel{\textsc{else}}\ \overbrace{\langle \textit{stmt} \rangle}^{\text{elseStmt}}\ \cancel{\textsc{endif}}\end{aligned}$$

This will generate the following node instead.

```
class IfStmtNode {
  public ExprNode expr;
  public StmtNode thenStmt;
  public StmtNode elseStmt;
}
```

Labeling to Merge Fields Just as we can give symbols distinct labels to create distinct fields, we can also give several several symbols the *same* label to assign them to the *same* field ... as long as they occur in different productions. One example of this appears above: Since both occurrences of ‹*stmt*› were given the label *thenStmt*, the `thenStmt` field will be populated regardless of whether production (3) or (4) was matched.

Labeling to Rename Fields Labeling can also be used to avoid illegal or undesirable field names. For example, were the IF token not omitted, it would generate a field named `if`, which is illegal in Java, so it could be labeled *ifToken* instead. Similarly, ‹*expr*› might be labeled *guardingExpression* to make its corresponding field name more descriptive.

Labeling to Distinguish, Rename, and Merge Node Classes Just as labeling the symbols on the right-hand sides of productions allows us to distinguish, rename, and merge fields, labeling the nonterminals on the left-hand sides of productions allows us to distinguish, rename, and merge AST node classes.

The text of the label assigned to a left-hand nonterminal determines the name of the AST node class generated for that nonterminal. By assigning a distinct label to each left-hand nonterminal, we ensure that a different AST node class will be generated for each nonterminal. By assigning the same label to several left-hand nonterminals, they can all correspond to the same AST node class.

But when is it useful to have just one node class correspond to several nonterminals?

Sometimes a grammar uses several nonterminals to refer to the same logical entity. This is probably most common in expression grammars, as we will see later, but we can illustrate it with our sample grammar. Note that our sample programming language contains two conditional constructs: an if-statement and an unless-statement. Suppose we want to represent both of these with a single node, *ConditionalStmtNode.* We will label the *left-hand* nonterminals with this name

$$\overbrace{\langle \textit{if-stmt} \rangle}^{\text{ConditionalStmtNode}} ::= \sout{\texttt{IF}}\ \langle \textit{expr} \rangle\ \sout{\texttt{THEN}}\ \overbrace{\langle \textit{stmt} \rangle}^{\text{thenStmt}}\ \sout{\texttt{ENDIF}}$$
$$\mid\ \sout{\texttt{IF}}\ \langle \textit{expr} \rangle\ \sout{\texttt{THEN}}\ \overbrace{\langle \textit{stmt} \rangle}^{\text{thenStmt}}\ \sout{\texttt{ELSE}}\ \overbrace{\langle \textit{stmt} \rangle}^{\text{elseStmt}}\ \sout{\texttt{ENDIF}}$$
$$\overbrace{\langle \textit{unless-stmt} \rangle}^{\text{ConditionalStmtNode}} ::= \sout{\texttt{UNLESS}}\ \langle \textit{expr} \rangle\ \overbrace{\langle \textit{stmt} \rangle}^{\text{elseStmt}}$$

in order to generate a *ConditionalStmtNode* class with the same three fields as before (cf. page 4). When an if-statement is matched, the `expr` and `thenStmt` fields will be set, and the `elseStmt` field may or may not be `null`. When an unless-statement is matched, `expr` and `elseStmt` will be set, but `thenStmt` will always be `null`.[3]

2.3 Annotation 3: Boolean Access

Consider the print-statement, defined in productions (6) and (7). Suppose that a print-stmt can write either to standard output or standard error, depending on whether or not the **to stderr** clause is present. Although we can omit the PRINT and TO tokens from the *PrintStmtNode* class, we cannot also omit the STDERR token without losing the semantic distinction between the two variants of the print-statement. The token STDERR is not important *per se;* what matters is whether or not it was present at all.

We will annotate a symbol with "(bool)" to indicate that its field should be of type `boolean` rather than of type `Token`. When the corresponding token is present, the field's value will be set to true; otherwise, it will be set to false. Accordingly,[4]

$$\langle \textit{print-stmt} \rangle ::= \sout{\texttt{PRINT}}\ \langle \textit{expr} \rangle$$
$$\mid\ \sout{\texttt{PRINT}}\ \langle \textit{expr} \rangle\ \sout{\texttt{TO}}\ \overbrace{\text{STDERR}}^{\text{(bool)}}$$

will generate the node class

```
class PrintStmtNode {
  public ExprNode expr;
  public boolean stderr ;
}
```

whose `stderr` field can be tested to determine whether the print-statement is intended to write to standard output or standard error.

---

[3] This is based on the intuition that **unless** *E S* is equivalent to **if** *E* **then no-op else** *S*.

[4] Again, our implementation uses a straightforward ASCII equivalent: `(bool):stderr`. Also, one should note that some obvious extensions of this annotation are possible, such as ones to assign an integer or an *enum* value.

2.4 Annotation 4: List Formation

Recursive productions are idiomatically used to specify lists. In our example grammar, the productions on line (1) indicate that a program is a list of one or more statements. In an AST, it is usually preferably to replace these recursive structures with an array, list, or whatever iterable construct is most common in the implementation language.

We will annotate left-hand nonterminals with "(list)" to indicate that the productions for that nonterminal describe a list.

$$\overbrace{\langle program \rangle}^{\text{(list)}} ::= \langle program \rangle\ \langle stmt \rangle \mid \langle stmt \rangle$$

Now, there is no need to generate a *ProgramNode* class: In its place, we can simply use a `List<StmtNode>`.

2.5 Annotation 5: Inlining

To illustrate our next annotation, suppose the productions defining an if-statement had been written as follows. Note that the syntax of the if-statement has not changed: The grammar is just slightly different.

$$\begin{aligned}
\langle if\text{-}stmt \rangle &::= \langle if\text{-}then\text{-}part \rangle\ \langle endif\text{-}part \rangle \\
&\quad \mid \langle if\text{-}then\text{-}part \rangle\ \langle else\text{-}part \rangle\ \langle endif\text{-}part \rangle \\
\langle if\text{-}then\text{-}part \rangle &::= \sout{\text{IF}}\ \langle expr \rangle\ \sout{\text{THEN}}\ \overbrace{\langle stmt \rangle}^{\text{thenStmt}} \\
\langle else\text{-}part \rangle &::= \sout{\text{ELSE}}\ \overbrace{\langle stmt \rangle}^{\text{elseStmt}} \\
\sout{\langle endif\text{-}part \rangle} &::= \text{ENDIF}
\end{aligned}$$

Compare these to productions (3) and (4) in the original sample grammar. In this version, IF ‹*expr*› THEN ‹*stmt*› has been "factored out" into its own nonterminal, ‹*if-then-part*›. This is commonly done to minimize duplication in the grammar. However, left alone, it adds unnecessary nodes to an AST. In this case, there will be *three* AST nodes, all devoted to defining the structure of an if-statement: *IfStmtNode, IfThenPartNode,* and *ElsePartNode*.[5]

To create the same nodes as before, we would *like* to do away with *IfThenPartNode* and *ElsePartNode* and instead have their fields—`expr`, `thenStmt`, and `elseStmt`—placed directly into the *IfStmtNode* class. In other words, we would like to *inline* these nodes: In the *IfStmtNode* class, rather than declaring an *IfThenPartNode* field, we will simply insert all of the fields that would be in an *IfThenPartNode* instead. We will denote this with an "(inline)" annotation

$$\begin{aligned}
\langle if\text{-}stmt \rangle &::= \overbrace{\langle if\text{-}then\text{-}part \rangle}^{\text{(inline)}}\ \langle endif\text{-}part \rangle \\
&\quad \mid \overbrace{\langle if\text{-}then\text{-}part \rangle}^{\text{(inline)}}\ \overbrace{\langle else\text{-}part \rangle}^{\text{(inline)}}\ \langle endif\text{-}part \rangle \\
\sout{\langle if\text{-}then\text{-}part \rangle} &::= \sout{\text{IF}}\ \langle expr \rangle\ \sout{\text{THEN}}\ \overbrace{\langle stmt \rangle}^{\text{thenStmt}} \\
\sout{\langle else\text{-}part \rangle} &::= \sout{\text{ELSE}}\ \overbrace{\langle stmt \rangle}^{\text{elseStmt}} \\
\sout{\langle endif\text{-}part \rangle} &::= \text{ENDIF}
\end{aligned}$$

[5] Notice that we have omitted ‹*endif-part*›, since its AST node is unnecessary.

which gives us the desired AST node. Notice that we have now omitted ‹*if-then-part*› and ‹*else-part*›: Since their contents are always inlined, there is no reason to generate these node classes.

```
class IfStmtNode {
  public ExprNode expr;      // Inlined from IfThenPartNode
  public StmtNode thenStmt;  // Inlined from IfThenPartNode
  public StmtNode elseStmt;  // Inlined from ElsePartNode
}
```

2.6 Annotation 6: Superclass Formation

Idiomatic Form The productions in line (2) illustrate another common idiom in BNF grammars:

‹*stmt*› ::= ‹*if-stmt*› | ‹*unless-stmt*› | ‹*print-stmt*›

states that an if-statement is a statement, an unless-statement is a statement, and a print-statement is a statement. In object-oriented languages, this *is-a* relationship is generally modeled using inheritance. Instead of generating a *Stmt* node with fields for the various types of statements, we can instead make *Stmt* an abstract class (or interface, in Java or C#) which is *subclassed* by *IfStmt, UnlessStmt,* and *PrintStmt.* We will indicate this preference by a "(superclass)" annotation on the left-hand nonterminal.

$\overbrace{\text{‹}stmt\text{›}}^{\text{(superclass)}}$ ::= ‹*if-stmt*› | ‹*unless-stmt*› | ‹*print-stmt*›

This generates the following.[6]

```
interface StmtNode { /*empty*/ }
class IfStmtNode implements StmtNode { ... }
class UnlessStmtNode implements StmtNode { ... }
class PrintStmtNode implements StmtNode { ... }
```

Non-idiomatic Form As in the preceding example, the (superclass) annotation is generally applied to a nonterminal whose productions are all of the form $A ::= B$, for nonterminals A and B. But it is also possible to apply this annotation when the productions do not have this form. To do this, we must explicitly label each *production* with a node class name. For example,

$\overbrace{\text{‹}if\text{-}stmt\text{›}}^{\text{(superclass)}}$ ::=

IF ‹*expr*› THEN $\overbrace{\text{‹}stmt\text{›}}^{\text{thenStmt}}$ ENDIF } IfThenNode

| IF ‹*expr*› THEN $\overbrace{\text{‹}stmt\text{›}}^{\text{thenStmt}}$ ELSE $\overbrace{\text{‹}stmt\text{›}}^{\text{elseStmt}}$ ENDIF } IfThenElseNode

allows us to have two *different* nodes for an if-statement, one for the if-then form and another for the if-then-else form.

[6] In our implementation, the interface/abstract superclass contains no fields or methods, as shown here. Later in this paper, we describe several ways to customize generated nodes. This empty interface is often an excellent candidate for customization, since certain behaviors may be common among the various subclasses.

```
interface IfStmtNode { /*empty*/ }

class IfThenNode
  implements IfStmtNode {
  public ExprNode expr;
  public StmtNode thenStmt;

}
```

```
class IfThenElseNode
  implements IfStmtNode {
  public ExprNode expr;
  public StmtNode thenStmt;
  public StmtNode elseStmt;
}
```

It should be noted that the labeling principles discussed in §2.2 apply to production labels as well. For example, two productions (or a production and a nonterminal) can be given the same label to assign them to the same node class.

2.7 Customization

In our experience, these annotations—omission, labeling, Boolean access, list formation, inlining, and superclass formation—allow satisfactory AST nodes to be generated for most constructs in most languages. However, it is sometimes desirable to "tweak" some of the generated AST node classes or to mix them with non-generated nodes. Our implementation [24] provides several means to customize the generated AST; the reader is referred to its documentation for details. In our experience, the two most important customizations are the following.

- *The user must be able to add methods to the generated node classes.* For example, it may be desirable to add a `getType()` method to expression nodes or a `resolveBinding()` method to identifier nodes. Our implementation follows a best practice described by Vlissides [35, p. 85]: The system generates an AST node class, and the user places additional methods in a custom subclass (or superclass).
- *The user must be able to write custom AST nodes when necessary.* Sometimes, the "obvious" grammatical representation of a language construct does not satisfy the constraints of the parser generator; this can result in productions which deviate wildly from the conceptual structures of the constructs they are intended to represent. In these cases, the user must be able to hand-code an AST node for that construct. Being able to intermix hand-coded and generated nodes is critical, because it provides the user with the flexibility of hand-coding when it is needed while alleviating the tedium and cost of developing and maintaining an entirely hand-coded infrastructure.

2.8 An Example

We will conclude this section with a fairly complex example: Using annotations to construct an AST for an expression grammar.

$$
\begin{array}{lcll}
\overbrace{\langle expr \rangle}^{\text{IExpression (superclass)}} & ::= & \overbrace{\langle expr \rangle}^{\text{lhs}}\ \overbrace{\text{PLUS}}^{\text{isPlus (boolean)}}\ \overbrace{\langle term \rangle}^{\text{rhs}} & \} \text{ BinaryExpr} \quad (1) \\
 & | & \langle term \rangle & (2) \\
\overbrace{\langle term \rangle}^{\text{IExpression (superclass)}} & ::= & \overbrace{\langle term \rangle}^{\text{lhs}}\ \overbrace{\text{TIMES}}^{\text{isTimes (boolean)}}\ \overbrace{\langle factor \rangle}^{\text{rhs}} & \} \text{ BinaryExpr} \quad (3) \\
 & | & \langle factor \rangle & (4) \\
\overbrace{\langle factor \rangle}^{\text{IExpression (superclass)}} & ::= & \overbrace{\langle factor \rangle}^{\text{lhs}}\ \overbrace{\text{EXP}}^{\text{isExp (boolean)}}\ \overbrace{\langle primary \rangle}^{\text{rhs}} & \} \text{ BinaryExpr} \quad (5) \\
 & | & \langle primary \rangle & (6) \\
\overbrace{\langle primary \rangle}^{\text{IExpression (superclass)}} & ::= & \langle constant \rangle & (7) \\
 & | & \sout{\text{LPAREN}}\ \langle expr \rangle\ \sout{\text{RPAREN}} & (8) \\
\langle constant \rangle & ::= & \text{INTEGER-CONSTANT} & (9) \\
 & | & \text{REAL-CONSTANT} & (10)
\end{array}
$$

The preceding grammar is a stereotypical example of an unambiguous grammar for arithmetic expressions. From lowest to highest precedence, it incorporates addition (left-associative), multiplication (left-associative), exponentiation (right-associative), and nested expressions. Annotations are used as follows.

- *Superclass Formation.* Production (1) indicates that a superclass, *IExpression,* will be generated, and that expressions of the form ⟨*expr*⟩ PLUS ⟨*term*⟩ will be parsed into a node called *BinaryExpression* which implements *IExpression.* Production (2) indicates that the AST node class for ⟨*term*⟩ will also implement *IExpression;* however, since the AST node class for ⟨*term*⟩ *is IExpression,* this has no effect. Likewise, production (7) indicates that *Constant,* the AST node class for ⟨*constant*⟩, will implement *IExpression.* Production (8) indicates that the AST node for ⟨*expr*⟩ should implement *IExpression,* but, again, this is trivially true.
- *Labeling.* Labels are used to name the fields in *BinaryExpr;* they are used to assign the same AST node class, *IExpression,* to ⟨*expr*⟩, ⟨*term*⟩, ⟨*factor*⟩, and ⟨*primary*⟩; and they are used to assign productions (1), (3), and (5) to the same node class, *BinaryExpr.* The parentheses delineating a nested expression in (8) are omitted, so they will not be included in the AST.
- *Boolean Access.* The tokens PLUS, TIMES, and EXP are assigned Boolean fields in the *BinaryExpr* node class.[7]

Ultimately, this results in the following node classes.

interface IExpression { /∗ *empty* ∗/ }

[7] This was primarily for illustrative purposes. We could have given productions (1), (3), and (5) different labels to have separate nodes types for addition, multiplication, and exponentiation expressions.

```
class BinaryExpr implements IExpression {
  public IExpression lhs;
  public IExpression rhs;
  public boolean isPlus;
  public boolean isTimes;
  public boolean isExp;
}
```

```
class Constant implements IExpression {
  public Token integerConstant;
  public Token realConstant;

}
```

3 Rewritable Abstract Syntax Trees

We will next present how to generate *rewritable* ASTs, which allow source code to be modified simply by adding, changing, moving, and removing nodes in the AST.

Traditionally, source code is modified either by prettyprinting a modified AST or by computing textual edits from an AST. Prettyprinting is a popular choice for source-to-source compilers and academic/research source code transformation tools (including Stratego/XT [34, 7], TXL [5], ASF+SDF [33], and many academic refactoring tools): Prettyprinting is straightforward to implement, and preserving comments and source formatting is usually a non-goal. On the other hand, commercial refactoring tools (including the Eclipse JDT [10] and CDT [9], NetBeans [2], and Apple Xcode [3]) generally use textual edits: AST nodes are mapped to offset/length regions in the source file, and the source text is changed by manipulating a text buffer directly (by specifying text to be added, removed, or replaced at particular offsets) or indirectly (by manipulating AST nodes and computing text buffer changes from the AST modifications). Prettyprinting is easier to implement but sacrifices output quality; textual edits produce "better" results at the expense of a significantly more complicated implementation.

Our solution fuses these two approaches: We will add "missing pieces"—dropped tokens, spaces, comments, etc.—back into the AST, but they will not be visible in its public interface. This will allow us, internally, to reproduce the original source text exactly using a simple traversal. Moreover, they will be tied to individual nodes in such a way that they move *with* the nodes when the AST is modified.

3.1 Whitetext Augmentation

If an AST contains every token in the original text, in the original order, the original source code can be reproduced almost exactly. The only pieces missing are what we refer to as *whitetext:* spaces, comments, line continuation characters, and similar lexical constructs. In order to reproduce the original text *exactly—including* whitetext—we propose the following.

- Rather than discarding whitetext, the lexical analyzer must "attach" all whitetext to exactly one token: either the token preceding it or the one following it.
- Most whitetext is attached to the *following* token.
- However, whitetext appearing at the end of a line is considered to be part of the *preceding* token, along with the carriage return/linefeed following it. Any additional whitetext beyond the carriage return/linefeed is attached to the *following* token.

The latter part of this heuristic ensures that any trailing comments on a line, as well as the carriage return/linefeed itself, are associated with the preceding token, while any subsequent indentation is associated with the token on the next line.

3.2 AST Concretization

Note that our heuristic uses tokens to *partition* the original source text: Every character in the original text is also present in a token, every character in a token is also present in the original text, and there is no overlap between tokens. Moreover, the characters retain their original source text order.[8]

Since a whitetext-augmented token stream partitions the original source text, it can be used to reproduce exact source text. Thus, it should also be possible to reproduce source text from an AST ... as long as every token is present, in the original order. Reviewing our list of AST annotations, we can see that this is not necessarily the case:

1. Tokens may be omitted.
2. Node classes may be omitted.
3. Nodes may be replaced with Boolean values.
4. A node's children are assigned to named fields; however, since several productions may be assigned to a single type of node, there is not necessary a single order in which these children can be traversed to preserve the original order. (For example, consider the node generated for the productions $A ::= ab \mid ba$.)

We can overcome these problems with the following, respectively.

1. Rather than omitting tokens from node classes, store them in a *private* field, making them accessible for source text reproduction while remaining absent from the node's interface.
2. When an omitted node class is used (and not inlined), simply store a string containing the node's original text or, equivalently, a list/array of tokens.
3. Rather than storing a Boolean value, store the original node/token, and provide a Boolean accessor method for that field.
4. If there *is* a single order in which children can be traversed to preserve the original token order, there is no problem; if no such order exists, then the node must include a field indicating the appropriate traversal order.

One method to determine an order for printing a node's children is the following. Each production generating a particular node defines a partial order on some of that node's children. The union of these partial orders is a relation describing the ordering of all of the node's children. We may treat this relation as a directed graph and topologically sort it using an algorithm that also detects cycles [6, p. 546]. If no cycles are found, the sorted graph gives a correct (total) order for printing the node's fields; if cycles are found, no such order exists.

[8] This assumes that there is at least one token in the original file. If the original source consists solely of whitetext—say, a C program that contains only a comment—this must be treated as a special case.

3.3 AST Rewriting

When the above changes are made, it is possible to reproduce the exact source code from which an AST was created simply by traversing the tree and outputting each token and its associated whitetext. However, such a tree is also ideal for *rewriting*. When a node is moved or removed within the tree, every token under that node—omitted or not—is moved with it, as are any comments and line endings associated with that node. Suppose, for example, that an *IfStmtNode* is moved to a new location in the AST: When the modified AST is traversed to reproduce source code, the entire if-statement—including the IF token, the line ending, and any comments—will appear at the new location within the source code.[9]

4 Representing Preprocessed Text

So far, we have ignored one mundane but highly nontrivial aspect of source code transformation: handling preprocessed code. Many tools—including `sed`, `m4`, and even Perl—can serve as preprocessors, making the topic impossible to treat in full generality. Thus, we will focus on one of the most commonly used tools: the C preprocessor. Its capabilities are stereotypical, so the conceptual model we develop is applicable, for example, when handling INCLUDE lines in Fortran or simple `sed` substitutions as well. We will assume that the reader is already familiar with the C preprocessor; a good introduction is provided in the GCC documentation [31], while the C and C++ language specifications [18, 19] provide normative references and are the source of the C preprocessor terminology in this section.

We will follow the *pseudo-preprocessing* model of [14]: We will assume that a customized version of the preprocessor will feed the lexical analyzer. While a normal preprocessor simply executes preprocessing directives and outputs either a stream of text or a stream of tokens, our "pseudo-preprocessor" will keep track of what output originated from what directives, passing this information on to the lexer (and parser) so that it can be incorporated into the AST. Retaining information about preprocessor directives in the AST is essential both for analyzing preprocessed code and for reproducing the original (un-preprocessed) source code from the AST.

The C preprocessor has three aspects to consider: substitutions, control lines, and conditional compilation.

4.1 Substitutions

Trigraphs,[10] backslash-newline sequences, `#include` directives, and macro expansions are all essentially textual substitutions, i.e., the logical text for some tokens may differ

[9] One should note that the if-statement will retain its indentation from the previous location. If the level of indentation needs to be adjusted, this must be done manually by modifying the whitetext of the tokens under the affected node.

[10] A trigraph is a sequence of three characters that takes the place of another character. For example, `??(` is equivalent to `{`.

from the physical text in the unpreprocessed file. These can be handled using a technique essentially similar to Garrido's [14], marking the affected tokens with the token's original text (or original preprocessor directive) from the unpreprocessed file and printing this, rather than the token's logical text, during source reproduction.

4.2 Control Lines

Control lines include macro definitions (`#define` and `#undef`), the line control directive (`#line`), `#error`, `#pragma`, and the null directive (`#` followed by a newline).[11] We will treat these as whitetext, but how we do so depends on the application. When source reproduction is the only goal, they can be treated as strings and affixed to tokens like spaces and comments. When some analysis is required, it is useful to represent them with nodes in the AST. One must note that inserting a directive in the middle of a statement

```
int
#define MACRO
some_function() {}
```

is perfectly legal; directives do not necessarily appear at statement boundaries. This may be particularly true when languages other than C and C++ are preprocessed using the C preprocessor. So in general, no assumptions can be made about where directives will or will not need to be placed in an AST.

One means to include nodes for these directives in the AST is to make tokens' associated whitetext a *sequence* of strings and preprocessor directives. The preprocessor directives can be treated as children of tokens during a traversal; this makes it possible to include them, for example, when implementing the Visitor pattern [12].

4.3 Conditional Compilation and Multiple Configurations

Conditional compilation is by far the most challenging feature to handle in a source code transformation tool. Consider the following example.

```
if (
 #if defined(A) || defined(B)
 variable
 #else
 function() < 1 && variable
 #endif
 < 2) x = 3;
```

We cannot treat conditional directives as whitetext (as we did for control lines) because doing so results in the token sequence

```
if (variable function() < 1 && variable < 2) x = 3;
```

which will not even parse.

[11] The `#include` directive is also a control line but is treated uniquely for our purposes.

Conceptually, conditional inclusion allows for variation in the AST depending on the preprocessor's *configuration,* that is, the set of macro definitions under which it is operating: The structure of the AST under one configuration is not necessarily the same as for another. If we intend to reproduce the original source code from an AST, we must be able to construct a single AST which captures *all* of the AST variations that may occur under *all* feasible configurations.

Representation Garrido's [14] technique for handling conditional compilation involves modifying the grammar to specify where conditionals may appear and then inserting a pass between lexical analysis and parsing which ensures that conditional directives appear only at these locations: If a conditional directive appears at an unexpected location, it is moved forward or backward as necessary, and tokens are copied into each branch of the conditional. This process is called *completing the conditional.* Although reasonable, it is language-specific, heuristic, and becomes complicated in the presence of nested conditionals. It also requires modifying the grammar and adding AST nodes for conditional compilation directives.

Our solution for representing multiple-configuration sources, which is intended to be language-agnostic and grammar-independent, is based on [32, p. 5], which uses a similar structure to represent ambiguities in parse trees for natural language. Figures 1(a) and (b) show the individual ASTs that would be constructed for the preceding example under each preprocessor configuration. Notice that the smallest subtree that differs between the two is the expression under the if-statement node. Figure 1(c) shows a "multiple-configuration" AST which combines the previous ASTs by inserting an "ambiguous" expression node whose children are the individual expression nodes guarded by the various preprocessor configurations.

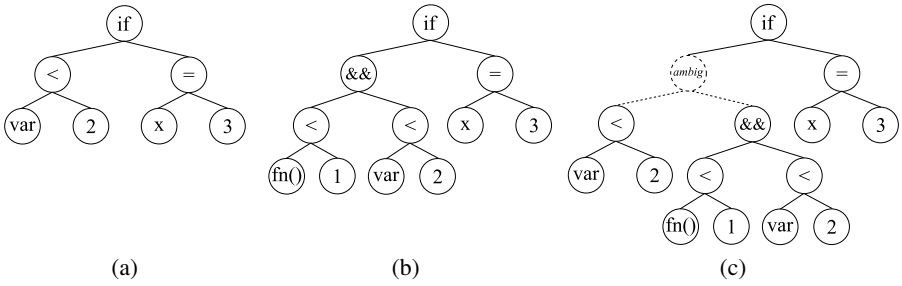

Fig. 1. (a) The AST for the configuration `defined(A) || defined(B)`. (b) The AST for the configuration `!(defined(A) || defined(B))`. (c) The multiple-configuration AST. The dotted node represents an "ambiguous" expression whose children are each guarded by a different preprocessor configuration.

This representation can be constructed by modifying an LA(1) shift-reduce parser (most likely an LALR(1) [8] parser). A detailed discussion is beyond the scope of this paper; a more substantial treatment by the present authors is in preparation. The following method is oversimplified—its positioning of ambiguous nodes in the AST is often

less than ideal—but it nevertheless conveys the gist of our technique.[12] The parser proceeds as usual until it reaches an `#ifdef` directive in the token stream. The parser clones itself so that a different copy of the parser can process each branch of the `#ifdef`. Each clone independently processes the tokens under its branch of the conditional, stopping when it is about to shift the token following the `#endif`. The clones are then compared for *equivalence:* If two or more clones are in the same state and have equivalent stack contents, those clones are merged into a single parser.[13] When parsers are merged, the topmost elements on their stacks are made to be children of an "ambiguous" node (as in Figure 1(c)), and this "ambiguous" node becomes the topmost element on the stack of the merged parser. After testing for equivalence and possibly merging parsers, each subsequent token is fed to all remaining clones, which shift the token and perform any reductions, stopping when they are prepared to shift the following token. The clones are again tested for equivalence, merged as necessary, and the cycle continues until only a single parser remains. In the worst case, this will happen at the end of the input.

An important implementation detail involves cloning parsers and testing for "equivalence" of parse stacks. When a parser is cloned, each clone should have an independent stack. However, there will be AST nodes present on the parser stack prior to cloning, and these nodes should remain pointer-identical among the clones' stacks. Now, we can define two or more parser stacks to have "equivalent" contents when (1) the number of elements on each stack is the same, (2) the AST nodes on the top of the stacks all have the same type, and (3) all elements below the top on each stack are pointer-identical among stacks. This ensures that the merging parsers, as described above, will be viable.

Source Reproduction Conditional completion and ambiguity control complicate source reproduction slightly, since some tokens may appear at several locations in a multiple-configuration AST even though they occur only once in the source text. Recall that, when a parser is cloned, any tokens already on the stack are pointer-identical among the clones; tokens beyond the `#endif` are fed to all clones, and, again, should be pointer-identical. Since the clones may place these tokens under different parents, it follows that these tokens may have *multiple* parents in the AST, i.e., the "multiple-configuration AST" is really a DAG. To reproduce the original source text, the conditional directives (`#if`, `#endif`, etc.) may be attached to tokens as whitetext, as before; during the traversal of the AST (DAG), a token's text is printed only once.

The question is, should tokens with multiple parents be printed the first time they are traversed or the last time? Clearly, any shared tokens preceding the `#ifdef` should be printed the first time they are traversed; tokens following the `#endif` should be printed the last time. We can achieve this by simply *flagging* tokens following the `#endif` that are fed to multiple clones; during a source reproduction traversal, flagged tokens are

[12] Our technique is based on theory stemming from incremental parsing [4], although it appears that the application of the same essential ideas to preprocessing was independently discovered and implemented in [29].

[13] Why is it safe to merge parsers under these conditions? In mathematical models of deterministic shift-reduce parsers, the transition relation between states is a *function* of the stack contents, the current state, and the remaining input: If all of these are identical among parsers, the parsers' subsequent behaviors will be identical.

printed the last time they are traversed (i.e., when they are traversed as a child of their rightmost parent), while all other tokens are printed the first time they are traversed.[14]

Multiple-Configuration Macro Expansion Handling multiple preprocessor configurations means also handling the possibility that macros will have several possible expansions, depending on the configuration. Fortunately, this can be handled similarly to `#ifdef`s: The parser "forks" into several clones, each of which handles one expansion, then the clones synchronously handle tokens following the macro call, eventually merging back into a single parser.

5 Empirical Results

A rewritable AST generator has been implemented in Ludwig, a lexer and parser generator which is available for download [24]. Ludwig's AST generator has been used in several projects; most notably, it has been used to generate the parser and rewritable AST in Photran [28], an open-source integrated development environment and refactoring tool for Fortran. Fortran 95 has an exorbitant amount of syntax—Photran's grammar is nearly 2,500 lines long (a similar annotated grammar for Java 1.0 is just over 600 lines)—which has made it an ideal candidate for AST generation. At the time of writing, Photran contains 329 AST node classes comprising 33,081 lines of code; the AST base classes, parser, and semantic actions comprise another 25,909 lines. In total, then, its 2,500-line annotated grammar generates nearly 59,000 lines of code.

Photran's parser was used to construct fully-concretized ASTs for 209 files from the source code for IBEAM [17], a massively parallel astrophysics framework based on the University of Chicago's FLASH code [1]. The results are summarized in Table 1. Note that the source file sizes were positively skewed ($skew = 2.99$) but fairly well-correlated with the number of AST nodes ($r = 0.77$), as one would expect in Fortran. (This would not necessarily be the case in C, which uses preprocessing much more heavily. In general, the data in Table 1 are intended to be informative, not representative, as they depend on the Fortran language, Photran's AST structure, and even IBEAM's coding conventions.)

The third section of Table 1 is perhaps the most interesting: A rough, implementation-independent approximation of the size of an AST was computed by assuming that each interior node requires 32 bytes of overhead and each token requires 64 bytes. The *non-concretized AST* size is intended to approximate the size of an AST similar to one used in a compiler, one which contains neither hidden tokens nor whitetext; it is computed as $32N_I+64N_N+C_N$ with variables defined according to Table 1. The *transformation baseline* assumes that the smallest possible AST is one in which every node has a fixed size, no hidden tokens are stored, and no text is stored (rather, each node contains an offset and length into a source file); to use this AST for source transformation, one would generally store the entire file (and any preprocessor-included files) in memory in addition

[14] Observe that a single flag will suffice regardless of the level of `#ifdef` nesting and number of parent nodes due to the lexical positioning of shared tokens.

to the AST, and so the total amount of memory required is $32N_I + 64N_N + C$. The *concretized AST* includes all source text in tokens and thus is computed as $32N_I + 64N_T + C$. The final two rows show the relative increase in memory requirements of a concretized AST over a non-concretized AST or the transformation baseline. One should note that the median increase in size due to concretization is relatively small—20% over a non-concretized AST and 14% over the transformation baseline—and even the largest concretized AST was estimated at less than 2 MB, a minimal demand on modern systems.

Description		**Median**	**Mean**	**St. Dev.**	**Min.**	**Max.**
AST nodes		2,612	5,203	7,992	15	50,084
Interior nodes	(N_I)	2,065 (79%)	3,931	6,198	8	41,003
Tokens	(N_T)	558	1,272	1,840	7	11,871
Hidden		169 (6%)	409	615	0	4,099
Not hidden	(N_N)	409 (16%)	863	1,232	7	7,772
Characters (with includes)	(C)	4,861	9,663	12,953	33	81,373
in non-hidden tokens' text	(C_N)	1,606 (33%)	3,993	5,826	27	42,612
in hidden tokens' text		223 (5%)	483	719	0	4,380
in whitetext		2,904 (60%)	5,187	6,884	6	46,521
Approximate AST size (bytes)						
Non-concretized AST		91,581	184,998	280,902	731	1,718,789
Transformation baseline		95,438	190,669	287,026	737	1,739,467
Concretized AST		108,780	216,858	325,216	737	1,942,877
Increase from non-concretized		20%	27%	16%	1%	98%
Increase from baseline		14%	15%	5%	0%	38%
Source file size (bytes)		4,114	7,926	10,891	33	69,472

Table 1. Summary of data collected from 209 Photran ASTs. The first section of the table decomposes the nodes of the concretized ASTs by type. The second describes the partitioning of characters among tokens. The third estimates AST sizes as described in §5, and the fourth gives the size of the original source files in bytes.

6 Related Work

There is an abundance of work on abstract syntax as well as work on handling the C preprocessor in the context of C and C++. The present work is distinguished by (1) its annotation-based approach for the definition of abstract syntax (as opposed to a more traditional declarative approach), (2) its language-agnostic handling of the C preprocessor, and (3) its focus on practical issues, including layout retention, customizability, and exposure as an API.

The Zephyr abstract syntax description language [36] is essentially a declarative language for "tree-like data structures" and resembles declarations of algebraic data types (à la ML). Wile [37] advocates a particular grammar notation (WBNF) which makes some aspects of abstract syntax (e.g., iteration, optionality, precedence, and nesting)

more explicit and suggests a heuristic process by which Yacc grammars can be converted and abstract syntax derived automatically. Among existing tools, ANTLR [27], LPG [23], and JavaCC/JJTree [20] all include AST generators; LPG's is derived directly from the concrete syntax, while ANTLR and JJTree rely on declarative specifications embedded in the grammar.

Our approach differs in many ways from all of these. For example, Zephyr is not integrated with a parser generator; ANTLR's ASTs allow a limited form of source rewriting [27, Ch. 9] but have a dramatically different structure than the ASTs we describe; and no existing AST generator can generate ASTs for C-preprocessed sources. However, the most prominent distinction of the present work is its annotation-based approach. Zephyr, ANTLR, and JJTree require the user to fully specify an abstract syntax. LPG constructs a primitive abstract syntax from the concrete syntax automatically. In contrast, our approach allows a *default* abstract syntax to be constructed from the concrete syntax and subsequently *refined* using annotations. This places less of an annotation burden on the programmer than requiring a fully explicit abstract syntax definition while simultaneously allowing more flexibility than a fully-inferred definition. One drawback of the annotation-based approach is that the structure of AST nodes is not immediately obvious, particularly when *(inline)* annotations are used; to remedy this, in Ludwig, we are developing a GUI which allows the user to view AST node structures as the grammar is being annotated.

Our heuristic for attaching whitetext to tokens in order to facilitate rewriting also appears to be new, although the fundamental idea of including whitetext in syntax trees appears elsewhere. Sellink and Verhoef [30] suggest placing whitespace and comments into a parse tree by (automatically) modifying the grammar to include a "layout" nonterminal prior to each occurrence of a token; Lämmel [21] includes such information as annotations in the parse tree.

To our knowledge, our work is also the first to treat language-independent handling of C-preprocessed code at length. An empirical confirmation of the importance of the C preprocessor is [11]. The conditional completion problem is defined in [14], and a solution similar to ours appears in [29]. Garrido [13–15] treats refactoring C-preprocessed code in detail; [25] takes a different approach, requiring a semi-automated replacement of C preprocessor directives with a syntactically embedded macro language.

Finally, inasmuch as the present work addresses the topic of language-independent refactoring (and language-independent program transformation), one should note that there is a great body of literature in these areas, and commonality extends beyond just syntactic and lexical issues. For example, Lämmel [22] suggests that refactorings have both language-independent and language-dependent components and can be built by parameterizing a transformation with language specifics, while Mens *et al.* [26] investigate the viability of graph rewriting as a formalism for specifying refactorings.

7 Conclusions

We have proposed six *grammar annotations*—omission, labeling, Boolean access, list formation, inlining, and superclass formation—that allow an abstract syntax for a language to be defined based on its concrete syntax. A tool can use such an annotated

grammar to generate both a parser and a rewritable AST. *Concretizing* the AST allows it to preserve the formatting of the original code even after rewriting. Furthermore, an AST generator based on our method can couple the generated parser/AST-builder with a *pseudo-preprocessor* to allow representation and manipulation of preprocessed code. Our AST generator has been implemented in a tool called Ludwig [24] and has been used to generate the rewritable AST in a refactoring tool for Fortran; the annotated grammar was more than an order of magnitude smaller than the generated code, and the overhead of concretizing ASTs was very reasonable.

Much future work is possible. The present authors are working on a complete description and formalization of the AST generation algorithm: It is easy to develop an annotated grammar that does not "make sense" (for example, if a node indirectly inlines itself, or if a field has an ambiguous type, or if a node is simultaneously declared as a superclass and a list), so an AST generator must be able to detect errors and supply the user with an informative message rather than failing or generating invalid code. Grammars for mainstream programming languages other than Fortran should be developed in order to better assess our choice of annotations and the overhead of concretization. Constructing multiple-configuration ASTs efficiently using recursive descent parsers remains an open problem. To our knowledge, no one has addressed refactoring in the presence of other preprocessors (e.g., `m5`) in detail; it is intriguing to consider the possibility that a programmer could chain together an arbitrary sequence of pseudo-preprocessors for refactoring, just as one would chain together several tools prior to compilation in a Makefile. Most importantly, very few programming languages currently have production-grade refactoring tools available; we hope that our work will enable researchers and practitioners to implement and investigate refactorings for a variety of languages, allowing programmers using those languages to see the same productivity gains that others already have.

This work was supported by the United States Department of Energy under Contract No. DE-FG02-06ER25752 and by the National Science Foundation under NSF award #0725070. The authors would like to thank the anonymous referees and the members of the UIUC Software Engineering Seminar, particularly Darko Marinov and Danny Dig, for their detailed feedback.

References

1. ASC Center for Astrophysical Thermonuclear Flashes. http://www.flash.uchicago.edu
2. Bečička, J., Hřebejk, P., Zajac, P. (Sun Microsystems): Using Java 6 compiler as a refactoring and an anaysis engine. 1st Workshop on Refactoring Tools (2007)
3. Bowdidge, R. (Apple Computer): Personal communication.
4. Celentano, A: Incremental LR parsers. Acta Inf. 10, 307–321 (1978)
5. Cordy, J.: The TXL source transformation language. Sci. Comp. Prog. 61, 190–210 (2006)
6. Cormen, T., Leiserson, C., Rivest, R., Stein, C.: Introduction to Algorithms. 2/e. MIT Press (2001)
7. De Jonge, M., Visser, E., Visser, J.: XT: A bundle of program transformation tools. Proc. Lang. Descriptions, Tools and Applications 2001. Elec. Notes in Theoretical Comp. Sci. 44, 211–218 (2001)
8. DeRemer, F.: Practical translators for LR(k) languages. PhD thesis, MIT (1969)

9. Eclipse C/C++ Development Tools. http://www.eclipse.org/cdt
10. Eclipse Java Development Tools. http://www.eclipse.org/jdt
11. Ernst, M., Badros, G., Notkin, D.: An empirical analysis of C preprocessor use. IEEE Trans. on Software Eng. 28, 1146–1170 (2002).
12. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design patterns: Elements of reusable object-oriented software. Addison-Wesley (1994)
13. Garrido, A., Johnson, R.: Challenges of refactoring C programs. Proc. Intl. Workshop on Principles of Software Evolution 6–14 (2002)
14. Garrido, A.: Program refactoring in the presence of preprocessor directives. PhD thesis, University of Illinois at Urbana-Champaign (2005)
15. Garrido, A., Johnson, R.: Refactoring C with conditional compilation. Proc. 18th IEEE Intl. Conf. on Automated Software Eng., 323–326 (2003)
16. Hopcroft, J., Motwani, R., Ulman, J.: Introduction to automata theory, languages, and computation. 2/e. Addison-Wesley (2001)
17. IBEAM. http://www.ibeam.org
18. ISO/IEC 9899:1999: Programming Languages – C.
19. ISO/IEC 14882:2003: Programming Languages – C++.
20. JJTree. https://javacc.dev.java.net/doc/JJTree.html
21. Kort, J., Lämmel, R.: Parse-tree annotations meet re-engineering concerns. Proc. Source Code Analysis and Manipulation 2003, 161–172.
22. Lämmel, R.: Towards generic refactoring. Proc. ACM SIGPLAN Workshop on Rule-Based Prog., 15–28 (2002)
23. LPG http://sourceforge.net/projects/lpg/
24. Ludwig. http://jeff.over.bz/software/ludwig
25. McCloskey, B., Brewer, E.: ASTEC: A new approach to refactoring C. Proc. 13th ACM SIGSOFT Intl. Symp. Found. Software Eng., 21–30 (2005)
26. Mens, T., Van Eetvelde, N., Demeyer, S., Janssens, D.: Formalizing refactorings with graph transformations. J. Software Maint. and Evolution 17, 247–276 (2005)
27. Parr, T.: The definitive ANTLR reference: Building domain-specific languages. Pragmatic Bookshelf (2007)
28. Photran. http://www.eclipse.org/photran
29. Platoff, M., Wagner, M., Camaratta, J.: An integrated program representation and toolkit for the maintenance of C programs. Proc. Conf. Software Maint., 129–137 (1991)
30. Sellink, M., Verhoef, C.: Scaffolding for software renovation. Proc. Conf. Software Maint. Reeng., 161–172 (2000)
31. Stallman, R., Weinberg, Z.: The C preprocessor. http://gcc.gnu.org/onlinedocs/cpp.pdf
32. Tomita, M.: Generalized LR parsing. Springer (1991)
33. Van Den Brand, M., Van Deursen, A., Heering, J., De Jong, H., De Jonge, M., Kuipers, T., Klint, P., Moonen, L., Olivier, P., Scheerder, J., Vinju, J., Visser, E., Visser, J.: The ASF+SDF Meta-Environment: a component-based language development environment. Compiler Construction 2001. LNCS 2027, 365–370 (2001)
34. Visser, E.: Stratego: A language for program transformation based on rewriting strategies. Rewriting Techniques and Applications. LNCS 2051, 357–361 (2001)
35. Vlissides, J.: Pattern hatching: Design patterns applied. Addison-Wesley (1998)
36. Wang, D., Appel, A., Korn, J., Serra, C.: The Zephyr abstract syntax description language. USENIX Workshop on Domain-Specific Langauges (1997)
37. Wile, D.: Abstract syntax from concrete syntax. Proc. 19th Intl. Conf. on Software Eng., 472–480 (1997)