Improving Programming Language Transformation

Steven A. O'Hara
Eagle Legacy Modernization, LLC, Austin, Texas, USA

Abstract - *Modernizing legacy computer programs is challenging. This paper introduces a generalized framework for language transformation with three main elements. First, target languages are shielded from the transformation process by a collection of interfaces such as "create an if statement." Transforming from Delphi is the same whether the target is Python, Java, C# or some other language that implements the interfaces. Second, the framework ensures synchronization between source language grammars and transformation tools, so changes to a grammar cannot be made without adjusting the impacted tools. This allows large scale transformation projects where both the grammars and the tools are under concurrent development. Third, source code generation is accomplished by adding output formatting annotations to the target language grammar.*

Keywords: Legacy modernization, programming language transformation.

1 Introduction

By some estimates, the global number of computer programming languages is about 565 [1], or about 1,500 including variations [2]. Sources that list the most popular programming languages, such as [3], do not include the most pervasive business programming language, COBOL.

Many of these programming languages have been in use for many decades, such as COBOL, Fortran, Natural, RPG, PL/I and others. They suffer from a shortage of skilled software developers, and many applications were built without internet security in mind, potentially exposing vulnerabilities.

New languages are introduced on a regular basis and existing languages continue to evolve, but legacy languages rarely become extinct. This proliferation of languages and their variations is not handled well by existing tools, such as [4] that tend to focus on a few specific language pairs, such as COBOL to Java [5].

"There are still hundreds of billions of lines of COBOL code in use today by banks, insurance companies and other organizations, and COBOL is still used somewhere in a large proportion of all business transactions." [6]

"Some 23 of the world's top 25 retailers, 92 of the top 100 banks, and the 10 largest insurers all entrust core operations to Cobol programs running on IBM mainframes." [7]

This paper deals with three issues, and introduces a new framework to facilitate more efficient, effective software modernization. The first issue is choosing a target language. We recognize that transformation does not have to utilize all the features of each target language. In fact, a relatively small set of features must be provided, such as "add a method to a class," and that these features are generally shared by the target languages.

When considering legacy language transformation, the choice of target programming languages typically includes Java, C#, Python and a few others. Transformation into legacy languages like COBOL is specifically not a part of this effort.

There are many efforts to introduce a common intermediate language definition, such as [8], to span multiple source languages. This paper only considers an intermediate interface for target languages.

There are also many point-to-point translation systems. Stratego/XT [9], for example, supports strategic term rewriting to perform transformations. Likewise, RascalMPL [10] has tools for term rewriting. Our contention is that the expressiveness of rules is well-suited for small transformation projects, but does not scale up to thousands of programs in dozens of languages. To achieve that scale, we believe that the transformation logic is best expressed in a classic programming language like Java.

The second issue is synchronization. In our experience with legacy modernization, grammars are continually being enhanced. Rarely are perfect grammars available that encompass all existing source code. It is normal that small adjustments have to be made to accommodate certain language patterns. By utilizing the work of [11], we are able to represent source grammars in Java code, rather than text files containing Backus-Naur Form (BNF) grammars or similar. Likewise, the transformation tools are also written in Java, within the same environment. With this approach, changes to the grammar will be detected directly by the compiler or, more commonly, the Integrated Development Environment (IDE).

The third issue is target language generation. After creating an in-memory representation of the new program, it must be converted to a text file. In our framework, we add Java annotations[1] (such as @NOSPACE, @INDENT, etc.) to the target Program Grammar to control word and line spacing, indentation, etc. Other systems, such as Ekeko/X [12] also use annotations, but we use them mainly for output generation, not for transformation logic.

Our experimental results involve three sets of language transformation pairs, all using the same framework.

- BNF to a Program Grammar (in Java)
- COBOL to Python
- Delphi to C#

No claim is being made that the transformation process is complete for any of these. Transformation is a complex process and this paper discusses many of those difficulties, and how to help alleviate some of the burden.

1. Transformation Framework

A traditional transformation process is described in detail in [13]. Here are the steps for our process:

- Using a Program Grammar, parse the source program, producing a Program Semantic Tree (PST). A PST is similar to an Abstract Syntax Tree (AST), but includes some semantic information about each token, such as a two-way cross reference between variable definitions and their references.
- Read the source program PST and generate a second PST in the target language using transformation logic, such as Sections 2.1, 2.2, and 2.2.3. This process is only aware of the source language; all references to the target language are hidden through interfaces.
- Using a Program Grammar for the target language, read the new PST and generate a source file.

The Parser and the Program Generator are language-independent. The Transformation Engine depends only on the source language, not the target language.

The main benefit of our framework is that the transformation process does not have access to the target language, just the interface layer. There is also just one grammar for each language, whether used for source parsing or target generation.

1.1. Source Program Grammars

Figure 1 shows a sample Program Grammar for the "IF" statement in COBOL. Additional details are available in [11].

The fields in a Token Sequence must appear in order, although fields may be optional, as denoted by the @OPT annotation. Token List means zero or more occurrences. The Program Grammar in Figure 1 is equivalent to the BNF version shown in Figure 2.

We have used these Program Grammars to parse many thousands of programs, and millions of lines, written in COBOL, RPG, Natural, PL/I, C, Java, C#, Python, JavaScript, HTML, CSS and others. All Program Grammars will be open-sourced so they can be used and managed by the community (see Section 4.1).

```
public class COBOL_IfStatement extends TokenSequence
{
  public COBOL_Keyword IF = new COBOL_Keyword("IF");
  public COBOL_Expression condition;
  public @OPT COBOL_Keyword THEN = new COBOL_Keyword("THEN");
  public TokenList<COBOL_Statement> thenActions;
  public @OPT COBOL_Else elseClause;
  public @OPT COBOL_Keyword ENDIF = new COBOL_Keyword("END-IF");

  public static class COBOL_Else extends TokenSequence
  {
    public COBOL_Keyword ELSE = new COBOL_Keyword("ELSE");
    public TokenList<COBOL_Statement> elseActions;
  }
};
```

Figure 1. COBOL "IF" Statement Program Grammar

```
  COBOL_IfStatement ::= "IF" condition ["THEN"]
    thenActions [elseClause] ["ENDIF"];
  condition ::= COBOL_Expression;
  thenActions ::= COBOL_Statement*;
  elseClause ::= "ELSE" elseActions;
  elseActions ::= COBOL_Statement*;
```

Figure 2. COBOL "IF" Statement in BNF

1.2. Transformation Logic

Figure 3 shows the logic used to transform one variation of the PERFORM verb in COBOL. The logic is specific to COBOL, but independent of the variations of COBOL (e.g., some versions use fixed columns for labels and comments; some are free-format).

```
if (token instanceof COBOL_PerformVarying)
{
  COBOL_PerformVarying varying = (COBOL_PerformVarying) token;

  // Collect all the inline statements
  ArrayList<Stmt> statements = new ArrayList<Stmt>();
  for (COBOL_Statement statement : inline.statements)
  {
    AbstractToken oldStatement = statement.getWhich();
    Stmt newStatement = trans.transformStatement(oldStatement);
    statements.add(newStatement);
  }
  Stmt action = trans._target.createStatementBlock(statements);

  String loopVar = trans.getFullVariableName(varying.id);
  Expr initVal = trans.transformExpression(varying.from);
  Expr incrVal = trans.transformExpression(varying.by);
  Expr until = trans.transformExpression(varying.until.cond);
  Expr term = trans._target._createExpression.createNot(until);

  Stmt forStatement = trans._target.createForStatement(loopVar,
      initVal, term, action, incrVal, varying);
  return forStatement;
}
```

Figure 3. Transforming "PERFORM" from COBOL

[1] Java annotations are a form of syntactic metadata that can be added to source code.

The `Expr` (and `Stmt`) terms are Java generic data types[2] that represent `Java_Expression`, `CSharp_Expression`, etc. depending on the target language. The `trans` variable is specific to COBOL, while `trans._target` is specific to the target interface layer.

1.3. Target Interface Layer

For a language to be a transformation target in this framework, there must be classes set up that implement the following six interfaces[3]: Transform Target (the main class), Create Program, Create Class, Create Method, Create Statement, and Create Expression.

The total count of methods that must be implemented for each language is approximately sixty, as of this writing. Although every method must be implemented for each target language, the implementations are often very similar.

```
public interface Create_Class<Cls extends AbstractClass,
    Meth extends AbstractMethod, Stmt extends AbstractStatement>
{
  public enum CLASS_QUALIFIERS { NONE, OVERRIDES }
  public AbstractComment addClassComment(Cls cls, String comm);
  public void addClassData(Cls cls, Stmt dataStmt);
  public void addMethod(Cls cls, Meth method);
  public void addConstructor(Cls cls, String className,
      AbstractExpression[] args);
  public Cls addInnerClass(PRIVACY privacy, Cls cls,
      String className, CLASS_QUALIFIERS qual);
  public void setClassExtends(Cls cls, String extendsClass);
  public Cls addInnerDataClass(Cls cls, String name, TYPES typ);
}
```

Figure 4. Target Interface for Create Class

The interface methods shown in Figure 4 (slightly condensed) must be implemented for each target language. The use of Java generics on the first two lines are crucial for this framework, because they retain the strong typing needed for scalability. With them, the Java target transformation code can be strongly typed, which avoids risky type-casts.

1.4. Target Program Grammars

The source Program Grammars and target Program Grammars are actually the same grammar, just used in different ways. The only significant difference is that the formatting annotations are only used when generating program output.

In Figure 5, the Token Chooser indicates that any one field or class instance can be used to satisfy that grammar rule. In this example, a statement could be just a semicolon (with a `@CURIOUS` warning), some data, an inner class, an enumeration, a statement block in braces, or one of the specified C# statements.

```
public class CSharp_Statement extends TokenChooser
{
  public @CURIOUS("Extra semicolon") PunctuationSemicolon semi;

  public CSharp_Data data;
  public CSharp_Class myclass;
  public CSharp_Enum enumeration;

  public static class CSharp_StmtBlock extends TokenSequence
  {
    public @INDENT PunctuationLeftBrace leftBrace;
    public @OPT TokenList<CSharp_Statement> statements;
    public @OUTDENT PunctuationRightBrace rightBrace;
  }

  public CSharp_BreakStatement breakStatement;
  public CSharp_ContinueStatement continueStatement;
  public CSharp_CheckedStatement checkedStatement;
  (etc. for each statement type)
```

Figure 5. C# Statement Program Grammar

The use of `@INDENT` means to indent after rendering this token while `@OUTDENT` means to un-indent before rendering this token. These annotations help make the output program file more readable. Additional annotations include `@NOSPACE` to prevent spaces before a token, and `@NEWLINE` to force a new line before a token.

1.5. Macro Pre-Processing

Many programming languages support macros and/or include files. Although compilers require access to all of the included files, our system continues processing even when some of those include files are not available.

In C/C++ this is exemplified by the `#define` macro capability. Our system can pre-process the source file(s) and expand macros. Because macros often depend on environment settings, macro expansion also allows project-specific controls to decide which macros to expand and what initial environment settings to use.

Other languages that utilize this pre-processor include PL/I, PHP, CMD (DOS), Delphi, and COBOL. The COBOL copy books are especially interesting with the REPLACING clause.

1.6. Comments

Many transformation systems either omit comments before transformation, or require comments as part of the grammar, in every place where they are used. In our system, we take a hybrid approach. If the Program Grammar includes comments, they are kept in the in-memory representation of the program. Otherwise, comments are discarded with a brief notification.

2. Experiments

Three distinct transformations are detailed in this paper, showing a range of capabilities. The point of this discussion is to describe the framework, not to claim that transformation from languages like COBOL or Delphi is fully implemented.

[2] Generics extend Java's type system to allow a type or method to operate on objects of various types while providing compile-time type safety.

[3] An interface in Java is an abstract type that is used to specify a behavior that classes must implement.

2.1. BNF to Program Grammar

To jump-start a new Program Grammar, it is often useful to use an existing BNF-like grammar.

Program Grammars do not represent terminal nodes (such as strings, numbers, punctuation, comments, etc.) in the grammar itself, rather, they use Java code that can be shared with grammars for other programming languages. For example, string literals and numbers are generally similar between languages. Furthermore, implementing terminal nodes (consider the BNF rules for a floating-point number) in a traditional grammar is often complicated and error-prone.

The following example parses a BNF grammar, transforms it to a Program Grammar (in Java), and uses that generated Program Grammar to re-parse the original grammar.

2.1.1. Source EBNF Program

Extended BNF (EBNF) has several variations. In this case, the asterisk `*` means zero or more of the previous item, a plus sign `+` means one or more. The vertical bar `|` means to choose one, square brackets `[` and `]` mean the item is optional. The `::=` marker is used to define a rule, with semicolon `;` indicating the end of a rule.

```
ebnf-program ::= ebnf-rule*;
ebnf-rule ::= ebnf-identifier '::=' ebnf-alternation ';';
ebnf-alternation ::= ebnf-expression ( '|' ebnf-expression )*;
ebnf-expression ::= ( literal | ebnf-identifier [ebnf-modifier] |
    ebnf-optional | ebnf-group )+;
ebnf-group ::= '(' ebnf-alternation ')' [ebnf-modifier];
ebnf-optional ::= '[' ebnf-alternation ']' [ebnf-modifier];
ebnf-modifier ::= '+' | '*';
```

Figure 6. Source EBNF Grammar for EBNF

The two terminal nodes (`literal` and `ebnf-identifier`) in Figure 6 are defined in Java, and are outside the scope of this document.

```
public static class BNF_ExpressionTerm extends TokenChooser
{
  public BNF_Literal literal;

  public static class BNF_Rulename extends TokenSequence
  {
    public BNF_Rule_Reference ref;
    public @NOSPACE @OPT BNF_PunctuationChoice starOrPlus =
      new BNF_PunctuationChoice("*", "+");
  }

  public static class BNF_Group extends TokenSequence
  {
    public PunctuationLeftParen leftParen;
    public @NOSPACE BNF_Expression expression;
    public @NOSPACE PunctuationRightParen rightParen;
    public @NOSPACE @OPT BNF_PunctuationChoice starOrPlus =
      new BNF_PunctuationChoice("*", "+");
  }

  public static class BNF_Optional extends TokenSequence
  {
    public PunctuationLeftBracket leftBracket;
    public @NOSPACE BNF_Expression expression;
    public @NOSPACE PunctuationRightBracket rightBracket;
  }
}
```

Figure 7. BNF Program Grammar Snippet

2.1.2. The BNF Program Grammar

Figure 7 shows a section of the Program Grammar for BNF. It is used to read the EBNF grammar in Figure 6.

2.1.3. Generated Program Grammar for EBNF

Figure 8 shows a section of the generated Program Grammar. All spaces and newlines are exactly as output from the transformation framework.

```
public static class BNF_Group_2 extends TokenChooser
{
  public BNF_Literal literal;

  public static class BNF_Sequence_1 extends TokenSequence
  {
    public BNF_EbnfIdentifier ebnfIdentifier;
    public @OPT BNF_EbnfModifier ebnfModifier;
  }
  public BNF_EbnfOptional ebnfOptional;
  public BNF_EbnfGroup ebnfGroup;
}
```

Figure 8. Generated Program Grammar Snippet

Class names are auto-generated in many cases because BNF grammars are not as descriptive as Program Grammars.

Somewhat recursively, the generated Program Grammar is used to parse the original BNF grammar again. The code snippet in Figure 8 contains almost all of the `ebnf-expression` rule (lines 4-5) in Figure 6. It does not include the `+` quantifier.

2.1.4. Re-Parsing the Original EBNF Grammar

The generated Program Grammar was used to re-parse the original EBNF grammar from Figure 6.

```
 Seq  SLn   SC  ELn   EC  Token Type           Text
==== ==== ==== ==== ====  ==================== =================
   1    1    1    8    0  Grammar_EBNF_bnf
   2    1    1    8    0  BNF_EbnfProgram
   3    1    1    8    0  TokenList<>
   4    1    1    2    0  BNF_EbnfRule
   5    1    1    1   12  BNF_EbnfIdentifier   ebnf-program
   6    1   14    1   16  BNF_Punct            ::=
   7    1   18    1   27  BNF_EbnfAlternation
   8    1   18    1   27  BNF_EbnfExpression
   9    1   18    1   27  TokenList<>
  10    1   18    1   27  BNF_Sequence_1
  11    1   18    1   26  BNF_EbnfIdentifier   ebnf-rule
  12    1   27    1   27  BNF_Punct            *
  13    1   28    1   28  BNF_Punct            ;
```

Figure 9. Parse Log from EBNF Re-Parse

Figure 9 shows a partial log from re-parsing the original EBNF grammar using the generated Program Grammar in Figure 8.

2.2. COBOL to Python (or Java or C#)

This example shows conversion of a COBOL program to Python. The exact same process is used for conversion to Java and C#, and the output is exactly the same.

2.2.1. Source COBOL Program

The COBOL program in Figure 10 prints 0-0-0 to 9-9-9 twice, using two variations on the `PERFORM` verb, like a mileage odometer.

```
WORKING-STORAGE SECTION.
01 Dial.
  02 Hundreds          PIC 99 VALUE ZEROS.
  02 Tens              PIC 99 VALUE ZEROS.
  02 Units             PIC 99 VALUE ZEROS.

PROCEDURE DIVISION.
Begin.
  DISPLAY "Using an out-of-line Perform".
  DISPLAY "Start mileage counter simulation 1".
  PERFORM CountMileage
    VARYING Hundreds FROM 0 BY 1 UNTIL Hundreds > 9
      AFTER   Tens FROM 0 BY 1 UNTIL Tens > 9
      AFTER   Units FROM 0 BY 1 UNTIL Units > 9
  DISPLAY "End of mileage counter simulation 1."
```

Figure 10. Excerpt from MileageCount.CBL

There are two paragraphs, called `Begin` and `CountMileage` with the former calling the latter. The primary focus of this experiment is converting two different forms of the PERFORM verb.

2.2.2. COBOL Transformation

The inline version of PERFORM is shown in Figure 11; the version that calls a separate paragraph was shown in Figure 3.

```
if (token instanceof COBOL_PerformUntil)
{
  COBOL_PerformUntil until = (COBOL_PerformUntil) token;

  // Collect all the inline statements
  ArrayList<Stmt> actions = new ArrayList<Stmt>();
  for (COBOL_Statement statement : inline.statements)
  {
    Stmt newStatement = trans.transformStatement(statement);
    actions.add(newStatement);
  }
  Expr termCond = trans.transformExpression(until.condition);
  Expr notTerm = trans._target.createNot(termCond);

  return trans._target.createDoStatement(actions, notTerm);
}
```

Figure 11. Transformation Logic for inline PERFORM

There are many additional variations on the `PERFORM` verb in COBOL, which are not part of this experiment.

```
import sys
class MileageCount:
  def Begin(self):
    print "Using an out-of-line Perform"
    print "Start mileage counter simulation 1"
    for Dial.Hundreds in range(0, (9) + 1, 1):
      for Dial.Tens in range(0, (9) + 1, 1):
        for Dial.Units in range(0, (9) + 1, 1):
          self.CountMilage()
    print "End of mileage counter simulation 1."
    sys.exit(0)
  def CountMilage(self):
    Disp.Hunds = Dial.Hundreds
    Disp.Tens = Dial.Tens
    Disp.Units = Dial.Units
    print '{}{}{}{}{}'.format(
        Disp.Hunds, "-", Disp.Tens, "-", Disp.Units)
```

Figure 12. Excerpt of Generated MileageCount.py

2.2.3. Generated Python Program

The generated Python program is shown in Figure 12. It was generated by parsing the input COBOL program, creating an in-memory Program Semantic Tree. Then the COBOL transformation engine was used to create an instance of a target PST in memory. The COBOL transformation engine was not aware of the target language and was also able to produce both Java and C# programs. The target PST was written to a text file, using the Java annotations in the Program Grammar to control formatting.

2.3. Delphi to C# (or Java or Python)

Delphi (originally Pascal) programs are relatively easy to transform because the language was designed to be simple and efficient [14].

2.3.1. Source Delphi Program

This sample program was written in Pascal before it was replaced by Delphi. The program prints the first few prime numbers.

```
Program Prime;

{$I IsPrime.p}

Var
  I : Integer;
  N : Integer;

Begin
  Write('Enter N -->');
  ReadLn(N);
  For I := 1 to N do Begin
    If IsPrime(I) then Begin
      WriteLn(I, ' is prime')
    End
  End
End.
```

Figure 13. Source Prime.pas

Note the `{$I IsPrime.p}` in Figure 13. Our framework has a full macro pre-processor to handle this, as described in Section 1.5.

```
Function IsPrime(N : Integer) : Boolean;

Var
  K : Integer;

Begin
  If N < 2 Then
    IsPrime := False
  Else If N = 2 Then
    IsPrime := True
  Else If N mod 2 = 0 Then
    IsPrime := False
  Else Begin
    K := 3;
    While K * K <= N Do Begin
      If N mod K = 0 Then Begin
        K := N;
        IsPrime := False
      End;
      K := K + 2
    End;
    IsPrime := True
  End
End;
```

Figure 14. Source IsPrime.p

The contents of IsPrime.p, shown in Figure 14, are merged into Prime.pas (in memory), and that merged file is used for transformation. As shown in the report in Section 3, our framework tracks original file names and line numbers.

2.3.2. Delphi Transformation

The logic shown in Figure 15 is part of the Delphi expression transformation logic. The `RELATIONALS` enumeration is used to shield Delphi from the actual representation of relational operators in the target language.

The left and right sub-expressions are handled on the Delphi side, using the `trans` instance. Each will return an expression in the target language. This modularity facilitates the development of large scale transformation efforts.

```
if (which instanceof Delphi_Relational)
{
  Delphi_Relational relational = (Delphi_Relational) which;
  AbstractToken whichOper = relational.relOp.getWhich();
  Expr left = transform(trans, relational.left);
  Expr right = transform(trans, relational.right);

  RELATIONALS rel;
  if (whichOper instanceof Delphi_PunctChoice)
  {
    String oper = ((Delphi_PunctChoice) whichOper).getValue();
    if (oper.equals("<")) rel = RELATIONALS.LT;
    else if (oper.equals("<=")) rel = RELATIONALS.LE;
    else if (oper.equals("=")) rel = RELATIONALS.EQ;
    else if (oper.equals("<>")) rel = RELATIONALS.NE;
    else if (oper.equals(">=")) rel = RELATIONALS.GE;
    else if (oper.equals(">")) rel = RELATIONALS.GT;
    else return null;
  }
  else return null;

  return trans._target.createRelational(left, rel, right);
}
```

Figure 15. Transforming Delphi Relational Expressions

The method called `createRelational` is target language specific and uses the `RELATIONALS` enumeration to decide how best to implement the relational operation.

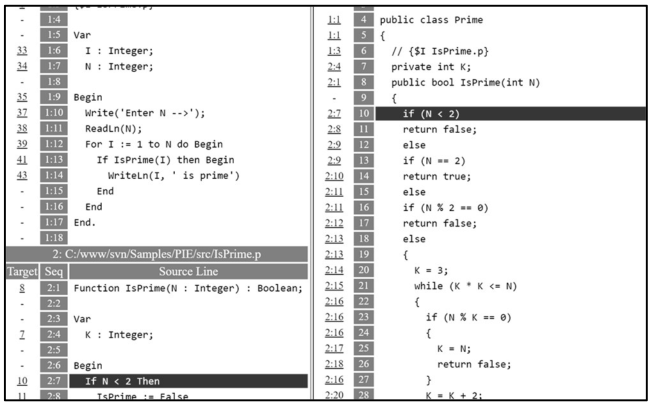
2.3.3. Running the Generated C# Program

This sample takes as input a number and lists the prime numbers up to the given number. The functional test framework supports simulated input in addition to validating output against expected values.

3. Transformation Reports

All of the transformation tools in this framework track the origin (source file, line and column numbers) of every token. This allows the creation of side-by-side reports, where the left side of the report has the original source file(s) and the right side has the transformed target. Each side has hyperlinks to the other side to show how individual statements are mapped.

Figure 16 shows a portion of a side-by-side report from Section 2.3.2. On the left side are the two source files (Prime.pas and IsPrime.p) and on the right side is the generated Prime.java. As a developer moves the mouse over the html report, the corresponding lines are highlighted. The lines are also hyperlinked to each other.



```
 -   1:4
 -   1:5   Var
33   1:6     I : Integer;
34   1:7     N : Integer;
 -   1:8
35   1:9   Begin
37   1:10    Write('Enter N -->');
38   1:11    ReadLn(N);
39   1:12    For I := 1 to N do Begin
41   1:13      If IsPrime(I) then Begin
43   1:14        WriteLn(I, ' is prime')
 -   1:15      End
 -   1:16    End
 -   1:17  End.
 -   1:18
        2: C:/www/svn/Samples/PIE/src/IsPrime.p
Target Seq  Source Line
 8   2:1   Function IsPrime(N : Integer) : Boolean;
 -   2:2
 -   2:3   Var
 7   2:4     K : Integer;
 -   2:5
 -   2:6   Begin
10   2:7     If N < 2 Then
11   2:8       IsPrime := False
```

```
1:1    4  public class Prime
1:1    5  {
1:3    6    // {$I IsPrime.p}
2:4    7    private int K;
2:1    8    public bool IsPrime(int N)
 -     9    {
2:7   10      if (N < 2)
2:8   11      return false;
2:9   12      else
2:9   13      if (N == 2)
2:10  14      return true;
2:11  15      else
2:11  16      if (N % 2 == 0)
2:12  17      return false;
2:13  18      else
2:13  19      {
2:14  20        K = 3;
2:15  21        while (K * K <= N)
2:16  22        {
2:16  23          if (N % K == 0)
2:16  24          {
2:17  25            K = N;
2:18  26            return false;
2:16  27          }
2:20  28          K = K + 2;
```

Figure 16. Sample Side-by-Side Report

4. Future Work

Programming language transformation is a challenging task. Expecting to solve it perfectly with automation is unrealistic. We are trying to provide a framework upon which a high level of automation can be achieved.

4.1. Open Source

All Program Grammars and transformation tools will be open sourced. It is unsustainable to expect any one group of developers to maintain all languages, and all transformation logic. The core parser using Program Grammars has been patented.

4.2. Dataflow Analysis

Data flow analysis and Control flow analysis [15] are interconnected. Following data and control flow across different programs is especially challenging. We are actively researching data flow analysis when data goes into a file or database in one program and back out in another program, often written in a different programming language.

We are also looking at end-to-end data flow from what is typed into a web screen, to what shows up on another web screen or report.

4.3. Abstract Data Types & Dynamic Analysis

Abstract Data Types [16] are fairly well defined but present some interesting issues when dealing with legacy languages. COBOL, for example, has a picture XX string that is limited to two characters. COBOL programmers often rely on automatic truncation into that string. In a modern language, there are several different ways to transform this string, depending on whether or not automatic truncation ever happens.

We are working on integrating dynamic analysis into our framework so we can collect run-time metrics on how often events occur, based on some test set. If the test set mirrors the production system closely enough, then transformation decisions can be automated.

5. Conclusions

Our first goal for this experiment was to transform several different source programs written in different languages, into various target languages. Specifically, the transformation logic was kept separate from the choice of target language. This was successfully demonstrated on three different examples.

Because the Program Grammars are expressed in Java, and the transformation logic is also expressed in Java, any changes made to the grammars will be immediately detected by the Java compiler. This has satisfied our second goal of synchronization.

Our third goal was to use existing Program Grammars for writing the generated program to a text file. This was achieved with annotations that are used by the program generator, but are ignored by the parser.

It is our hope and intention that these tools achieve success by open-sourcing them. Programming language transformation has not been fully "solved" by the community. As the existing software base ages, it will be harder and harder to migrate legacy applications. Building shiny new applications is fun, but businesses rely on the logic embedded in existing application programs.

References

[1] W. Rösler, "The Hello World Collection," April 2017. [Online]. Available: https://helloworldcollection.github.io/.

[2] O. Schade, G. Scheithauer and S. Scheler, "99 Bottles of Beer," May 2017. [Online]. Available: http://www.99-bottles-of-beer.net/.

[3] S. Cass, "The Top 10 Programming Languages," *IEEE Spectrum,* p. 68, July 2014.

[4] "Modern Systems," May 2017. [Online]. Available: http://modernsystems.com/.

[5] H. M. Sneed, "Migrating from COBOL to Java," in *IEEE International Conference on Software Maintenance (ICSM)*, Timisoara, Romania, 2010.

[6] P. Rubens, "Why it's time to learn COBOL," April 2016. [Online]. Available: http://www.cio.com/article/3050836/developer/why-its-time-to-learn-cobol.html.

[7] R. L. Mitchell, "Meet Cobol's hard-core fans," 21 August 2014. [Online]. Available: http://www.computerworld.com/article/2491375/enterprise-applications/meet-cobols-hard-core-fans.html.

[8] "Knowledge Discovery Metamodel," May 2017. [Online]. Available: http://www.omg.org/technology/kdm/.

[9] M. Bravenboer, K. T. Kalleberg, R. Vermaas and E. Visser, "Stratego/XT 0.17. A language and toolset for program transformation," *Science of Computer Programming 72,* pp. 52-70, 2008.

[10] P. Klint, T. van der Storm and J. Vinju, "EASY Meta-programming with Rascal," in *Generative and Transformational Techniques in Software Engineering III*, Berlin, Springer-Verlag, 2011, pp. 222-289.

[11] S. A. O'Hara, "Programmars: A Revolution in Computer Language Parsing," in *SERP '15: International Conference on Software Engineering Research and Practice*, Las Vegas, NV, 2015.

[12] T. Molderez and C. De Roover, "Automated Generalization and Refinement of Code Templates with EKEKO/X," Software Languages Lab, Vrije Universiteit, Victoria, BC, Canada, 2014.

[13] J. R. Cordy, "The TXL source transformation language," *Science of Computer Programming,* pp. 190-210, 2006.

[14] N. Wirth, "The design of a pascal compiler," *Software: Practice and Experience,* pp. 309-333, 1971.

[15] O. Shivers, *Control-Flow Analysis of Higher-Order Languages,* Pittsburgh, PA: School of Computer Science, Carnegie Mellon University, 1991.

[16] N. Dale and H. M. Walker, Abstract Data Types: Specifications, Implementations, and Applications, Burlington, MA: Jones & Bartlett Learning, 1996.