Reinventing the Wheel Or Not Yet Another Compiler Compiler Compiler

Donald E. Yessick
Joel Jones
University of Alabama
Tuscaloosa, AL USA 35487-0290
dyessick@cs.ua.edu

Abstract- This paper discusses a tool developed for generating code generators. The first author, dismayed with the clumsiness of lex, yacc, JLex, and Java Cup, proposes a new tool that serves to augment the above tools, discusses challenges met during the first generation of said tool, and lays out plans for future revisions.

1 Introduction and Previous Work.

A typical compiler course introduces students to the compiler writing tools lex and yacc[5], or JLex[1] and Java Cup[3], if the professor is a Java aficionado. The professor organizes students into groups and challenges each group to write a semester project compiler for some small language subset, often using a stack-based language as the target language. Such a task, however, is little different from translating to a high level language. The first author further expounds our motivation for tool development from personal experience in this section.

1.1 The Tale of Tools.

As an undergraduate, I too received the compiler project assignment, but without the introduction of the compiler writing tools lex and yacc. Our assignment required us to hand write a compiler using recursive decent parsing methods and implement a language of our own design. I loved developing the grammar and sought to include all kinds of marvelous and innovative statements before unknown, but, alas, when it came time to build my recursive descent parser I balked—the task was just too simple, tedious and uninteresting. Looking at the grammar I could see the end product. Writing the code would be a repetitive and monotonous waste of time. And what if the grammar changed? Modifying the resulting parser would be harrowing to say the least.

I solved my problem by writing a program that built a recursive descent parser from the grammar. I included in the grammar details so my program would even generate the correct stack code and it had variables and all kinds of neat tricks. This project became fascinating and hard. Hand coding my parser would have taken a quarter of the time I spent writing the tool. But when the tool was complete I could change my grammar at will and generate new languages with every change. My original language grew and in the end was far richer and complex than any other project turned in that semester. I handed in my tool as part of my project as it was the only way I could explain the lack of documentation in my recursive descent parser. It was then that my instructor informed me of such wonders as lex and yacc, and I seethed, realizing I had just reinvented the wheel, and wondered why had we not been instructed to use these wonderful compiler generating tools.

Some ten years later, in a graduate compiler course, I had my first excuse to use lex and yacc. After a short introduction to the tools, the professor commanded us to form groups and develop a compiler for a small language. Generating the grammar was easy of course. Generating the code for the language appeared simple as well. Inexperience with lex and yacc led to a general breakdown of our whole process. Although our learning experience was of immeasurable value, our compiler proved woefully inadequate. During the project we spent a short time writing our code and needed the rest of eternity to debug the resulting compiler and of course time ran out. In retrospect we mishandled the debugging process since group dynamics led to a late start on the project and some poor decisions. Yet, we were competent programmers who wrestled with and fell to unfamiliar beasts.

A couple of years later, my research advisor taught the compiler class for his first time. Having mixed emotions about my previous compiler experiences, I signed up for the class as an independent study to get a new spin on the subject. This time the compiler tools introduced were JLex and Java Cup, the Java equivalents of lex and yacc. The essential distinction between yacc and Java Cup being that variables in Java Cup can be assigned real names, otherwise my experiences from lex and yacc applied well. And names are good things.

A funny thing happened. This time the course focused more heavily upon the concept of abstract syntax trees and I remembered that old program that wrote a recursive descent

parser and I made a connection. This time when the group assignment was made I went to work not on my grammar, but on my tool. We had six weeks to finish our project. I finished my tool in three. I used JLex and Java Cup to build it. The project was completed two days later. What in the world would I tell my group?

1.2 "It ain't broke!" they cried.

I do not pretend to have done anything novel or marvelous. As my advisor told me when I confessed a predilection for the subjects of lexical analysis (lexing) and parsing grammars, "Lexing and parsing are solved problems." If these problems are solved, I thought, then why are the tools so difficult to use? Surely there is room for improvement.

2 Not Rocket Science.

The key observation is that tools that identify the syntactic structure, while extremely useful, do not completely solve the language translation problem. Our contribution is to provide mechanisms to aid in the expression of the transformation from source to target language. Anyone who has constructed a syntax tree recognizes the relationship of the data structure to the grammar. The tedium of coding the data structure is well known—a rote process—just follow the grammar. After the data structures are created you return to the grammar and integrate code for building the tree. Our tool automates this process.

Which came first: the grammar or the tree? Using traditional compiler writing tools requires the development of the grammar before implementing the syntax tree, yet the syntax tree must be clearly defined to properly decorate a grammar. Lexical analysis tools generally require writing the parser first, which by definition depends heavily on the lexer. This tight interdependence of components is not how most programming projects should be designed and certainly a compiler is nothing more than a bulky program. Why should writing a compiler be different than other large programming task? The answer, of course, is that it is not different, if you have the right tools.

3 A Brief Example.

The tool hides complexity, allowing the designer to concentrate on the transformation, rather than the design and construction of the syntax tree and awkward print commands to produce target code. This section clarifies the tool's abilities, demonstrating how one might go about writing a C to Ada function translation.

Consider the transformation of a simple C function into an equivalent Ada function. We want the C function:

```
int foo ( int j )
{
    return j + 1;
}
```

to become the Ada function:

```
function foo ( j : in integer )
return integer is
begin
        return j + 1;
end foo;
```

The similarities between these languages are straightforward and one might think the transformation trivial, but expressing the transformation process in JLex and Java Cup can be both difficult and ugly.

Our tool could describe the transformation with the following:

```
cfun::=rtype fid parmlist lbracket body
       rbracket
    ::={:
        function @fid@ @parmlist.gCode()@
                return @rtype.gCode()@ is
        begin
                @body.gCode()@
        end @fid@;
    :};
```

Certainly this has elements which are not beautiful, yet we contend that it is easier to produce and cleaner than equivalent code expressed in yacc or Java Cup. Even without further explanation the fundamental concepts are discernible. Further explanation is provided below.

First we have the grammar rule:

```
cfun::=rtype fid parmlist lbracket body
       rbracket
```

In English you might read this as: A C function is a return type (rtype), followed by a function identifier (fid), followed by a parameter list (parmlist), then followed by a left bracket ("{"), a body of C statements, and finally a closing right bracket ("}").

Obviously this is not the complete grammar and similar rules must be defined for the nonterminals: rtype, fid, parmlist, and body. Our goal is not to explain the mechanisms of a grammar but to illuminate the differences and advantages of our tool over yacc and Java Cup. The first difference is removal of the need to supply names to grammar elements[1]. In Java Cup the same grammar, to be useful, would be expressed:

```
cfun::=rtype:a fid:b parmlist:c lbracket
       body:d rbracket
```

[1] To handle duplicate values on the right-hand side, an extra rule must be inserted.

For purposes of brevity, unimaginative names are shown here, but it should be clear that the addition of names does not enhance the readability of the grammar. Furthermore, if more imaginative names were used, readability would likely deteriorate. Yacc, on the other hand, automatically generates variable names. However, these names are only integers referring to the position of the symbol in the production rules. Simple modifications to the rules can introduce errors.

The second major part of our C function to Ada function translator, defines the transformation process. It is shown again here:

```
function @fid@ @parmlist.gCode()@
    return @rtype.gCode()@ is
begin
    @body.gCode()@
end function @fid@;
```

Generally speaking, this is text that will be copied directly to the output file. The @'s are used to identify areas where substitution needs to occur. @fid@ for example, leads to the substitution of the function id of the source C program. In this case the function id would be copied unchanged. @parmlist.gCode()@ informs the tool that the C parameter list needs to undergo a transformation (defined by the function gCode) before it is printed. rtype also must go through a transformation as well. A simple rule for rtype would be:

```
rtype::=int::={:integer:};
```

Other types would need to be defined, but this should paint the general picture.

This leaves questions about the semantics of gCode. We have already witnessed an example. We provided the rule cfun that must form part of a larger grammar. In another rule, where the nonterminal cfun formed part of the production, we can use the transformation just examined by coding @cfun.gCode()@ when defining the transformation. If we wanted the to show cfun as it appeared in the source file, we would simply put @cfun@, although white space is not perfectly preserved.

4 Other Features.

While the transformation function is perhaps the most visibly beneficial feature, the tool also allows definition of the lexer within the same file, simplifying the compiler building process. Also, the tool provides shortcuts for constructing lists and delimited lists.

In a typical yacc or Java Cup program, a list would be indicated thusly:

```
List ::= ListItem
       |  List ListItem;
```

The tool allows the equivalent list construction with:

```
List ::= ListItem list;
```

A comma delimited list might be described in yacc as:

```
Dlist ::= ListItem
        |  Dlist T_Comma ListItem
```

Which we express:

```
Dlist::=ListItem list T_Comma;
```

Of course, the tool produces code for processing these lists and generating transformations.

5 Can it Really Save Time?

As mentioned earlier, the tool was originally conceived to aid in developing a project in a compiler course. The project was to develop a little language for transferring relational database files into CSV files (comma separated values) and vice versa. While leeway was provided as to whether to generate code or execute as an interpreter, the instructions required the use of JLex and Java Cup for the lexical analysis and parsing of the little language. The new tool generates both the JLex .l file as well as the Java Cup .y file. In addition it builds a Java source file for representing the syntax tree. A 559 line .sly file (.sly since it produces the syntax tree, the .l, and the .y files) when fed to the tool generated a 1729 line Java file for the syntax tree, a 62 line JLex .l file, as well as a 440 line Java Cup .y file. The tool itself was composed of an additional 1606 lines of code in Java, .l and .y combined. It is estimated that around 625 lines of complicated code were eliminated from the coding and debugging process, even after considering the cost of developing the tool first. Additionally, enhancements to the little language were made on the fly and without fear after the tool's construction.

6 Calculator Example.

A common textbook example of the use of a parser/lexer is a simple calculator. The task is to read in a series of numeric expressions and produce as output the resulting values. Here is the solution for the tool (for brevity gCode has been changed to g):

```
1.  T_Semi::={:";":};
2.  T_Plus::={:"+":};
3.  T_Minus::={:"-":};
4.  T_Mult::={:"*":};
5.  T_Div::={:"/":};
6.  T_Mod::={:"%":};
7.  T_Exp::={:"^":};
8.  T_Open::={:"(":};
9.  T_Close::={:")":};
10. T_Num::={:{DIGIT}*:};
11. abstract::={:".":};
```

```
12. abstract::={:[\n]:};
13. Grammar ::= Statement list;
14. Statement::=Expr T_Semi
15. ::={:@expr@    =    @expr.g()@
16. :};
17. abstract Expr
18.   Expr1::=Expr + Fact
19.   ::={:@(INT(expr.g())
            +INT(fact.g()))@:};
20.   Expr2::=Expr - Fact
21    ::={:@(INT(expr.g())
            -INT(fact.g()))@:};
22.   Expr3::=Fact
23.   ::={:@INT(fact.g())@:};
24. ;
25. abstract Fact
26.   Fact1::=Fact * Exp
27.   ::={:@(INT(fact.g())
            *INT(exp.g()))@:};
28.   Fact2::=Fact / Exp
29.   ::={:@(INT(fact.g())
            /INT(exp.g()))@:};
30.   Fact3::=Fact % Exp
31.   ::={:@(INT(fact.g())
            %INT(exp.g()))@:};
32.   Fact4::=Exp
33.   ::={:@INT(exp.g())@:};
34. ;
35. abstract Exp
36.   Exp1::=Term ^ Exp
37.   ::={:@pow(INT(term.g()),
                INT(exp.g()))@:};
38.   Exp2
39.   ::=Term::={:@INT(term.g())@:};
40. ;
41. abstract Term
42.   Term1::=T_Num
43.   ::={:@t_num@:};
44.   Term2::=(Expr)
45.   ::={:@INT(expr.g())@:};
46. ;
```

Below is an example of the output produced by the resulting compiler (since the compiler echoed the input it has been omitted):

```
1 + 2 * ( 3 + 4 ) * 5    =    71
1 + 2 ^ 3 + 4 ^ 5    =    1033
( 1 + 2 ) ^ 3 + 4 ^ 5    =    1051
2 ^ 3 ^ 2    =    512
1 - 2 + 3 / 4 + 5    =    4
( 2 ^ 3 - 2 ) / 4    =    1
```

Now let us run through this example and see how it works.

On startup the tool adds code to open the output files and begin the code generation process from the abstract syntax tree. The tool generates a compiler that builds the entire syntax tree before entering code generation rather than generating code while parsing.

Lines 1-12 are used to construct the lexer. Certain macros are predefined such as DIGIT since they occur frequently enough to warrant inclusion. Lines 11-12 describe lexing items which do not return tokens.

Line 13 begins the grammar and one can see that the calculator language consist of a list of statements. The tool created the following in the Java Cup .y file:

```
GRAMMAR::=STATEMENT_LIST:list
  {:RESULT=new Grammar(list);:};
STATEMENT_LIST::=STATEMENT:statement
  {:RESULT=
    new Statement_List(statement,null);:}
|STATEMENT:statement STATEMENT_LIST:list
  {:RESULT=
    new Statement_List(statement,list);:};
```

Note that right recursion is used rather than left recursion. Reversing this to left recursion should not be difficult at some later development stage if that need arises using the keyword leftlist.

Line 14 describes defines a statement as an expression followed by a semicolon, while lines 15-16 describe how the statement line should be translated. @expr@ instructs the tool to echo the input—that is print the expression unchanged to the target file. “ = ” instructs the tool to print “\t=\t” to the target file and @expr.g()@ instructs the tool to print the translated (or in this case calculated) value in the target file. Line 16 is on its own line to cause a line break after printing the result. All white space is captured as entered in the transformation functions.

Lines 17-24 start the familiar grammar for an expression without using yacc or Java Cup’s precedence operators. It is uncertain the omission of precedence declarations will prove a defect although they would be simple enough to add to the tool’s features. While line 17 begins the “familiar grammar” it has a rather odd presentation here. Line 17 begins an abstract rule. Selection is implemented in our tool through an abstract rule. Our tool restricts selections—they must be named. Related selections are contained within an abstract rule. Lines 18-23 define 3 subrules under Expr. Notice the use of “+” and “-” in the first two subrules instead of T_Plus and T_Minus as might expect in a yacc or Java Cup file. Since “+” and “-” do not interfere with a grammar definition these were added as abbreviations in the tool. Other punctuation symbols with the exception of “;”, which is needed to terminate each production, have also been added as abbreviations and generally improve the readability. In the .y file, lines 17–24 expand to

```
EXPR::=EXPR1:abstractval
     {:RESULT=abstractval;:}
|  EXPR2: abstractval
     {:RESULT=abstractval;:}
```

```
|  EXPR3: abstractval
     {:RESULT=abstractval;:}
;
EXPR1::=EXPR:expr PLUS:t_plus FACT:fact
     {:RESULT=
       new Expr1(expr, t_plus, fact);:}
;
EXPR2::=EXPR:expr MINUS:t_minus FACT:fact
     {:RESULT=
       new Expr2(expr, t_minus, fact);:}
;
EXPR3::=FACT:fact
     {:RESULT=new Expr3(fact);:}
;
```

Which is equivalent to the textbooks: E::= E+F | E-F | F, but is spread out here for two reasons. Primarily this follows the design of the syntax tree. Each option for expression represents a different type of node. In generating the syntax tree classes, the tool defines abstract class Expr and classes Expr1, Expr2, and Expr3 extend this base class. Secondly, this forces rules to fall into two categories. A rule is either a choice of single nonterminal expressions (R::=A | B | C | . . . | Z) or a rule is a unique combination of terminals and nonterminals (R::=A B C . . . Z). The forced naming has another benefit, which was heavily exploited during the first project—it promotes reuse since all subrules are named and may be used freely in other productions.

We also note here that it is permissible for abstract rules to have abstract subrules. The following construct is acceptable:

```
abstract A
    A1::=B C D E;
    abstract A2
        A2a::=B C D;
        abstract A2b
            A2b1::=B C;
            A2b2::=B D;
        ;
        A2c::=C D;
    ;
    A3::=C D E;
;
```

Unless there is a need to separate productions A2 and A2b (perhaps needed in other productions) this is equivalent to:

```
abstract A
    A1::=B C D E;
    A2a::=B C D;
    A2b1::=B C;
    A2b2::=B D;
    A2c::=C D;
    A3::=C D E;
;
```

Even if A2 and A2b are to be used elsewhere in the grammar there is a better way to write the productions although this creates two extra productions (and classes).

```
abstract A
    A1::=B C D E;
    A2_abstract::=A2
    A3::=C D E;
;
abstract A2
    A2a::=B C D;
    A2b_abstract::=A2b
    A2c::=C D;
;
abstract A2b
    A2b1::=B C;
    A2b2::=B D;
;
```

Lines 19, 21, and 23 are used in generating the gCode functions in the syntax trees for Expr1, Expr2 and Expr3 respectively. The INT function has been added to the base class of the syntax tree for simplicity. It simply returns the int value of a string or syntax tree node and INT(Expr) is clearer than either of: `Integer.parseInt(Expr.toString());` or `Integer.parseInt(Expr.gCode());`. Here is the syntax tree generated for Expr1:

```
class Expr1 extends Expr{
    Expr expr;
    Terminal t_plus;
    Fact fact;
    Expr1(Expr _expr,
        Terminal _t_plus,
         Fact _fact){
         expr=_expr;
         t_plus=_t_plus;
         fact=_fact;
    }
    public String toString(){
        return ""+expr+WS+t_plus+WS+
               fact;
    }
    public String gCode(){
        return ""+(INT(expr.gCode())+
    INT(fact.gCode()))+"";
    }
}
```

For each production the tool automatically generates a class with the terminals and nonterminals as instance variables. All the syntax tree classes derive from a common syntax tree base class where INT is defined. Several constants (e.g., WS, TAB, NEWLINE) have also been defined in the

syntax tree base class. These were added to promote readability during the tool's initial development, but subsequently have seen less use.

The rest of the grammar follows similarly from Lines 19-24. The only interesting part left is how exponents (Exp, lines 35-40) were handled to properly handle precedence from right to left.

7 Final Example.

We give one final example to demonstrate the unsightly programming avoided using the tool. As mentioned earlier, the first project the tool was used for created a little language to aid the transfer of data between a database and a CSV file system. The language produced Java code as its target. Below is a fragment taken from the grammar used to produce the little language:

```
TransportStatement7::=
  FromClause ToClause WhereClause
  SelectClause InitClause OnNextClause
    ::={:
        @fromclause.gCode()@
        @toclause.gCode()@
        @initclause.gCode()@
        while (source.scan()){
            @whereclause.gCode()@
            {
                @selectclause.gCode()@
                target.writeRow();
                @onnextclause.gCode()@
            }
        }
        target.close();
    :};
```

This statement when processed by the tool produced the following code for the syntax tree:

```
class TransportStatement7
extends TransportStatement{
    FromClause fromclause;
    ToClause toclause;
    WhereClause whereclause;
    SelectClause selectclause;
    InitClause initclause;
    OnNextClause onnextclause;
    TransportStatement7(
            FromClause _fromclause,
            ToClause _toclause,
            WhereClause _whereclause,
            SelectClause _selectclause,
            InitClause _initclause,
            OnNextClause _onnextclause
    ) {
        fromclause=_fromclause;
        toclause=_toclause;
        whereclause=_whereclause;
        selectclause=_selectclause;
        initclause=_initclause;
        onnextclause=_onnextclause;
    }
    public String toString(){
        return ""+fromclause+WS+
               toclause+WS+
               whereclause+WS+
               selectclause+WS+
               initclause+WS+
               onnextclause;
    }
    public String gCode(){
        return "\n"+"\t\t\t"+
        fromclause.gCode()+
        "\n"+"\t\t\t"+
        toclause.gCode()+
        "\n"+"\t\t\t"+
        initclause.gCode()+
        "\n"+"\t\t\t"+
        selectclause.gCode2()+
        "\n"+
        "\t\t\twhile (source.scan()){\n"+
        "\t\t\t\t"+whereclause.gCode()+
        "\n"+"\t\t\t\t{\n"+
        "\t\t\t\t\t"+selectclause.gCode()+
        "\n"+
        "\t\t\t\t\ttarget.writeRow();\n"+
        "\t\t\t\t\t"+onnextclause.gCode()+
        "\n"+
        "\t\t\t\t}\n"+
        "\t\t\t}\n"+
        "\t\t\ttarget.close();\n"+
        "\t";
    }
}
```

While this code is not perfectly formatted, the automatic pretty printer provided by capturing and copying white space to the output aids readability. Here's an example of the source code accepted by the finished compiler:

```
transport from table assignments
 to file assignments.csv
    where i % 2 == 0
    values i, .class, column 2,
           .descript, column 5, .maxscore
    init {:
        int  i=0;
    :}
    onnext{:
        i=i+1;
    :};
```

And the target code produced by the finished compiler:

```
source =
    new tableSource(jdbc,"assignments",null);
target = new fileTarget("assignments.csv");
int  i=0;
while (source.scan()){
     if ( i % 2 == 0){
        target.writeField( i);
        target.writeField(
   source.getColumnData("class"));
        target.writeField(
   source.getColumnData(2));
        target.writeField(
   source.getColumnData("descript"));
        target.writeField(
   source.getColumnData(5));
        target.writeField(
   source.getColumnData("maxscore"));
        target.writeRow();
        i=i+1;
    }
}
target.close();
```

The rest is left to imagination.

8 Related Work

There is a long history of tools for generating lexers and parsers. As mentioned earlier, the most well-known are lex and yacc and their GNU counterparts, flex and bison [5]. The lex lexical analysis generator produces recognizers for regular expressions and the yacc parser generator produces recognizers for context-free grammars. As described above, the implementation of our tool makes use of their Java counterparts, JLex[1] and Java Cup[3]. The most closely related tool to ours is TXL[4]. It is specifically targeted to translating one language to another but is suited to general purpose tasks.

XML and XSLT are also related to the current tool[2]. However, these technologies solve much smaller problems. Our tool can parse XML, but can also parse any other context-free language. Furthermore, our tool can be used to generate XML documents from any document which is in a context-free form and unlike XSLT does not require the presence of XML tags. We have already used the tool to generate XML and HTML files from a structured document.

9 Status, Future Work, and Conclusions.

Bootstrapping has uncovered many of the tool's original limitations. The principal limitation was the inability to generate a compiler that generates multiple files. The project the tool originally was applied to required only a single output file. To bootstrap, the tool had to create a compiler capable of generating at least three output files and should be able to produce any number of files. Coupled with this was a limit of two transformation functions per grammar rule (gCode, and gCode2 which has not been shown here). There also appeared during the bootstrapping process a need to handle list definitions so that lists could be traversed from distant areas of the syntax tree. All of these issues have been overcome and have led to improvements in the lexical analysis portion of the tool as well as refinements in the grammar productions.

Another common theme in language design is optional clauses. An optional language clause should be a simple and concise definition. As an example: `R::= A B C | A C;` might be more concisely expressed as `R::= A [B] C;` without losing meaning. This power is currently being tested in the bootstrap version (now officially named SLY). Transformation functions must be concisely and accurately defined in the presence or absence of optional symbols. This is being accomplished by creating Null Objects[6] for the missing symbols. A Null Object is used to represent the missing symbol and has the advantage that it can be defined to do nothing or to provide a default value. Once the new features are in place the size of the bootstrap code should shrink considerably as the current implementation requires similar rules to be duplicated to handle small nuances. [2] The current version of the bootstrap is 1365 lines but generates over 5000 lines when compiled. Two of the output files also generate Java code using Java Cup and JLex, so the final number of Java lines generated is over 8400 lines.

Up to this point many semantic features of compiler generation have largely been temporarily ignored while syntactic and code generation techniques have been the focus. Registering names and identifiers for type analysis is one simple yet important example of where semantic checking can be built into the tool.

To enhance readability of the tool source we will produce an HTML version where grammar symbols will form links to governing rules and the grammar will be color-coded.

To enhance testing the tool will produce an undecorated version of the grammar to allow testing of the grammar only. Allowing the developer to separate testing of the transformation code and the parsing should speed development. This will allow testing of the parser even when the transformation functions contain errors.

Many programming tasks can be solved using programming language technology. We have shown here that those

[2] As an example, we saw earlier the production TransportStatement7. This was one of eight productions, each describing one variant of a general transport statement. Using the optional clause with Null Objects collapses the eight productions into a single equivalent production with the form:

```
TransportStatement::=FromClause ToClause
  [WhereClause] [SelectClause] [InitClause]
  [OnNextClause]
```

The transformation function of this rule is exactly identical to the one seen earlier.

tasks include implementing programming languages and that sly tools can ease the process.

References

[1] Elliot Berk. JLex: A lexical analyzer generator for Java. http://www.cs.princeton.edu/~appel/modern/Java/JLex/, 2000.

[2] James Clark. XSL transformations (XSLT) version 1.0. http://www.w3.org/TR/xslt, 1999.

[3] Scott Hudson. Java cup LALR parser generator for Java. http://www.cs.princeton.edu/~appel/modern/Java/CUP/, 1999.

[4] TXL Software Research Inc. The TXL transformation system. http://www.txl.ca/, 2001.

[5] John Levine, Tony Mason, and Doug Brown. *lex & yacc*. O'Reilly, 2nd edition, 1992.

[6] Bobby Woolf. Null object. In Robert Martin, Dirk Riehle, and Frank Buschman, editors, *Pattern Languages of Program Design 3*, pages 5–18. Addison-Wesley, 1998.