Eaglegacy Decapentannual Report

2011 - 2026

Steven A. O'Hara, PhD

Eagle Legacy Modernization, LLC

This document was written by Steven A. O'Hara, PhD

CEO and Founder of Eagle Legacy Modernization, LLC

Published January 2026

TABLE OF CONTENTS

1. Summary 1
1.1. Parser 1
1.2. Programmars 1
1.3. Parsing Management Tools 1
1.4. Testing 1
1.5. Analysis Tools 1
1.6. Transformation 2
2. Background 2
2.1. Existing Parsing Tools 2
2.2. Existing Solution Attempts 3
2.3. Modernization History 4
2.4. Parsing for Analysis versus Compiling 6
3. Eagle Legacy History 6
3.1. March 2008 to November 2010, Prior to Founding 6
3.2. December 2010 to October 2011, Invention Phase 6
3.3. June 7, 2011, Launch 10
3.4. Progress from 2011 to 2026 10
4. Architectural Principles 11
4.1. Do Not Alter Original Source Code 11
4.2. Do Not Require All Dependent Files 11
4.3. Users Can Write a Little Java 11
4.4. Distance Matters 11
4.5. Open Source all the Programmars 12
4.6. Test as Much as Possible 12
5. Eagle Legacy Status 12
5.1. The Parser 13
5.2. Eagle Projects 17
5.3. Programmar Debugging 18
5.4. "Pretty Printing" 20
5.5. Online Parser – (Undergoing conversion from AWS Elastic Beanstalk to AWS Lambdas) 22
5.6. Macro Processing 23
5.7. Interpretation 24

6. Visualization Tools 2
6.1. Project Browser 2
6.2. Language Browser 7
6.3. FileInventory.csv 11
6.4. Symbol Definitions and References Visualization 12
6.5. Functional Tests 13
7. Transformation 14
7.1. Translating Human Languages 14
7.2. Manual Transformation 15
7.3. Fully Automatic Transformation 15
7.4. Hybrid Approaches 15
7.5. Eagle Legacy Transformation 15
8. Programmars 17
8.1. C / C++ Language 18
8.2. COBOL 19
8.3. HTML 21
8.4. Java 22
8.5. Lisp 23
8.6. Natural 23
8.7. PHP (Perl) 24
8.8. PL/I, Programming Language One 24
8.9. Python 25
8.10. Report Program Generator (RPG) 26
8.11. Other Languages 27
9. Parsing Patent 28
10. Publications 28
10.1. Paper #1: "Programmars: A Revolution in Computer Language Parsing" 28
10.2. Paper #2: "Toward Effective Management of Large-Scale Software" 28
10.3. Paper #3: "Improving Programming Language Transformation" 29
10.4. Paper #4: "Modernizing Parsing Tools" 29
10.5. Unpublished Research on Cross-Language Data Flow 30
11. Biography 30

LIST OF FIGURES

Figure 1. Sample Antlr Grammar 2
Figure 2. "A COBOL Migration you do NOT want" 4
Figure 3. California DMV 1994 Failure 5
Figure 4. California 2008 Restart 5
Figure 5. California Project Canceled in 2013 5
Figure 6. Traditional Parsing Approach 7
Figure 7. New Parsing Approach 7
Figure 8. Simplified COBOL PERFORM Verb (old way) 8
Figure 9. Simplified COBOL PERFORM Verb (new way) 8
Figure 10. Old-style Grammar for a Java Floating Point Number 9
Figure 11. New-style Programmar for Java Numbers 9
Figure 12. Some C Expression Operators (slightly abbreviated) 10
Figure 13. Java's Not Operator (!), Unedited 12
Figure 14. Basic Syntax Elements 13
Figure 15. Basic Non-Terminal Token Types 13
Figure 16. Token Types 14
Figure 17. Basic Terminal Token Types 14
Figure 18. Source Code Line Repair 15
Figure 19. EagleParser Options 15
Figure 20. Sample Programmar XML Snippet 16
Figure 21. Sample Expression Unit Test 16
Figure 22. Sample Logic Unit Test 16
Figure 23. Sample Expression Interpreter (slightly condensed) 17
Figure 24. Simple Project Definition (condensed) 17
Figure 25. Sample Parse Failure (slightly condensed) 18
Figure 26. PPSM Traditional Grammar 19
Figure 27. PPSM Programmar 19
Figure 28. Sample Trace Output 19
Figure 29. Sample Debugging Session 20
Figure 30. Programmar for Javascript "try" Block 21
Figure 31. Google Search Result (Before) 21
Figure 32. Google Search Result (After), unedited 22
Figure 33. The Online Parser Screen 22
Figure 34. Online Parse Success (truncated) 23
Figure 35. Online Parse Failure 23
Figure 36. Sample AWK Expression Tests 24
Figure 37. Sample Julia Statement Logic Test 25
Figure 38. Roman Numeral Conversion Test Output 25
Figure 39. Ambiguous Functions in Python 1
Figure 40. Sample Running an Ada Program 2
Figure 41. Sample Dashboard Projects View 3
Figure 42. Sample View for One Project 4
Figure 43. Summary View for One Project 4

Figure 44. Sample Program File View ... 5
Figure 45. Successful Parse Display ... 6
Figure 46. Parse Failed Display ... 6
Figure 47. Dashboard Languages Browser ... 7
Figure 48. Summary Languages Browser, by Project ... 8
Figure 49. Sample Grammar View ... 8
Figure 50. Programmar Instances ... 9
Figure 51. Highlighted View of a Token ... 10
Figure 52. COBOL Parse Failures ... 10
Figure 53. Sample PST Listing ... 11
Figure 54. FileInventory.csv Excerpt ... 11
Figure 55. Sample BNF file ... 12
Figure 56. Sample Symbol Table ... 12
Figure 57. Project Browser Function Test ... 13
Figure 58. Project Browser Directory Test ... 13
Figure 59. Lisp Browser Functional Test ... 13
Figure 60. CMD Browser Functional Test ... 14
Figure 61. Symbol Browser Functional Test ... 14
Figure 62. Grammar Token Functional Test ... 14
Figure 63. Sample COBOL Paragraph (Function) ... 16
Figure 64. Generated C# Method ... 16
Figure 65. Left side (C) of Side-by-Side Report ... 17
Figure 66. Right side (Python) of Side-by-Side Report ... 17
Figure 67. Sample Macro Function in C (slightly condensed) ... 18
Figure 68. After Macro Processing ... 19
Figure 69. COBOL Derision ... 19
Figure 70. Different COBOL Languages (simplified) ... 20
Figure 71. Sample COBOL Level Numbers ... 20
Figure 72. HTML Script Definition (condensed) ... 21
Figure 73. HTML Mixed with Javascript and Jinja ... 22
Figure 74. Partial JavaP Output ... 23
Figure 75. Lisp 'defun' Function Definition and Usage ... 23
Figure 76. Small Natural Code Snippet ... 24
Figure 77. PHP Derision ... 24
Figure 78. Snippet of PHP Code ... 24
Figure 79. PL/I Derision ... 25
Figure 80. Snippet of PL/I Code ... 25
Figure 81. Snippet of Python Code ... 26
Figure 82. Python 'print' Statement ... 26
Figure 83. Sample RPG Snippet ... 26
Figure 84. RPG Wide and Narrow Formats ... 27

1. Summary

After leaving my job as CTO at a Software Modernization company, I realized that the single biggest obstacle in legacy modernization is Scalability. Eagle Legacy Modernization, LLC was created to address that problem. This document is a technical summary report of the first fifteen years of development in the company, briefly summarized in this section.

The goal of Eagle Legacy Modernization, LLC is to create core functionality that will enable a wide variety of Analysis tools to be built, helping organizations modernize their existing software, especially when many different languages are involved.

1.1. Parser

The first step was to build a Parser capable of parsing all major computer languages without custom pre-processing steps. This work was completed in 2011, published in a peer-reviewed paper in 2015, and awarded a Patent in 2017. The crucial aspect of this parser is that the grammar is ordinary Java code (called a Program Grammar, or "Programmar") so grammars and analysis tools are forced to stay in sync. The Parser has been very stable since about 2011.

1.2. Programmars

There is an on-going effort to create Programmars for all computer languages. All existing Programmars have been published to a GitHub[1] account. Currently, there are 34 languages supported, such as COBOL, Natural, RPG, PL/I, C, C++, Java, C#, HTML, DOS, PHP and others. This company's intention is to create a global standard Programmar for every major computer language.

1.3. Parsing Management Tools

Each Project can have millions of lines of code in thousands of files written in dozens of computer languages, which introduces many challenges. An interactive Debugger is available, as well as several Tracing tools, to help tune Programmars. A web-based Browser application is available as well to visualize parsing progress, view source files, view Programmars, etc.

1.4. Testing

Writing so many Programmars is challenging, so a Testing framework has been established. It allows small pieces of code to be written and interpreted without access to the original program compiler. For example, there are COBOL tests that verify that expressions are parsed correctly. This means, of course, that the Programmars have hooks for interpreting (evaluating) the parse results. Many additional unit tests and functional tests are in place.

1.5. Analysis Tools

Efforts are still in progress for providing tools to assist with data flow, control flow, file dependencies, etc. All of which are vital for getting a thorough analysis of software. Research on the core issues of classes, super-classes, variable scoping, etc. is still ongoing. The framework for variable cross-referencing is in place, but many of the Analysis Tools are interdependent.

[1] github.com/oharasteve/eagle_legacy_programmars

1.6. Transformation

Several demonstrations have been built, but fully automated Transformation is an extraordinarily challenging problem. Our demonstrations provide essentially simple transformations from many programming languages to just a few target languages, namely C#, Java and Python.

2. Background

There are hundreds of billions of lines of code in use today[2]. Several hundred billion of them are in antiquated languages like COBOL, RPG and Natural. The original developers are unlikely to still be available to maintain these systems, and it is very difficult to find or train programmers in these legacy languages.

The technologies presented here are equally applicable to modern programming languages, because they continue to evolve as well. Legacy systems frequently have modern "wrappers" on them to allow the applications to run over the internet. To get a full analysis requires spanning both legacy and modern systems.

2.1. Existing Parsing Tools

Most tools in use (such as Antlr[3] and Bison[4]) today are based on yacc[5], which was invented in the early 1970's. See Figure 1 for a trivial ANTLR grammar[6]. The lack of standard grammars has made it difficult to tackle modernization problems. Furthermore, the grammars are written as text files, while the tools are written in a programming language like C++ or Java. There is a large gap between the two, where changes made to the grammar are very difficult to detect in the tools. Unfortunately, grammars are usually under continuous change to accommodate new source code so the two systems often get out of sync, with no easy way to force consistency.

```
eval : additionExp ;  /* entry point */

additionExp : multiplyExp ( '+' multiplyExp | '-' multiplyExp )* ;
multiplyExp : atomExp ( '*' atomExp | '/' atomExp )* ;
atomExp : Number | '(' additionExp ')' ;  /* highest precedence */

Number : ('0'..'9')+ ('.' ('0'..'9')+)? ;

WS : (' ' | '\t' | '\r'| '\n') {$channel=HIDDEN;} ;  /* Whitespace */
```

Figure 1. Sample Antlr Grammar

The typical output from existing tools is called an Abstract Syntax Tree (AST). It is generally saved as a text file much like XML, and contains details about all the tokens found in the original computer program.

In contrast, our system introduces the Programmar Semantic Tree (PST). It is an instance of a Java Programmar class, rather than a text file, and contains details about all the tokens in the original

---

[2] e.g. embedded.com/electronics-blogs/break-points/4026908/A-Trillion-Lines-of-Code-
[3] en.wikipedia.org/wiki/ANTLR
[4] en.wikipedia.org/wiki/GNU_bison
[5] en.wikipedia.org/wiki/Yacc
[6] stackoverflow.com/questions/1931307/antlr-is-there-a-simple-example (condensed)

computer program. However, all those details are stored as instances of other objects, rather than just as text.

Interestingly, the PST (output from the Parser) is generally stored in precisely the same classes as the Programmar (input to the Parser).

2.2. Existing Solution Attempts

There are two primary approaches to commercializing Legacy Modernization. Service companies like Accenture or Deloitte will bill hourly for labor, and perform a largely manual Modernization project. Perhaps it will be outsourced, and perhaps they will use some internal tools to assist, but they are not motivated to build tools that customers can use themselves.

Product companies such as Appian[7] and Dell Technologies[8] have some core technology, and layer services on top to perform legacy transformations. Unfortunately, the tension between quality and readability must always be resolved in favor of quality. Customers need to be confident that their software still runs correctly post-transformation. Even if the new software is unmaintainable and still looks and behaves like the old software.

This is a difficult problem, but it is also a vital problem facing many businesses. According to a 2020 article in Business Wire[9], “74% Of Organizations Fail to Complete Legacy System Modernization Projects”. We do not claim to have a magical solution – rather we are providing a solid foundation upon which solutions can be built. Basing modernization tools on 1970’s technologies like yacc and lex is ironic at best.

The pejorative term "JOBOL" is often used to describe the results of a COBOL to Java transformation that still has relics from COBOL implemented in Java (such as PICTURE clauses). See Figure 2 for an example[10].

[7] appian.com

[8] dell.com/en-us/lp/dt/cloud-application-modernization

[9] businesswire.com/news/home/20200528005186/en/74-Of-Organizations-Fail-to-Complete-Legacy-System-Modernization-Projects-New-Report-From-Advanced-Reveals

[10] Copied from www.semdesigns.com/Products/Services/EraneaNACA.html, as something to avoid

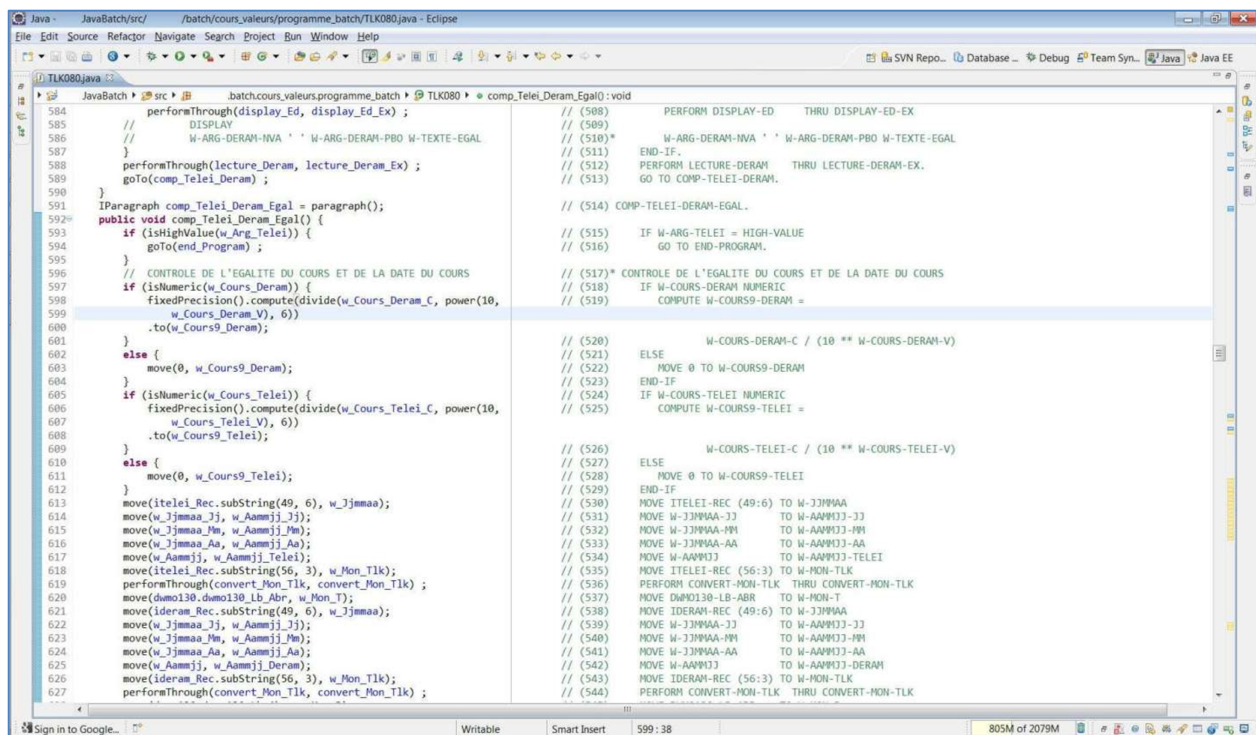

Figure 2. "A COBOL Migration you do NOT want"

A few companies attempt to bridge the gap with a hybrid approach. In all cases, the end goal is confidence in transformation, which comes from extensive Testing. There is no way to bypass the need for thorough Testing of all software, both before and after transformation.

2.3. Modernization History

This field has an abundance of well-publicized software failures, notably California DMV (See Figure 3 for 1994[11], Figure 4 for 2008[12] and Figure 5 for 2013[13]). Even as the needs are increasing daily, the tools are stagnant with little innovation. Large scale successful transformations are very expensive and generally produce un-modifiable software. Taking a COBOL mainframe application and converting it to run on the latest iPhone is an extraordinary challenge.

However, consider the recent advances made in human language translation and interpretation. These efforts have taken decades to become proficient. The new Large Language Models (LLM) technology has finally made this feasible. However, perfection in human languages is not required because the output is generally meant for humans, whereas the output from Legacy Modernization is for computers and must be verifiably perfect.

It is our belief that the optimal approach is to build interpreters to collect dynamic analysis metrics, build detailed unit and functional tests and compare results before and after transformation. Verifiability is vital.

[11] articles.latimes.com/1994-04-27/news/mn-50941_1_modernization-project
[12] www.digitalcommunities.com/articles/California-DMV-Awards-76-Million-Contract.html
[13] www.latimes.com/local/la-xpm-2013-feb-14-la-me-dmv-project-20130215-story.html

Los Angeles Times | CALIFORNIA & LOCAL | ENTERTAINMENT | SPORTS | BUSINESS | TECHNOLOGY | NATION | POLITICS | WORLD | MORE

YOU ARE HERE: LAT Home → Collections → Mismanagement

FROM THE ARCHIVES

Top California courts administrator to step down
March 23, 2011

MORE STORIES ABOUT

Mismanagement
Computer Applications
California Department Of Motor Vehicles
California -- Government Agencies -- Finances

DMV Spent $44 Million on Failed Project : Technology: Agency's director says six-year effort at computer modernization can't be saved. Legislative probe is ordered.

April 27, 1994 | CARL INGRAM | TIMES STAFF WRITER

Email Share G+1 2 Tweet Recommend 0

SACRAMENTO — The California Department of Motor Vehicles has spent $44 million over the last six years on a computer modernization project it now admits is a hopeless failure, prompting the Legislature to order an investigation.

DMV Director Frank S. Zolin, who has been on the job since 1991, said Tuesday that the project got off on the wrong track and that he has ended it.

A report by the independent legislative analyst's office said the DMV didn't understand the technology and mismanaged the project.

Figure 3. California DMV 1994 Failure

digital communities

Digital Communities
Current Issue
Past Issues
Subscribe
Digital Communities Blogs
In the Trenches
Notes from a City CIO
Intelligent Communities
Surveys and Awards

California DMV Awards $76 Million Contract to Modernize Applications

January 8, 2008 By News Report
The California Department of Motor Vehicles (DMV) awarded EDS a $76 million, six-year contract to assist in modernizing applications technology. The Information Technology Modernization (ITM) project will update the DMV's legacy software application technology to more current application and database structures.

Tweet
Recommend 0
G+1
You May Also Like

Figure 4. California 2008 Restart

Sections | Los Angeles Times | SUBSCRIBE | LOG IN

CALIFORNIA

Half-finished $208-million DMV technology overhaul canceled

By Chris Megerian, Los Angeles Times

Feb. 14, 2013 12 AM PT

SACRAMENTO — California's computer problems, which have already cost taxpayers hundreds of millions of dollars, have mounted as state officials cut short work on a $208-million DMV technology overhaul that is only half done.

Figure 5. California Project Canceled in 2013

2.4. Parsing for Analysis versus Compiling

There are two major distinctions between parsing for analysis and parsing for compilation. In order to compile, the exact library paths must be specified, and all dependent files must be available. For analysis, it is very common to be missing dependent files, and parsing should still be possible. This means that object types are not necessarily known while parsing.

The other major issue is that a compiler is supposed to reject invalid programs. But parsing for analysis can assume that the source program is already well-formed and has been compiled. This, of course, isn't always true. For example, declaring a method as both Public and Private at the same time makes no sense. But it is not necessary, in general, for our parsing tools to reject programs like this.

An additional distinction is that parsing for analysis can be used to help identify "curious" artifacts that should probably be Warnings but are not. For example, Python does not expect semicolons (;) at the end of lines. But it allows them, because the semicolon is a statement separator. In a Programmar, we can label a token with @CURIOUS("message") and it will parse successfully, but display the message upon completion.

3. Eagle Legacy History

3.1. March 2008 to November 2010, Prior to Founding

Dr O'Hara was Chief Technical Officer at a software company specializing in legacy modernization. During his tenure there, he introduced standard tools as well as modern software engineering practices. He also led development teams on many Analysis projects, as well as several successful, but relatively small, Transformation projects. A barrier that was consistently encountered was Scalability. Grammars were always being updated, and the tools that processed them were often failing blindly. They used a hybrid approach, with automation up front, followed by groups of experienced C# developers polishing and verifying the transformed software.

3.2. December 2010 to October 2011, Invention Phase

After leaving the software company, several ex-employees had some great ideas and built lots of experiments on how to solve the Scalability problem. At the outset, we didn't really have a clear vision of what the new technology would look like. It was during this period that several crucial ideas were tested and turned out to be successful. The items in the next few sections are essentially the basis for the Programmar Patent (see section 9, "Parsing Patent").

Graphically, the difference between Figure 6 and Figure 7 is the core of the new Parser. The first figure includes two elements that are no longer necessary. Traditional Grammars and the separate Semantic Information database are now part of the PST's.

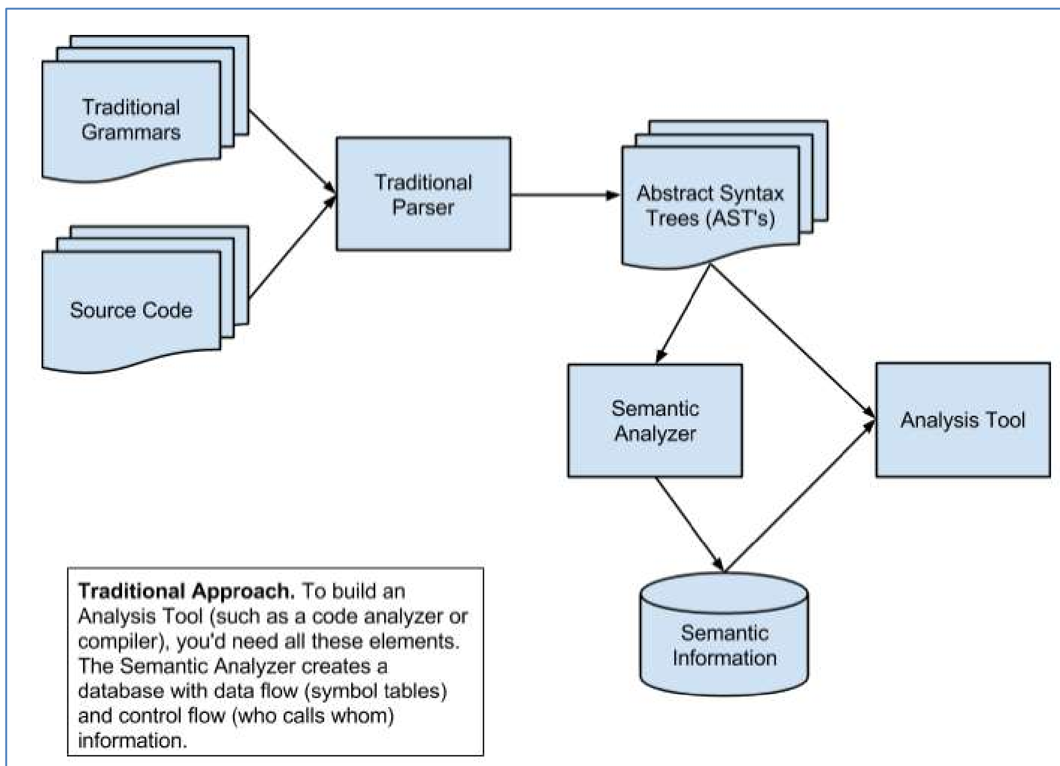

Figure 6. Traditional Parsing Approach

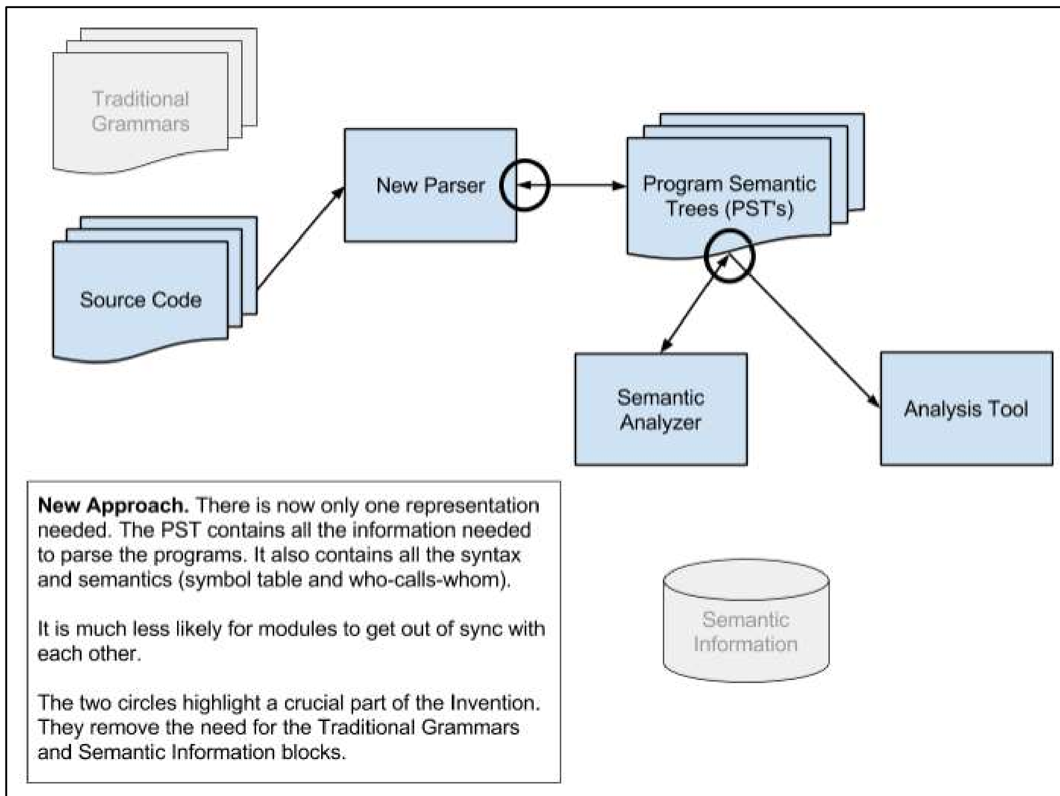

Figure 7. New Parsing Approach

3.2.1. Store Grammars in Java Classes

The single biggest part of the invention was to create grammars using ordinary Java code, not as a separate text file. This involved a mapping between the elements found in a traditional grammar and some abstract Java classes. For example, compare Figure 8 with Figure 9. The new way is more verbose,

but now elements have names. Dealing with the old AST's was like reading XML documents. You had to use an XPath[14]-like expression to navigate and find the element you needed. With the new PST, elements are referenced by Java classes and fields.

```
cPerform := "PERFORM" cParagraph
    [("THROUGH" | "THRU") cParagraph] [cPerfTimes];
cPerfTimes := cExpression "TIMES";
cParagraph := cIdentifier;
cExpression := cIdentifier | cNumber;
cIdentifier := cLetter (cLetter | cDigit | "-")*;
cNumber := cDigit cDigit*;
cLetter := "A" .. "Z";
cDigit := "0" .. "9";
```

Figure 8. Simplified COBOL PERFORM Verb (old way)

```
class COBOL_Perform extends COBOL_AbstractStatement {
  @S(10) COBOL_Keyword PERFORM = new COBOL_Keyword("PERFORM");
  @S(20) COBOL_Paragraph startPara;
  @S(30) @OPT COBOL_PerformThrough through;
  @S(40) @OPT COBOL_PerformTimes times;

  class COBOL_PerformThrough extends TokenSequence {
    COBOL_KeywordChoice THRU = new COBOL_KeywordChoice("THRU", "THROUGH");
    COBOL_Paragraph endPara;
  }
  class COBOL_PerformTimes extends TokenSequence {
    COBOL_Expression number;
    COBOL_Keyword TIMES = new COBOL_Keyword("TIMES");
  }
}
```

Figure 9. Simplified COBOL PERFORM Verb (new way)

A surprising number of secondary benefits came along almost for free with this approach. Grammars could now be modular, split into many separate pieces, and pulled together by Java. Grammars could be abstract -- COBOL free-format and COBOL fixed-format share almost all their elements. Grammars could be encapsulated – HTML can simply reference Javascript or CSS; no need for monolithic grammars.

The `@S()` annotation is used to specify parsing Sequence. Early versions of the Programmars relied simply on the order of the fields inside the classes. Early versions of Java specified that the order of fields returned using reflection was not predictable. We started using the `@S()` annotations and decided that they are helpful even if they are no longer necessary. We also use `@P()` annotations in expressions to denote operator Precedence.

3.2.2. Process Terminal Nodes with Java Methods

It is perhaps surprising, to somebody who hasn't worked on many traditional grammars, that the terminal nodes are often much more difficult to define in a grammar than the non-terminal nodes. Figure 10 shows a sample grammar for a Java floating point number[15]. Limiting the value of the exponent, for example, is very difficult to do, especially if it depends on the "`float_suffix`" value.

[14] en.wikipedia.org/wiki/XPath
[15] cui.unige.ch/isi/bnf/JAVA/float_literal.html

```
float_literal ::=
  ( decimal_digits "." [ decimal_digits ] [ exponent_part ] [ float_suffix ] )
  | ( "." decimal_digits [ exponent_part ] [ float_suffix ] )
  | ( decimal_digits [ exponent_part ] [ float_suffix ] )
decimal_digits ::= "0..9" { "0..9" }
exponent_part ::= "e" [ "+" | "-" ] decimal_digits
float_suffix ::= "f" | "d"
```

Figure 10. Old-style Grammar for a Java Floating Point Number

It also turns out that terminal nodes are often similar between languages. String literals generally use single or double quotes, optionally allow doubled quotes, optionally have an escape character, etc. Figure 11 shows the implementation of Java numbers (other than hexadecimal constants). The parameters are: exponent characters, suffix characters, whether a trailing period is allowed, and whether there is an ignorable character available for readability. For example, Java allows `2_000_000` to mean two million.

```
public class Java_Number extends TerminalNumberToken {
   @Override
   public boolean parse(EagleFileReader lines) {
      return genericNumber(lines, "Ee", "LlFfDd", true, true, '_');
   }
}
```

Figure 11. New-style Programmar for Java Numbers

It is sometimes necessary for a Programmar to parse an element without using the shared parsers, so calling the generic number parser is optional.

3.2.3. Persist Parse Results in Java Classes

In a traditional parsing system, a grammar is used to parse the program, and the results are generally stored in an Abstract Syntax Tree (AST). In our system, a Programmar is represented in Java code directly. The results of parsing are stored in instances of the same Java code, which we call a Programmar Semantic Tree (PST). The Semantic part has two claims.

First, the parsers are context-sensitive, with the ability to examine their context while parsing. Most traditional parsers only support context-free grammars. The Programmars can optionally consider their context to decide how to parse programs. For example, parsing COBOL Level numbers depends on the preceding Level numbers, so they can be properly nested.

Second, the parsers can store semantic information collected during or after the parsing process. For example, cross-reference information on variable definitions and references is stored directly in the Java classes; no need for a separate database.

3.2.4. Use Annotations to "Pretty Print" Source Code

When doing transformations, the generated code should be readable to people as well as computers. It is also sometimes useful to be able to reprint the original source code, but reformatted so it looks nicer. This involves generating source code from a PST. By adding Java annotations to the Programmar, we can control spacing, blank lines, etc. The main formatting annotations are `@NEWLINE`, @NOSPACE, @INDENT and @OUTDENT.

3.2.5. Reduce Arithmetic Expression Complexity

Consider Figure 1 again. Each precedence level is intertwined with the adjacent levels. The definition of the addition operators depends on the multiply operators, which depend on the atomic operators, which depend on the addition operators. This gets out of control in languages like C which have a dozen or more precedence levels. In our Expression parser (called a "PrecedenceChooser"), the levels are defined completely independent of the adjacent levels. See Figure 12 for the definition of some C operators with precedence.

```
public class C_Expression extends PrecedenceChooser {
    <-- other operators -->
  public @P(1030) C_MultiplicativeExpression multiplicative;
  public @P(1040) C_AdditiveExpression additive;
  public @P(1050) C_ShiftExpression shift;
    <-- other operators -->
}
-------------------------------------------------------------------------------
public class C_MultiplicativeExpression extends PrecedenceOperator {
  public @S(10) C_Expression left = new C_Expression(this, Prec.ATLEAST);
  public @S(20) C_Punctuation operator = new C_Punctuation ("*", "/", "%");
  public @S(30) C_Expression right = new C_Expression(this, Prec.HIGHER);
}
public class C_AdditiveExpression extends PrecedenceOperator {
  public @S(10) C_Expression left = new C_Expression(this, Prec.ATLEAST);
  public @S(20) C_Punctuation operator = new C_Punctuation("+", "-");
  public @S(30) C_Expression right = new C_Expression(this, Prec.HIGHER);
}
public class C_ShiftExpression extends PrecedenceOperator {
  public @S(10) C_Expression left = new C_Expression(this, Prec.ATLEAST);
  public @S(20) C_Punctuation operator = new C_Punctuation("<<", ">>", ">>>");
  public @S(30) C_Expression right = new C_Expression(this, Prec.HIGHER);
}
```

Figure 12. Some C Expression Operators (slightly abbreviated)

Operator precedence is defined by the `@P()` precedence annotations. The `Prec.ATLEAST` and `Prec.HIGHER` parameters control the relative precedence levels. For example, the Additive Expression must have a precedence at least as high as the left argument and higher than the right argument. Hence, the left argument could be as high as another Additive Expression, but the right argument must be a Multiplicative Expression or less.

The second innovation is that empty levels in the resulting PST are omitted. Very often, expressions are very simple, perhaps just an addition of two variables. In a traditional AST, normally all the other levels are present, which can make the AST very unwieldy. In the PST, those empty levels are omitted.

3.3. June 7, 2011, Launch

Eagle Legacy Modernization, LLC was incorporated in Texas, with Steven A. O'Hara as the sole proprietor.

3.4. Progress from 2011 to 2026

The core Parser has been stable since 2011, with only minor adjustments. Since then, the work has divided up into the following categories:

- Building many grammars

- Parsing millions of lines of code in dozens of languages
- Building debugging and analysis tools
- Work on interpreting (running) parsed programs
- Work on transforming code to modern languages
- Testing, testing, testing

4. Architectural Principles

This list of Principles has been used through the development of the company. They are intended to help create core technology that can be used pervasively in our industry, and to enable organizations to perform software analysis and transformation themselves.

4.1. Do Not Alter Original Source Code

It may be simpler to just change the source code so it can be parsed, but that is never a good option. There are three main approaches to fixing problematic source code. First, update the Programmar as needed, so later programs won't have problems. Second, update the Programmar, but annotate it with `@CURIOUS` for later analysis. Third, use the 'repair' function described in section 5.1.5 to modify the in-memory version of the file.

4.2. Do Not Require All Dependent Files

If some dependent files are missing, we should still attempt to parse the file. In fact, if parsing fails for some reason, we should still attempt to parse the remainder of the file.

4.3. Users Can Write a Little Java

There are no configuration files at all in our system. They are very inflexible and difficult to read out of context. We have chosen to use a “Project” file, written in Java, instead of configuration files. It controls parsing and analysis. This is where all the logic is kept for:

- Source code and output file locations,
- Source file filtering and editing,
- Programming language identification,
- Where to find dependent files, and
- Overriding partial Programmar elements.

See Figure 24 for a simple Project definition.

4.4. Distance Matters

The main advantage of our approach over traditional approaches is Scalability. In our experience, it was very common for a grammar to be updated that caused analysis tools to break. But they usually broke silently. If the analysis tool searched the AST for instances of some grammar element, and that grammar element was moved or renamed, the analysis tool simply didn’t find it. No warnings, no errors, no alerts.

Continuing with that idea, we have collected (up to) four different types of logic into our Programmars.

1. Parsing Logic – using Reflection
2. Interpretation Logic – for running the program
3. Transformation Logic – to C#, Java and Python
4. Generation Logic (only for C#, Java and Python)

Here is the complete, unedited, Programmar for Java's "!" operator which has all four components.

```
public class Java_LogicalNotExpression extends PrimaryOperator
        implements EagleRunnable, EagleTransformableExpression {
    public @S(10) Java_Punctuation notOperator = new Java_Punctuation('!');
    public @S(20) Java_Expression expr;

    @Override
    public void interpret(EagleInterpreter interpreter) {
        boolean value = interpreter.getBoolValue(expr);
        interpreter.pushBool(!value);
    }

    @Override
    public AbstractExpression transformExpression(EagleTransformer transformer,
            EagleGenerator generator) {
        AbstractExpression theExpr = transformer.transformExpression(
            generator, expr);
        return generator.newNotExpression(theExpr, this);
    }

    public Java_Expression generateLogicalNot(Java_Expression theExpr,
            AbstractToken source) {
        this.expr = theExpr;
        this.setTransformationSource(source);
        return Java_Generator.wrapExpression(this);
    }
}
```

Figure 13. Java's Not Operator (!), Unedited

4.5. Open Source all the Programmars

The only way to get broad acceptance is to open source as much as possible. This starts with all the Programmars, but also includes analysis tools, visualization tools, etc.

4.6. Test as Much as Possible

Write as many unit tests and functional tests as possible. We use Selenium for testing the Eagle website (eaglegacy.com) as well as testing the Project Browser (flask-based).

5. Eagle Legacy Status

This section is intended to be a fairly deep dive into the technology that has already been developed. For some readers, it may be appropriate to hand this section off to a developer.

5.1. The Parser

The job of the Parser is to process a computer program and store it in a form that Analysis Tools can use. Parsers have been around for many decades, albeit with the shortcomings mentioned in Section 2.1. This section describes the main elements of Eagle Legacy's parser.

5.1.1. The Programmar is in Ordinary Java Classes

Probably the biggest advantage of the Programmar approach over traditional grammars is that the parsing rules are stored in the same language as the tools that analyze the parsing results. This means that the two cannot easily get out of sync, without causing a compiler error; which, in turn, allows for much greater Scalability. It is now possible to have many developers working concurrently on both the Programmars and the Analysis Tools simultaneously.

Second, a whole collection of advantages come along, virtually for free. Having an abstract Programmar with multiple instances is easy. Referencing a Programmar from within another Programmar is easy. Adding unit and functional tests is easy. Splitting up a Programmar into smaller, manageable modules is easy. During the early months of research (section 3.2), these advantages were unanticipated, but welcomed.

5.1.2. Separate Syntax Object

Some languages have variations in syntax without changes in their semantics. COBOL, for example, has both fixed-width programs and free-format programs. Most of the distinction is kept in the Syntax class, not the Programmar. Figure 14 shows some of the entities maintained in the Syntax object, which is embedded in each programming language definition.

Syntax Element	Default	Description
Case Sensitive	false	Are keywords case sensitive?
Continuation Char	*none*	Languages like Visual Basic use an underscore (_)
Extra Characters	*none*	Additional characters to allow in keywords
Auto Advance	true	Set to false if white space (end-of-line) matters
Punctuation Exceptions	*none*	Prevent "==" from matching the pattern "="
Comment Instance	*none*	When parse fails, discard comments and continue

Figure 14. Basic Syntax Elements

5.1.3. Non-Terminal Programmar Tokens

In a Programmar, non-terminal tokens are the ones that do not directly consume source characters, as shown in Figure 15.

Non-Terminal Token	Description
Token Sequence	Ordered list of tokens to match. Use @OPT for optional tokens.
Token Chooser	Unordered list of tokens. Can match any one of them.
Token List<T>	Sequence of one or more token of type T (@OPT to allow none)
Precedence Chooser	Handles precedence, such as multiplication before addition.
Separated List<T,P>	Sequence of one or more T, separated by P (such as a comma).
Unparsed Element	When the parser fails, but can continue by skipping some tokens.

Figure 15. Basic Non-Terminal Token Types

Figure 16 shows a diagram of the main token types. These correspond closely to the familiar BNF-like token types (for the non-terminal nodes). The Separated List token type is a short hand notation for an alternating Token List, and provides no new functionality, just brevity.

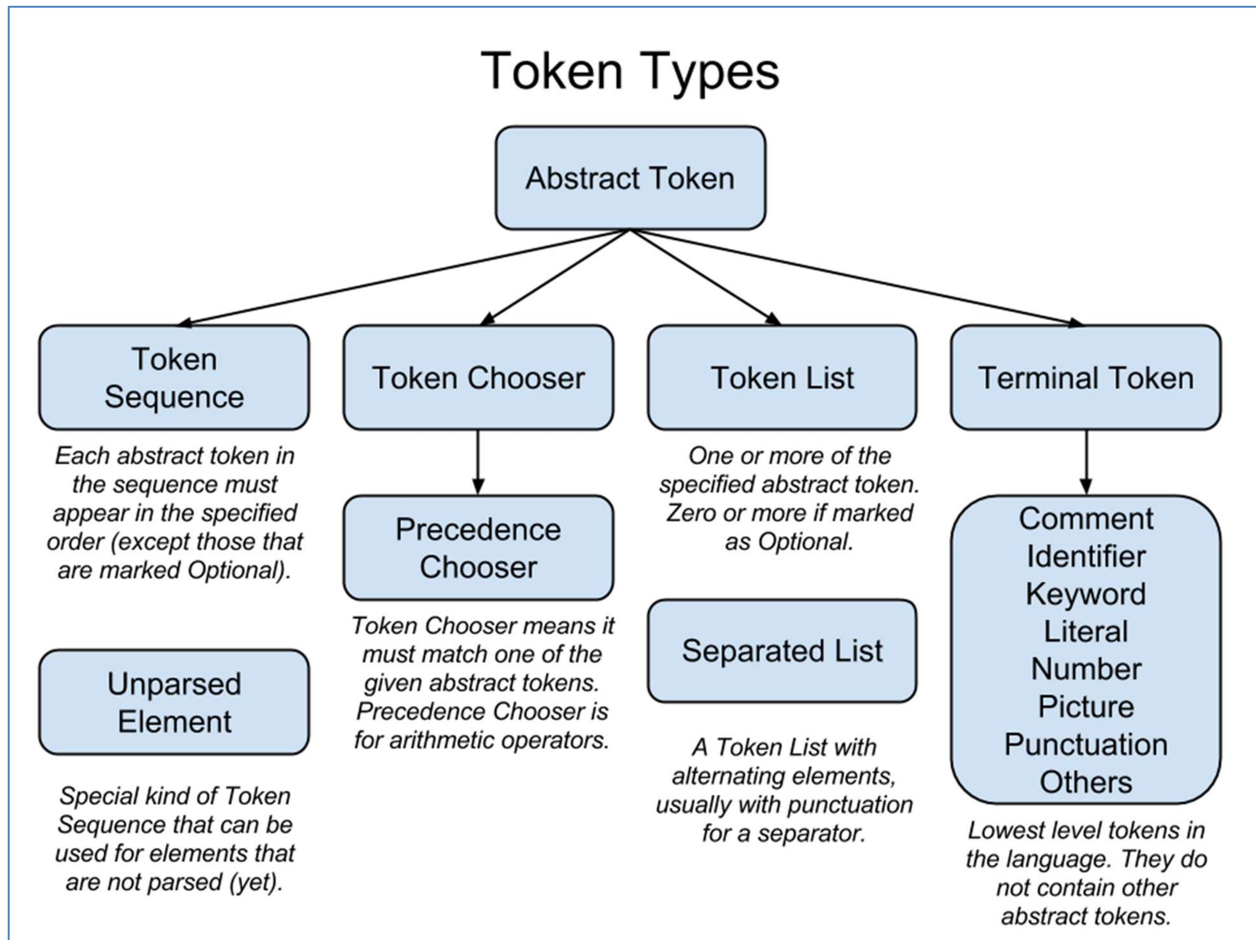

Figure 16. Token Types

5.1.4. Terminal Programmar Tokens

In a Programmar, terminal tokens are the ones that actually read and consume characters from the source computer program. Programmars can introduce custom terminal tokens. The main categories of terminal tokens are listed in Figure 17.

Terminal Token	Description
Comment	Different types of comments in each language.
Identifier	Symbols, variables, labels, function names, class names, etc.
Keyword	"while", "for", "if", etc. Some keywords are reserved words.
Literal	String literals, usually inside single (') or double (") quotes.
Number	Integers and floating-point numbers. Binary and hex are separate.
Punctuation	Punctuation can be several characters long, such as "<<<".

Figure 17. Basic Terminal Token Types

These categories are also useful for color-coding program listings, as in section 6.1.3.

5.1.5. Source File Modifications

One of the core tenets of Eagle Legacy is that we never change source code. However, sometimes errors are encountered in source code that we wish we could repair and continue. To handle this, we support a 'repair' function, in the Project. It uses pattern matching on a specific line number in a specific file. For example, Figure 18 shows an extra semicolon in a line of COBOL that should be removed, as well as the call to repair it. The 'repair' method call replaces the pattern ";" with " " on line 184 of the file "MFOLE.CPY" in the top-level directory.

```
180:        01 DVTARGETDEVICE     TYPEDEF.
181:           03 DVTD-tdSize                  DWORD.
182:           03 DVTD-tdDriverNameOffset      WORD.
183:           03 DVTD-tdDeviceNameOffset      WORD.
184:           03 DVTD-tdPortNameOffset;       WORD.
185:           03 DVTD-tdExtDevmodeOffset      WORD.
186:           03 DVTD-tdData                  BYTE.
187:
--------------------------------------------------------------------------------
    repair("MFOLE.CPY", 184, ";", " ", "Extra semicolon");
```

Figure 18. Source Code Line Repair

The repair is only done to the in-memory copy of the source file, and does not alter the file on disk. History is kept of the change, as well as the reason for the change, for reporting purposes.

5.1.6. Unexpected Comments

Our parser does not pre-process source code to remove all Comments, as some other parsers require. Instead, we allow Comments to be matched and retained, but if an unexpected Comment is detected, it can be reported and skipped. This allows the best of both worlds – keep those comments you wish to keep while not requiring markers in every possible location that might have Comments.

5.1.7. Command Line Parser

There is a command line parser available. It has options to trace the parsing process (see section 5.3.1), to display the resulting PST, to write the output to an XML or Java file, etc. You can also direct it to parse all the files of a particular language in a Project, or just those that have not yet parsed successfully.

```
 EagleParser [-trace] [-dump] <proj> <lang> <file> [<xmlFile> | <javaFile>]
       <file> can be 'all' or 'failed' as well.
```

Figure 19. EagleParser Options

Figure 19 shows the main options to the EagleParser program. There are several additional command line tools available in both Windows and Linux formats.

5.1.8. Programmar Semantic Tree File Formats

The PST can be stored as an XML file or a Java file. We can also store it as binary as well, but it is not significantly faster than XML.

See Figure 20 for a slightly abbreviated snippet of the XML format. T=Token, TT=Token Type, N=Name, SC/SL/EC/EL are starting and ending / character and line, V=Value.

```
<EagleProgram Created="Sun Jan 24 05:53:18 EST 2016"
         Elapsed="960" Language="PHP" Steps="100025" Tokens="2430">
<T EC="0" EL="183" SC="1" SL="1" TT="PHP_Program">
 <T EC="0" EL="183" N="entry" SC="1" SL="1" TT="List">
  <T EC="0" EL="183" N="entry[0]" SC="1" SL="1" TT="PHP_Entry">
   <T EC="0" EL="183" N="entry[0]" SC="1" SL="1" TT="HTML_Program">
    <T EC="0" EL="183" N="elements" SC="1" SL="1" TT="List">
     <T EC="0" EL="3" N="elements[0]" SC="1" SL="1" TT="HTML_Element">
      <T EC="0" EL="3" N="elements[0]" SC="1" SL="1" TT="HTML_Tag">
       <T EC="1" EL="1" N="startTag" SC="1" SL="1" TT="HTML_Punctuation" V="<"/>
       <T EC="5" EL="1" N="tag" SC="2" SL="1" TT="HTML_Identifier" V="html"/>
       <T EC="6" EL="1" N="closer" SC="6" SL="1" TT="HTML_PunctuationChoice" V=">"/>
      </T>
     </T>
```

Figure 20. Sample Programmar XML Snippet

5.1.9. Unit Tests

Some tokens need additional testing, such as terminal tokens and arithmetic expression tokens. For example, most programming languages use a generic string literal parser. It can optionally support doubled quotes, escaped quotes, multi-line quotes, etc. Unit tests are designed to validate these different options. Similar tests are set up for other terminal tokens like numbers and multi-character punctuation.

Arithmetic expressions have unit tests set up that validate the precedence of the parsed expression. For example, "1 + 2 * 3" needs to have a value of 7, not 9, in most programming languages. Mechanisms are established for interpreting expressions and comparing the results, by implementing some special Java Interfaces.

```
    @Override
    public void addTests() {
        addTestExpr("Add1", "2 + 3 + 4", "9");
        addTestExpr("Add2", "2 + 3 * 4", "14");
        addTestExpr("Add3", "2 * 3 + 4", "10");
        addTestExpr("Subtract1", "27-10-1", "16");
        addTestExpr("Subtract2", "27-(10-1)", "18");
        addTestExpr("FiveSix", "five + six", "11");
    }
```

Figure 21. Sample Expression Unit Test

Figure 21 shows some simple expression unit tests for Java's addition and subtraction operators. The parser is called to produce a PST, and the PST is evaluated.

```
    @Test
    public void testSimple() throws Exception {
        String[] input = new String[] {
                "data n = 75;",
                "n = n + 10;",
                "print n + 15;",
        };
        Template_Program lang = new Template_Program();
        runTest(_interpreter, "testSimple", input, "100\n", lang);
    }
```

Figure 22. Sample Logic Unit Test

Figure 22 shows a sample unit test for logic testing. [Note that the "Template" language is not a real programming language – it was established as a simple template for new Programmars.] The addition and subtraction operator definitions in the Programmar are implemented as shown in Figure 23.

```
public class Template_Additive_Expr extends PrecedenceOperator
          implements EagleRunnable {
  public @S(10) Template_Expr left = new Template_Expr(this, Prec.ATLEAST);
  public @S(20) Template_Puncts oper = new Template_Puncts("+", "-");
  public @S(30) Template_Expr right = new Template_Expr(this, Prec.HIGHER);

  @Override
      public void interpret(Interpreter interpreter) {
          int leftValue = interpreter.getIntValue(left);
          int rightValue = interpreter.getIntValue(right);
          switch (oper.toString())
          {
          case "+":
              interpreter.pushInt(leftValue + rightValue);
              break;
          case "-":
              interpreter.pushInt(leftValue - rightValue);
              break;
          default:
              throw new RuntimeException("Unexpected operator: " + oper);
          }
      }
  }
```

Figure 23. Sample Expression Interpreter (slightly condensed)

The majority of the testing comes from parsing large collections of computer programs written in many different styles from many sources.

5.2. Eagle Projects

We have introduced the concept of a Project as a manager for source files to be parsed and analyzed. In addition, it provides several functions. Figure 24 shows a very simple Project. Associated with each Project is a simple JUnit test case that parses every source file, and generates some reports.

```
public class PPSM_Project extends EagleProject {
    @Override
    public String getName() {
        return "PPSM";
    }
    public PPSM_Project() {
        _artifactBase = ROOT + "/Samples/PPSM";
        _sourceBase = _artifactBase + "/src");
    }
}
```

Figure 24. Simple Project Definition (condensed)

5.2.1. Language Identification

Given a collection of source files, it is sometimes non-trivial to determine which language they are written in. RPG, for example, doesn't always distinguish the different language variations through a naming convention. In fact, many RPG programs don't use filename suffixes at all. It is up to the Project to assign a language to each file, including those that should not be parsed at all, like binary files. There are, of course, default suffix-to-language mappings.

5.2.2. Language Overrides

In some cases, the Project may have customized their compiler or used a pre-compiler to modify the language. In that case, the core language must be slightly modified to adapt. But we don't want to replicate the entire Programmar definition, so there is a mechanism in place to just override small pieces of the Programmar. This is sometimes referred to as the Robot's Leg problem[16], and Injection[17] is often the solution. We use a simple version of Injection to allow customizing a piece of a Programmar without modifying the whole Programmar.

5.2.3. Macro Processing

Some languages depend heavily on macros: C, PL/I, Assembler and COBOL for example. Please refer to section 7 for details on a language-by-language basis. Note that macros always retain a history of changes made as the macro is processed, so the original source code is still available for reports.

5.3. Programmar Debugging

Parsing, in general, is a complicated process. It can be thought of as a depth-first search through a collection of AST's to try to find a complete AST. This "forest of trees" structure is difficult to visualize and especially difficult to debug. Some parsers don't provide meaningful diagnostics in the event of an unsuccessful parse. Our parser shows the farthest point reached in the program, along with the surrounding lines, to help see the actual point of failure. Often, this is sufficient to tune the Programmar so the parser can succeed.

```
Parse failure of Samples\APAC\APACSRC8\DEBT\STMFMT.CBL at line 409 column 33
   404:              03  W15-LINES OCCURS 66.
   405:                  05  W15-CPI    PIC  9(02).
   406:                  05  W15-COLS   PIC  9(04).
   407:                  05  W15-DETAIL PIC  X(136).
   408:
   409:                                   STATEMENT LAYOUT
                                          ^
Deepest failure point (line:col)
   409:33    -> .COBOL.COBOL_DataDeclaration.COBOL_DataComment
   407:16    -> .COBOL.COBOL_DataDivision.COBOL_DataDeclaration
   404:12    -> .COBOL.COBOL_DataDivision.COBOL_DataDeclaration
   403:8     -> .COBOL.COBOL_DataDivision.COBOL_DataDeclaration
    84:8     -> .COBOL.COBOL_DataDivision.COBOL_DataDeclaration
    69:8     -> .COBOL.COBOL_DataDivision.COBOL_WorkingStorageSection
    69:8     -> .COBOL.COBOL_DataDivision.COBOL_DataSection
    62:8     -> .COBOL.COBOL_DataDivision
     1:7     -> .COBOL.COBOL_Program_Fixed_Format
```

Figure 25. Sample Parse Failure (slightly condensed)

Figure 25 shows a sample Parse failure, which failed on the unexpected word "STATEMENT".

5.3.1. Parser Tracing

Sometimes, however, it is necessary to watch the parser in action. The simplest form of this is tracing. Two types of trace output are supported. The traditional grammar might be Figure 26, for a trivial language we invented called PPSM[18].

---

[16] Characterized as "I love your Robot, but I want to use my own Robot Leg, not yours"
[17] For example, see tutorialspoint.com/spring/spring_dependency_injection.htm
[18] See Post-Production Software Management, for a paper we submitted

```
<PPSM_Program> ::= <PPSM_Element>+;
<PPSM_Element> ::= <PPSM_First> [<PPSM_Second>];
<PPSM_First> ::= 'a' | 'b';
<PPSM_Second> ::= 'c' | 'd' | 'e';
```

Figure 26. PPSM Traditional Grammar

The Programmar equivalent is Figure 27.

```
public class PPSM_Program extends EagleLanguage {
  public @S(10) TokenList<PPSM_Element> elements;
}
public class PPSM_Element extends TokenSequence {
  public @S(10) KeywordChoice first = new KeywordChoice("a", "b");
  public @S(20) @OPT KeywordChoice second = new KeywordChoice("c", "d", "e");
}
```

Figure 27. PPSM Programmar

The sample file called p2.ppsm has just one line: "a b c".

```
File: Eagle\Samples\PPSM\src\p2.ppsm
Lang: PPSM (com.eagle.programmar.PPSM.PPSM_Program)

Depth  Syntax   Line  Char   Next     Pattern
=====  ======   ====  ====  ======= ========================================
    1  PPSM        1     1  a b c   . ? PPSM_Program
    2  PPSM        1     1  a b c   . . ? PPSM_Element
    3  PPSM        1     1  a b c   . . . ? PPSM_KeywordChoice (a)
    3              1     3  b c     . . . ***** Matched PPSM_KeywordChoice (a)
    3  PPSM        1     3  b c     . . . ? PPSM_KeywordChoice (c)
    3  PPSM        1     3  b c     . . . Failed PPSM_KeywordChoice (c)
    2              1     3  b c     . . ***** Matched PPSM_Element
    2  PPSM        1     3  b c     . . ? PPSM_Element
    3  PPSM        1     3  b c     . . . ? PPSM_KeywordChoice (a)
    3              1     5  c       . . . ***** Matched PPSM_KeywordChoice (b)
    3  PPSM        1     5  c       . . . ? Terminals.PPSM_KeywordChoice (c)
    3              2     1  (EOF)   . . . ***** Matched PPSM_KeywordChoice (c)
    2              2     1  (EOF)   . . ***** Matched PPSM_Element
    2  PPSM        2     1  (EOF)   . . ? PPSM_Element
    2              2     1  (EOF)   . ***** Matched PPSM_Program

Parsed PPSM Eagle\Samples\PPSM\src\p2.ppsm in 16 ms. Parse steps = 8.
   1 line, about 60 lines per second
```

Figure 28. Sample Trace Output

Figure 28 shows the trace from parsing the simple program. Each "?" represents a parsing "step" and should be read as "Can I match this token now?". This output is also available in HTML form, with clickable labels to expand and contract the search tree.

5.3.2. Parser Debugger

If tracing is not sufficient, an interactive debugger is available. It provides single-step functions (step-in, step-over and step-out) as well as breakpoints. Each "step" roughly corresponds to one line in the trace shown in the previous section. Figure 29 shows a sample screen from the debugger.

This is a Java Swing application at the moment. Conversion to a web-based application is under consideration. Also, converting it to an Eclipse plug-in is being considered.

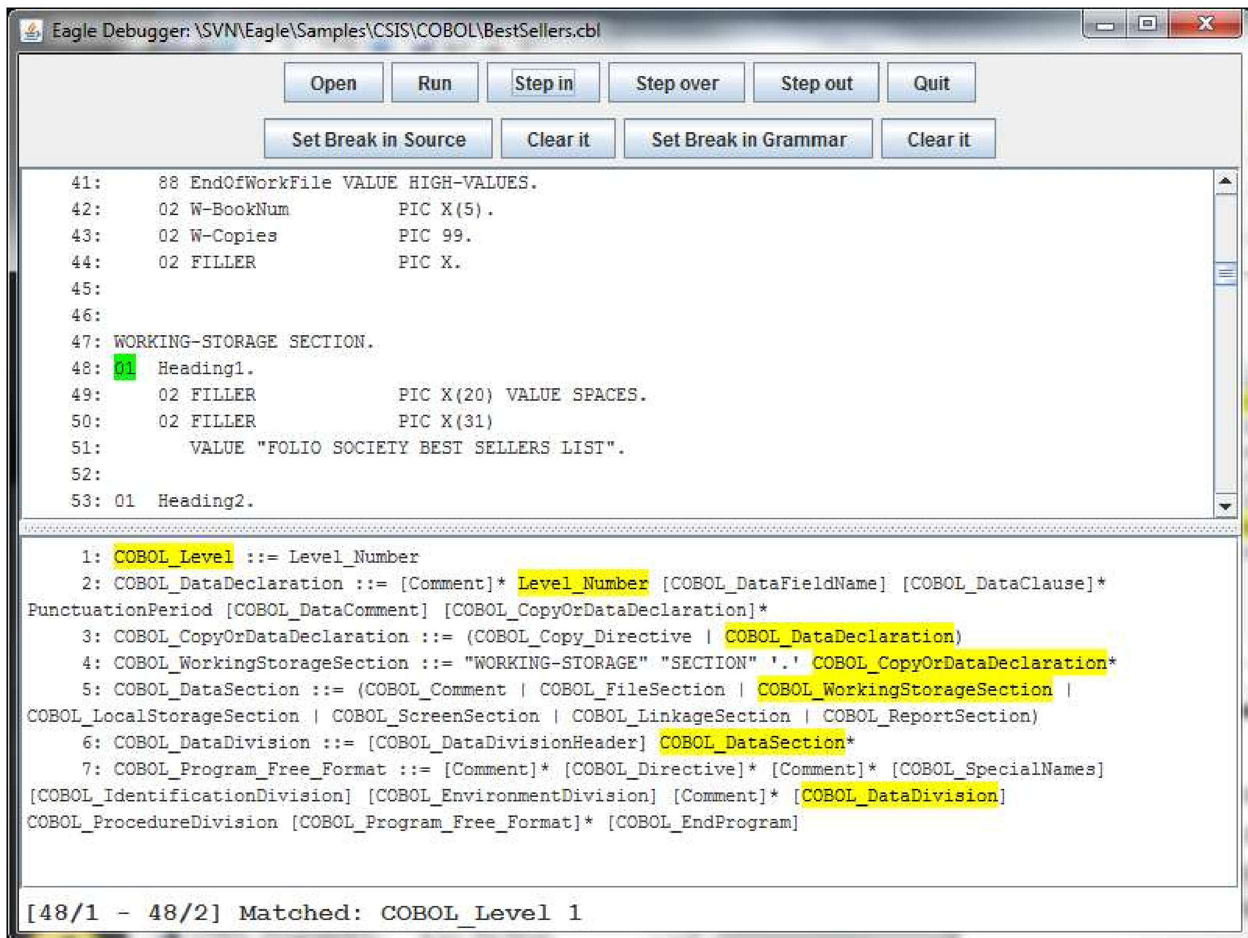

Figure 29. Sample Debugging Session

Source code is shown on the top panel, and the progress through the Programmar is shown in the bottom panel. It has just successfully matched the `COBOL_Level` token (highlighted on line 48, on the top) and will check for a `COBOL_DataFieldName` next (line 2 in the Programmar, on the bottom). The line at the bottom of Figure 29 includes start and ending line numbers and character positions for the matched token (line 48, columns 1-2).

5.4. "Pretty Printing"

As a programmer back in the 1970's, I encountered another programmer who believed that COBOL Paragraphs should really be written in paragraph form. His programs were completely unreadable, regardless of his claim that the compiler was just a little bit faster. For whatever reason, it is often useful to be able to parse a program, and then print it back out again in accordance with some formatting rules.

Google's search engine[19] is invoked billions of times each day. Immense effort is made to save every possible byte on downloading search results to users, including HTML, CSS and Javascript code. Take two aspirin and do a "View Source" in a browser some time when looking at some Google search results (or look at Figure 31).

[19] google.com

```
public class Javascript_TryStatement extends TokenSequence {
  public @S(10) @NEWLINE Keyword TRY = new Javascript_Keyword("try");
  public @S(20) @INDENT PunctuationLeftBrace leftBrace;
  public @S(30) @OPT TokenList<Javascript_StatementOrComment> statements;
  public @S(40) @OUTDENT PunctuationRightBrace rightBrace;
  public @S(50) @OPT TokenList<Javascript_CatchBlock> catchBlocks;
  public @S(60) @OPT Javascript_FinallyBlock finallyBlock;

  public static class Javascript_CatchBlock extends TokenSequence {
    public @S(10) @NEWLINE Keyword CATCH = new Javascript_Keyword("catch");
    public @S(20) PunctuationLeftParen leftParen;
    public @S(30) @NOSPACE Javascript_Variable_Definition id;
    public @S(40) @NOSPACE PunctuationRightParen rightParen;
    public @S(50) Javascript_Statement catchStatement;
  }

  public static class Javascript_FinallyBlock extends TokenSequence {
    public @S(10) @NEWLINE Keyword FINALLY = new Javascript_Keyword("finally");
    public @S(20) Javascript_Statement finallyStatement;
  }
}
```

Figure 30. Programmar for Javascript "try" Block

Using Java Annotations, we can "pretty print" programs. Note the usage of `@NEWLINE`, `@INDENT`, `@OUTDENT` and `@NOSPACE` in Figure 30.

5.4.1. Before and After Pretty Printing

```
<!doctype html><html itemscope="" itemtype="http://schema.org/WebPage"><head><meta content="/images/google_favicon_128.png" itemprop="image"><meta content
window.google={kEI:"hbvQUpKoNJD2oASzwYDQCg",getEI:function(a){for(var b;a&&(!a.getAttribute||!(b=a.getAttribute("eid")));)a=a.parentNode;return b||google.
d.onabort=function(){delete f[e]};f[e]=d;c||-1!=b.search("&ei=")||(g="&ei="+google.getEI(h));c=c||"/"+(k||"gen_204")+"?atyp=i&ct="+a+"&cad="+b+g+"&zx="+go
pm:"p",u:"c03934fb"},Toolbelt:{},y:{},x:function(a,b){google.y[a.id]=[a,b];return!1},load:function(a,b,c){google.x({id:a+l++},function(){google.load(a,b,c
window.chrome||(window.chrome={});window.chrome.sv=2.00;window.chrome.searchBox||(window.chrome.searchBox={});var m=function(){var a=encodeURIComponent(wi
google.arwt=function(a){a.href=document.getElementById(a.id.substring(1)).href;return!0};</script><script>(function(){'use strict';var f=this,h=Date.now||
w&&3==e)}g&&(b="clickkey")}for(g=e=a.srcElement||a.target;g&&g!=this;g=g.__owner||g.parentNode){var n=g,k;var l=n;k=b;var m=l.__jsaction;if(!m){m={};l.__j
(k="clickonly");k=(m=m[k])?{o:k,action:m}:void 0;if(k){b={eventType:k.o,event:a,targetElement:e,action:k.action,actionElement:n,timeStamp:h()};break e}}b=
try{google.pt=window.chrome&&window.chrome.csi&&Math.floor(window.chrome.csi().pageT);}catch(d){}})();
</script><style id="ostyle">.gb_Ea{display:inline-block;padding:0 0 0 15px;vertical-align:middle}.gb_Ca{position:relative}.gb_q{display:inline-block;outli
function g(c){var d="undefined",a="1";if(c&&c.getElementById)if(typeof XMLHttpRequest!=d)a="2";else if(typeof ActiveXObject!=d){var b,e,f="MSXML2.XMLHTTP"
maybeRedirectForGBV(document,location,"2");window.rwt=function(a,g,h,m,n,i,c,o,j,d){return true};
(window['gbar']=window['gbar']||{})._CONFIG=[[[0,"www.gstatic.com","og.og.en_US.sKkAlMsNbYM.O","com","en","1",1,[3,2,".64.40.36.36.","r_cp.r_qf.","31216,3
try{
var ka;var ja;gbar_.m=this;gbar_.n=function(b,c){for(var d=b.split("."),e=c||gbar_.m,f;f=d.shift();)if(null!=e[f])e=e[f];else return null;return e};gbar_.
gbar_.da=function(b){var c=typeof b;if("object"==c)if(b){if(b instanceof Array)return"array";if(b instanceof Object)return c;var d=Object.prototype.toStri
else if("function"==c&&"undefined"==typeof b.call)return"object";return c};gbar_.ea=function(b){return void 0!==b};gbar_.fa=function(b){return"array"==(0,
ka=function(b,c,d){if(!b)throw Error();if(2<arguments.length){var e=Array.prototype.slice.call(arguments,2);return function(){var d=Array.prototype.slice.
gbar_.la=function(b,c){var d=Array.prototype.slice.call(arguments,1);return function(){var c=d.slice();c.push.apply(c,arguments);return b.apply(this,c)}};
gbar_.v=function(b){Error.captureStackTrace?Error.captureStackTrace(this,gbar_.v):this.stack=Error().stack||"";b&&(this.message=String(b))};(0,gbar_.u)(gb
gbar_.oa=function(b,c){for(var d=0,e=String(b).replace(/^[\s\xa0]+|[\s\xa0]+$/g,"").split("."),f=String(c).replace(/^[\s\xa0]+|[\s\xa0]+$/g,"").split(".")
gbar_.y=Array.prototype;gbar_.pa=gbar_.y.indexOf?function(b,c,d){return gbar_.y.indexOf.call(b,c,d)}:function(b,c,d){d=null==d?0:0>d?Math.max(0,b.length+d
gbar_.ra=gbar_.y.filter?function(b,c,d){return gbar_.y.filter.call(b,c,d)}:function(b,c,d){for(var e=b.length,f=[],g=0,h=(0,gbar_.q)(b)?b.split(""):b,l=0;
gbar_.ta=gbar_.y.reduce?function(b,c,d,e){e&&(c=(0,gbar_.s)(c,e));return gbar_.y.reduce.call(b,c,d)}:function(b,c,d,e){var f=d;(0,gbar_.qa)(b,function(d,h
gbar_.ya=function(b){b=String(b);if(/^\s*$/.test(b)?0:/^[\],:{}\s\u2028\u2029]*$/.test(b.replace(/\\["\\\/bfnrtu]/g,"@").replace(/"[^"\\\n\r\u2028\u2029\x
var Ba;gbar_.Aa=function(b,c,d){for(var e in b)c.call(d,b[e],e,b)};Ba="constructor hasOwnProperty isPrototypeOf propertyIsEnumerable toLocaleString toStri
gbar_.z=function(){};gbar_.A=function(b,c,d,e){b.a=[];c||(c=d?[d]:[]);b.b=""!=d?d:void 0;b.k=c;if(e)for(c=0;c<e.length;c++)b.k[e[c]]=b.k[e[c]]||[]};gbar_.
var Da=function(b){(0,gbar_.A)(this,b,"",[])};(0,gbar_.u)(Da,gbar_.z);var Ea=function(b){(0,gbar_.A)(this,b,"",[])};(0,gbar_.u)(Ea,gbar_.z);gbar_.Fa=funct
gbar_.F=function(){};(0,gbar_.t)("gbar_._DumpException",function(b){if(gbar_.Pa)throw b;(0,gbar_.F)(b)});var Ua;var Qa;Qa=function(){this.d=!1;this.b=[];t
(0,gbar_.t)("gbar.ldb",(0,gbar_.s)(gbar_.G.g,gbar_.G));gbar_.Va=function(){};gbar_.Va.prototype.cb=!1;gbar_.Va.prototype.X=function(){this.cb||(this.cb=!0
gbar_.ab=-1!=lb.indexOf("Mac");gbar_.bb=-1!=lb.indexOf("Win");gbar_.cb=-1!=lb.indexOf("Linux");gbar_.mb=!!eb()&&-1!=(eb().appVersion||"").indexOf("X11");n
t:{var pb="",qb;if(gbar_.hb&&gbar_.m.opera)var rb=gbar_.m.opera.version,pb="function"==typeof rb?rb():rb;else if(gbar_.ib?qb=/rv\:([^\);]+)(\)|;)/:gbar_.L
var Db;var Cb;gbar_.Bb=!gbar_.L||gbar_.L&&9<=gbar_.Ab;Cb=!gbar_.L||gbar_.L&&9<=gbar_.Ab;Db=gbar_.L&&!(0,gbar_.wb)("9");!gbar_.jb||(0,gbar_.wb)("528");gbar
gbar_.Eb=function(b,c){gbar_.K.call(this,b?b.type:"");this.d=this.b=this.target=null;this.keyCode=this.button=this.clientY=this.clientX=0;this.w=!1;this.a
gbar_.Eb.prototype.init=function(b,c){var d=this.type=b.type;this.target=b.target||b.srcElement;this.b=c;var e=b.relatedTarget;if(e){if(gbar_.ib){var f;t:
gbar_.Eb.prototype.stopPropagation=function(){gbar_.Eb.F.stopPropagation.call(this);this.a.stopPropagation?this.a.stopPropagation():this.a.cancelBubble=!0
var Hb;var Fb;Fb="closure_listenable_"+(1E6*Math.random()|0);gbar_.Gb=function(b){try{return!(!b||!b[Fb])}catch(c){return!1}};Hb=0;var Ib=function(b,c,d,e
```

Figure 31. Google Search Result (Before)

Figure 31 shows a typical search result, while Figure 32 shows the same search result, after pretty printing.

```
<!DOCTYPE html>
<html itemscope="" itemtype="http://schema.org/WebPage">
<head>
<meta content="/images/google_favicon_128.png" itemprop="image">
<meta content="origin" id="mref" name="referrer">
<title>
iguana - Google Search
</title>
<script>
  (
    function()
    {
      window.google =
      {
        kEI : "hbvQUpKoNJD2oASzwYDQCg", getEI : function(a)
        {
          for (var b; a && (! a.getAttribute || ! (b = a.getAttribute("eid")));)
          a = a.parentNode;
          return b || google.kEI
        }, https : function()
        {
          return "https:" == window.location.protocol
        }, kEXPI : "31216,3300036,3300104,3300130,3300137,3300161,3310366,3310597,3310870,3311943,4000116,4005878,4007278,4007661,4007830,4008067,4008133,
        {
          e : "31216,3300036,3300104,3300130,3300137,3300161,3310366,3310597,3310870,3311943,4000116,4005878,4007278,4007661,4007830,4008067,4008133,40081
        }, authuser : 0, ml : function()
        {

        }, kHL : "en", time : function()
        {
          return (new Date).getTime()
        }, log : function(a, b, c, h, k)
        {
          var d = new Image, f = google.lc, e = google.li, g = "";
          d.onerror = d.onload = d.onabort = function()
          {
            delete f[e]
```

Figure 32. Google Search Result (After), unedited

This process is somewhat tedious, because every Programmar must have Java Annotations on it to control the result. By default, every terminal element is printed with one space separating tokens.

5.5. Online Parser

On the main website (eaglegacy.com), there is an option to "Try the Parser". It brings up the screen as shown in Figure 33. It allows you to upload a program to see whether it can be parsed. This application is written in Amazon Web Services (AWS), using the Java variation. It is a fairly typical Http Servlet.

Return to: Eagle Main Screen

THIS SITE IS NOT SECURE. PLEASE ONLY PARSE PUBLIC FILES.

Sample Programmar parsing web application.

Organization:

User: **PW:**

Choose a language: *Auto*
**Auto* means use filename suffix to decide.*

Tab conversion (blank for none):
Use this to convert tabs to spaces.

Choose source file to upload: Choose File No file chosen

Parse *

** Contact us at help@eaglegacy.com for access.*

Figure 33. The Online Parser Screen

Please contact us for access credentials.

5.5.1. Successful Parse

Figure 34 shows the (truncated) output from a successful online parse, with the "Show parse tree?" option selected. The "Elapsed time" measurement is inside the Parser itself, not the total time.

```
Language = Java
Filename = Python_Expression.java
Date = Sat Jan 16 23:26:10 UTC 2016

Parse success! Elapsed time = 4443 ms, parse steps = 60322.

  Seq Parent Depth   SLn   SC   ELn   EC  Type                           Text
===== ====== ===== ===== ==== ===== ====  ============================== ==================================
    1      0     1     1    1   441    0  Java_Program
    2      1     1     1    1     2   50  TokenList<>
    3      2     2     1    1     1   54  Java_Comment                   // Copyright Eagle Legacy Modernization LL
    4      2     2     2    1     2   50  Java_Comment                   // Original author: Steven A. O'Hara, Nov
    5      1     2     4    1     6    0  Java_Package
    6      5     3     4    1     4    7  Java_Keyword                   package
    7      5     2     4    9     4   35  TokenList<>
    8      7     3     4    9     4   11  Java_Identifier                com
    9      7     3     4   12     4   12  PunctuationPeriod              .
   10      7     3     4   13     4   17  Java_Identifier                eagle
   11      7     3     4   18     4   18  PunctuationPeriod              .
   12      7     3     4   19     4   28  Java_Identifier                programmar
   13      7     3     4   29     4   29  PunctuationPeriod              .
   14      7     3     4   30     4   35  Java_Identifier                Python
   15      5     3     4   36     4   36  PunctuationSemicolon           ;
   16      1     1     6    1    37    0  TokenList<>
   17     16     2     6    1     7    0  Java_Import
   18     17     3     6    1     6    6  Java_Keyword                   import
   19     17     3     6    8     6   10  Java_Identifier                com
```

Figure 34. Online Parse Success (truncated)

5.5.2. Parse Failed

In the event of a parse failure, a screen like Figure 35 is shown. Note in this case, we tried to parse a COBOL program as if it was a Python program. It didn't get far.

```
Language = Python
Filename = APACTERM.CBL
Date = Sun Jan 17 03:44:38 UTC 2016

Parse failed: Parse failure. Stopped at line 1 col 7.
     1:        $set LINKCOUNT"384"
               ^
     2:        *****************************************************************
     3:        *                                                               *

Deepest failure point (line:col)
     1:7     -> .Python.Python_Statement.Python_Simple_Statement
     1:1     -> .Python.Python_Statement.Python_Statement_List
     1:1     -> .Python.Python_Statement.Python_StatementOrComment
     1:1     -> .Python.Python_Statement
     1:1     -> .Python.Python_Program

Contact: Eagle Legacy
```

Figure 35. Online Parse Failure

5.6. Macro Processing

There are a number of programming languages that rely heavily on macros or preprocessors. The aim of our organization is to support macro processing, but not to depend on it. This is very different than a compiler, as discussed in section 2.4. This makes parsing significantly more difficult in many cases, especially when symbol tables are indeterminate.

Most Assemblers, C/C++, PL/I, COBOL, Django, CMD and many others make multiple passes over the source code, where the output from the first pass is really just a slightly modified version of the source code. Three different approaches to macro processing are being utilized concurrently.

5.6.1. Macro Expansion

In C some macros have to be expanded in order to parse a file. For example, macros that contain semicolons or braces often require expansion. In section 8.1.2, we discuss in some detail how macros are processed in C (and C++). Briefly, it involves parsing the C program using a CMacro Programmar and then evaluating (running) the CMacro Programmar. It handles most macro functions and embedded macros.

5.6.2. Pre-processing

Some languages, like Assemblers, can produce very useful compilation reports. These reports are intended for human consumption, but they often contain very helpful hints on macros, addresses, etc. At this point, the best example of this is Java, section 8.4.1. The 'javap' program is part of the standard java distribution. It reads a .class file and produces a human-readable report. We have a JavaP Programmar in place to parse that report, and use the results for associating symbols with their fully qualified class names.

5.6.3. Leaving Code Alone

Each Project (as in section 5.2) can choose which macros to expand and which to leave intact. There is some risk to leaving macros intact because the parser may be required to accept non-standard syntax. An example of a commonly used macro in C that should be left alone is one that allocates memory (i.e., wraps a call to 'malloc' or 'calloc'). Expanding this kind of macro would only make Analysis even more difficult.

5.7. Interpretation

As an example of an Analysis tool, we have built interpreters for (currently) 29 different languages. None of these are complete, but all of them are designed to be expanded to cover much more of each language.

5.7.1. Source Programs

For each of the 29 languages, we have three different test programs, where possible.

First, we have an expression test program. There are several dozen expressions that check to see how expressions are evaluated. For example, is 5 / 4 equal to 1 or 1.25? Each language has its own rules for operator precedence, etc. Here are a few of the 36 expression tests from AWK:

```
addTestExpr("Div1", "201 / 5", "40.2");
addTestExpr("Div2", "201 % 5", "1");
addTestExpr("Conc1", "\"abc\" \"def\" \"ghi\" == \"abcdefghi\"", "true");
addTestExpr("Conc2", "\"abc\" five six \"7\" == \"abc567\"", "true");
addTestExpr("Rel1", "200 < 100 + 99", "false");
```

Figure 36. Sample AWK Expression Tests

Second, we have a control logic test program. How does each language support loops, etc. Did you know that a `DO / CONTINUE` loop is Fortran will always run at least once even if the test is always false? Also, how do you break out of a For loop in each language? Here is one of the 11 statement logic tests in Julia.

```
a = 20
x = 100
for i in 10:a-1
  global x = x + i
end
if x == 245
  ok += 1
else
  println("for01 failed")
end
```

Figure 37. Sample Julia Statement Logic Test

Third, we have a small Roman Numeral test program written in all 29 languages. It converts numbers to Roman Numerals and back and has some robust unit tests built in. These programs are several hundred lines long and cover many of the basic features in each language, such as arrays, calling functions, etc. The output from all 29 programs is identical to the following.

```
Converted 812 to DCCCXII as expected
Converted 43 to XLIII as expected
Converted 49 to XLIX as expected
*** Number must be in the range 1 to 3999, not 4000
*** Error converting 4000, got Error instead of MMMM
Converted CCCXIV to 314 as expected
Converted MCMXCIX to 1999 as expected
Converted CXXXI to 131 as expected
Converted III to 3 as expected
*** Invalid roman numeral: MMMM starting at 4
*** Error converting MMMM, got 0 instead of 4000
Validated all conversions from 1 through 3999
```

Figure 38. Roman Numeral Conversion Test Output

5.7.2. Verification

For all 29 languages, for each of the three different source test programs, we have verified that our interpreters behave correctly by running each of these programs in a real compiler or interpreter. This generally is done using online source playgrounds[20] like idecom.com, rextester.com and tio.run.

5.7.3. Dynamic Analysis

The real reason for writing all these interpreters is to support dynamic analysis. Many languages perform actions at runtime that cannot easily be predicted by viewing the code. It is often necessary to run the code to observe the actual behavior. Consider the three sample functions in Python (Figure 39) and some calls to them.

[20] wikipedia.org/wiki/Comparison_of_online_source_code_playgrounds

```
def plus(a,b):
  return a+b

plus(3.00,.4)
plus((3),(4))
plus([3],[4])
plus('3','4')
plus((3,),(4,))
```

```
def minus(a,b):
  return a-b

minus(3,(4))
minus((3),4)
minus((3),(4))
minus({3},{4})
minus(4_5,3_5)
```

```
def times(a,b):
  return a*b

times('3',4)
times(3,'4')
times((3,),4)
times(3,[4])
times(True,False)
```

Figure 39. Ambiguous Functions in Python

What is the output from each of the calls? Try it! You might be surprised at the results.

While running the programs, we collect metrics that in turn are used for Transformation (see Section 7). We collect metrics like:

- argument types when calling functions
- operand types to operators like + and *
- number of times a loop is "broken" out of
- range of values for a numeric assignment
- longest string for a string assignment

These metrics are used for Transformation. If there is still ambiguity (see Figure 39), an error is displayed. If the test cases are not comprehensive, there is no reliable solution and Transformation may fail.

5.7.4. Interactive Interpreter

Being able to run a parsed program is very powerful to help understand exactly what it is doing. In some cases, such as PL/I, there are virtually no available compilers without access to a mainframe computer. It is especially helpful to be able to run a parsed program one step at a time and monitor the values of all the variables.

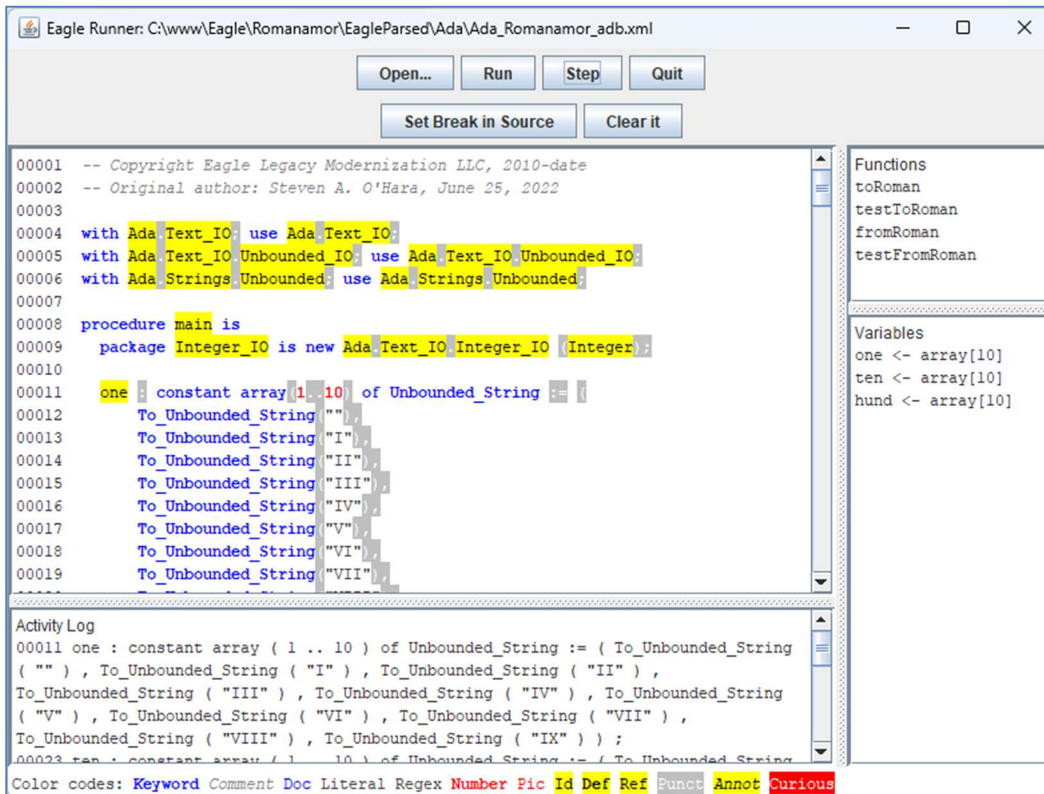

Figure 40. Sample Running an Ada Program

There is a command line tool to run the programs. We generally run all these programs using JUnit so we can easily verify results.

6. Visualization Tools

The tools in this section are generally reading csv files that were output from the Reporting tools, so it is easy to replace these reports. Tables can generally be sorted on other columns.

6.1. Project Browser

Being able to track multiple Projects concurrently is important for very large software repositories, especially when the Projects are millions of lines each. A full dashboard view is available, using flask[21] (Python version), as shown in Figure 41. Note that the Project names are obscured for privacy.

The information used for display is also available in CSV form (see section 6.3), so it can be loaded into Microsoft Excel (or similar) and manipulated, using Pivot Tables or similar for additional views.

[21] flask.palletsprojects.com

Project	Failed	Parsed	Percent	Lines	Parsed
A	17	2,349	99%	1,448,841	1,437,842
B	4	106	96%	7,767	7,431
b	-	4	100%	454	454
C	-	1	100%	42	42
C	-	5	100%	2,928	2,928
C	-	17	100%	2,181	2,181
C	-	62	100%	7,748	7,748
C	-	1,782	100%	753,079	753,079
D	-	162	100%	14,504	14,504
D	-	8	100%	1,712	1,712
E	-	3,448	100%	180,403	180,403
E	-	6,483	100%	1,501,611	1,501,611
G	-	1	100%	251	251
G	-	1	100%	65	65
IE	-	1	100%	909	909
L	-	15	100%	1,806	1,806
M	-	59	100%	12,575	12,575
C	-	409	100%	68,951	68,951
C	-	44	100%	6,846	6,846
P	-	4	100%	101	101
P	-	6	100%	350	350
P	-	40	100%	6,534	6,534
P	-	1	100%	1	1
P	-	22	100%	311	311
R	-	4	100%	680	680
S	-	74	100%	1,705	1,705
S	-	116	100%	32,100	32,100
S	-	21	100%	308	308
S	-	240	100%	14,006	14,006
S	150	790	84%	756,073	392,060
S	174	319	64%	6,463,752	4,699,197
S	-	179	100%	25,433	25,433
S	-	2,746	100%	731,924	731,924
TOTAL (33)	*345*	*19,519*	*98.26%*	*12,045,941*	*9,906,048*

Figure 41. Sample Dashboard Projects View

6.1.1. Browse One Project

Clicking on a Project's "Files" link reveals more details for that particular Project, for example Figure 42 is a partial listing for the “Dave” project which has 162 files total.

Dave Project

Seq	Directory	File Name	Suffix	Bytes	Date	Language	Lines	Parsed	PreFail	PostFail	Tokens	Tokens / Line	Elapsed (ms)	Lines / Sec	Parsing Steps	Steps / Token
1	family	add_marriage.php	php	2,664	2017-05-10	PHP	87	Yes			932	10.71	37	2,351	28,192	30.25
2	family	bday.php	php	4,710	2017-05-10	PHP	182	Yes			2,437	13.39	203	896	201,207	82.56
3	family	csv.php	php	2,014	2017-05-10	PHP	58	Yes			799	13.78	60	966	60,814	76.11
4	family	done_details.php	php	2,569	2017-05-10	PHP	77	Yes			1,107	14.38	81	950	84,531	76.36
5	family	done_marriage.php	php	2,629	2017-05-10	PHP	87	Yes			1,189	13.67	104	836	115,049	96.76
6	family	done_person.php	php	5,848	2017-05-10	PHP	171	Yes			2,465	14.42	179	955	197,499	80.12
7	family	edit_details.php	php	11,642	2017-05-10	PHP	332	Yes			4,443	13.38	415	800	450,880	101.48
8	family	edit_person.php	php	9,854	2017-05-10	PHP	283	Yes			3,430	12.12	236	1,199	199,186	58.07
9	family	index.php	php	5,386	2017-05-10	PHP	156	Yes			2,169	13.90	157	993	153,186	70.63
10	family	outlook.php	php	1,600	2017-05-10	PHP	49	Yes			647	13.20	48	1,020	53,678	82.96
11	family	txt.php	php	5,010	2017-05-10	PHP	171	Yes			2,578	15.08	285	600	317,590	123.19
12	originals	index0.html	html	14,541	2017-05-10	HTML	206	Yes			8,629	41.89	61	3,377	54,479	6.31
13	photos	add.php	php	4,486	2017-05-10	PHP	152	Yes			2,040	13.42	187	812	202,624	99.33
14	photos	count.php	php	2,154	2017-05-10	PHP	90	Yes			1,021	11.34	82	1,097	91,104	89.23
15	photos	countpix.php	php	4,586	2017-05-10	PHP	132	Yes			1,955	14.81	151	874	175,649	89.85

Figure 42. Sample View for One Project

For each project, a summary view is also available that shows information on a per-language basis, as shown in Figure 43, for the same "Dave" project as Figure 42.

Language	Files	Parsed	Bytes	Lines	Tokens	Tokens / Line	Elapsed (ms)	Lines / Sec	Parsing Steps	Steps / Line
AWK	22	22	10,934	475	7,434	15.65	129	3,682	71,966	151.51
CMD	73	73	61,439	1,595	24,482	15.35	171	9,327	126,639	79.40
CSS	1	1	243	19	113	5.95	4	4,750	1,065	56.05
HTML	3	3	14,823	217	8,730	40.23	63	3,444	54,885	252.93
Java	20	20	215,057	6,696	110,968	16.57	2,131	3,142	2,009,670	300.13
PHP	42	42	160,409	5,360	73,355	13.69	6,015	891	6,544,633	1221.01
SQL	1	1	5,667	142	2,177	15.33	72	1,972	8,057	56.74
Total (7)	*162*	*162*	*468,572*	*14,504*	*227,259*	*15.67*	*8,585*	*1,689*	*8,816,915*	*607.90*

Figure 43. Summary View for One Project

6.1.2. View Program File

Clicking on a "File Name" link yields a very simple view of the original file, as in Figure 44. This view is available whether the file was parsed successfully or not.

Seq	Line
1.	$ SET SOURCEFORMAT"FREE"
2.	IDENTIFICATION DIVISION.
3.	PROGRAM-ID. Iteration-If.
4.	AUTHOR. Michael Coughlan.
5.	
6.	DATA DIVISION.
7.	WORKING-STORAGE SECTION.
8.	01 Num1 PIC 9 VALUE ZEROS.
9.	01 Num2 PIC 9 VALUE ZEROS.
10.	01 Result PIC 99 VALUE ZEROS.
11.	01 Operator PIC X VALUE SPACE.
12.	
13.	PROCEDURE DIVISION.
14.	Calculator.
15.	PERFORM 3 TIMES
16.	DISPLAY "Enter First Number : " WITH NO ADVANCING
17.	ACCEPT Num1
18.	DISPLAY "Enter Second Number : " WITH NO ADVANCING
19.	ACCEPT Num2
20.	DISPLAY "Enter operator (+ or *) : " WITH NO ADVANCING
21.	ACCEPT Operator
22.	IF Operator = "+" THEN
23.	ADD Num1, Num2 GIVING Result
24.	END-IF
25.	IF Operator = "*" THEN
26.	MULTIPLY Num1 BY Num2 GIVING Result
27.	END-IF
28.	DISPLAY "Result is = ", Result
29.	END-PERFORM.
30.	STOP RUN.

Figure 44. Sample Program File View

6.1.3. View Successful Parse

If the file was successfully parsed, a color-coded version is displayed when clicking on the "Yes" link, as in Figure 45. Variable can be cross-referenced in this view and online documentation can be available for language-specific keywords.

Seq	Line
1.	$ SET SOURCEFORMAT"FREE"
2.	IDENTIFICATION DIVISION.
3.	PROGRAM-ID. Iteration-If.
4.	AUTHOR. Michael Coughlan.
5.	
6.	DATA DIVISION.
7.	WORKING-STORAGE SECTION.
8.	01 Num1 PIC 9 VALUE ZEROS.
9.	01 Num2 PIC 9 VALUE ZEROS.
10.	01 Result PIC 99 VALUE ZEROS.
11.	01 Operator PIC X VALUE SPACE.
12.	
13.	PROCEDURE DIVISION.
14.	Calculator.
15.	PERFORM 3 TIMES
16.	DISPLAY "Enter First Number : " WITH NO ADVANCING
17.	ACCEPT Num1
18.	DISPLAY "Enter Second Number : " WITH NO ADVANCING
19.	ACCEPT Num2
20.	DISPLAY "Enter operator (+ or *) : " WITH NO ADVANCING
21.	ACCEPT Operator
22.	IF Operator = "+" THEN
23.	ADD Num1, Num2 GIVING Result
24.	END-IF
25.	IF Operator = "*" THEN
26.	MULTIPLY Num1 BY Num2 GIVING Result
27.	END-IF
28.	DISPLAY "Result is = ", Result
29.	END-PERFORM.
30.	STOP RUN.

Figure 45. Successful Parse Display

6.1.4. View Failed Parse

If it was not parsed, an error message is displayed when clicking on the "No" link, as in Figure 46. The `COBOL_OccursClause` was expecting a `COBOL_Number`, not "no-of-chars".

```
TEST/TESTWIN.CBL

Parse failure of TEST/TESTWIN.CBL at line 9 column 37
     4:            78 note-width  value 41.
     5:            78 no-of-chars  value note-height * note-width.
     6:            01 note-window  pic x(10).
     7:            01 dummy    pic x.
     8:            01 note-data value all "- wallpaper ".
     9:              03 note-char pic x occurs no-of-chars.
                                          ^

Deepest failure point (line:col)
     9:37     -> .COBOL.Terminals.COBOL_Number
     9:30     -> .COBOL.COBOL_DataDeclaration.COBOL_OccursClause
     9:30     -> .COBOL.COBOL_DataDeclaration.COBOL_DataClause
     9:11     -> .COBOL.COBOL_DataDeclaration
     9:11     -> .COBOL.COBOL_DataDivision.COBOL_CopyOrDataDeclaration
     8:9      -> .COBOL.COBOL_DataDeclaration
     8:9      -> .COBOL.COBOL_DataDivision.COBOL_CopyOrDataDeclaration
     2:9      -> .COBOL.COBOL_DataDivision.COBOL_WorkingStorageSection
     2:9      -> .COBOL.COBOL_DataDivision.COBOL_DataSection
     2:9      -> .COBOL.COBOL_DataDivision
     1:8      -> .COBOL.COBOL_Program_Fixed_Format
```

Figure 46. Parse Failed Display

6.2. Language Browser

In addition to a Project-specific browser, there is a language-specific browser. It provides details on a per-language basis, across multiple Projects, as shown in Figure 47.

Programmar	Failed	Parsed	Percent	Lines	Parsed
AWK	-	25	100%	564	564
Bash	-	6	100%	209	209
Basic	-	6	100%	286	286
BNF	-	5	100%	461	461
C	323	1,138	77%	7,225,201	5,096,973
CMD	5	243	97%	16,592	14,948
COBOL	17	2,413	99%	1,453,592	1,443,571
Cpp	-	11	100%	3,849	3,849
CSharp	-	320	100%	51,008	51,008
CSS	-	21	100%	1,764	1,764
Delphi	-	35	100%	5,127	5,127
Gupta	-	1	100%	65	65
HTML	-	345	100%	231,741	231,741
IBMASM	-	1	100%	909	909
IntelASM	-	1	100%	24	24
Java	-	5,604	100%	1,013,146	1,013,146
JavaP	-	6,483	100%	1,501,611	1,501,611
Javascript	-	23	100%	2,489	2,489
JSON	-	3	100%	9,100	9,100
Lisp	-	15	100%	1,806	1,806
MSSolution	-	5	100%	423	423
Natural	-	74	100%	1,705	1,705
Perl	-	5	100%	2,244	2,244
PHP	-	2,088	100%	122,381	122,381
PL/I	-	40	100%	6,534	6,534
Powershell	-	208	100%	13,585	13,585
PPSM	-	1	100%	1	1
Property	-	1	100%	22	22
Python2	-	275	100%	15,460	15,460
Python3	-	18	100%	1,451	1,451
RPG	-	4	100%	680	680
SQL	-	65	100%	352,009	352,009
VB	-	2	100%	268	268
XML	-	34	100%	9,634	9,634
TOTAL (34)	*345*	*19,519*	*98.26%*	*12,045,941*	*9,906,048*

Figure 47. Dashboard Languages Browser

Note that the number totals match with Figure 41.

There is also two summary reports for each language, providing information on a project-by-project basis, as shown in Figure 48. One summary report for files that parsed, one for files that did not parse. This is for the CMD language files that did parse, across all the Projects. Note that the Project names have been obscured.

Project	Files	Parsed	Bytes	Lines	Tokens	Tokens / Line	Elapsed (ms)	Lines / Sec	Parsing Steps	Steps / Line
A	110	110	288,188	12,430	154,010	12.39	1,183	10,507	1,259,371	101.32
D	73	73	61,439	1,595	24,482	15.35	171	9,327	126,639	79.40
O	15	15	4,273	112	1,511	13.49	14	8,000	9,444	84.32
S	24	24	9,742	340	4,756	13.99	41	8,292	26,706	78.55
S	1	1	138	8	91	11.38	1	8,000	689	86.13
S	13	13	13,807	405	7,512	18.55	44	9,204	43,825	108.21
S	7	7	2,247	58	822	14.17	16	3,625	4,789	82.57
Total (7)	*243*	*243*	*379,834*	*14,948*	*193,184*	*12.92*	*1,470*	*10,168*	*1,471,463*	*98.44*

Figure 48. Summary Languages Browser, by Project

6.2.1. Programmar Viewer

Programmars can be viewed in the old-style grammar production rule style, with the addition of counts and statistics, as shown in Figure 49 for the BNF Programmar. All the statistics are based on the test suite as shown in Figure 47.

Main program: BNF_Program

BNF syntax:

- Case sensitive? no
- Auto advance? yes (cannot use this if end-of-line matters)
- Continuation character: none (usually needed if end-of-line matters)
- Extra characters: none (allowed in identifiers)
- Multichar Punctuation: `::=`

<table>
<tr><th>Count</th><th colspan="2">Terminals:</th></tr>
<tr><td>409</td><td colspan="2"><code>BNF_Literal: quotes:'" escape:\ doubled:no multiline:no</code></td></tr>
<tr><td>191</td><td colspan="2"><code>BNF_Rule_Definition: An identifier</code></td></tr>
<tr><td>503</td><td colspan="2"><code>BNF_Rule_Reference: An identifier</code></td></tr>
<tr><th>Count</th><th>Statistics(*)</th><th>Tokens:</th></tr>
<tr><td>353</td><td>1.86 2.72(29%)</td><td><code>BNF_Expression: BNF_ExpressionTerm* [BNF_Alternation]*</code></td></tr>
<tr><td>1,074</td><td>(38% | 47% | 9% | 6%)</td><td><code>| BNF_ExpressionTerm: (BNF_Literal | BNF_Rulename | BNF_Group | BNF_Optional)</code></td></tr>
<tr><td>5</td><td>38.20</td><td><code>BNF_Program: BNF_Rule*</code></td></tr>
<tr><td>191</td><td>all all all all</td><td><code>BNF_Rule: BNF_Rule_Definition "::=" BNF_Expression ';'</code></td></tr>
<tr><td>275</td><td>all 1.52</td><td><code>BNF_Alternation: '|' BNF_ExpressionTerm*</code></td></tr>
<tr><td>99</td><td>all all all all</td><td><code>BNF_Group: '(' BNF_Expression ')' [("*"|"+")]</code></td></tr>
<tr><td>63</td><td>all all all</td><td><code>BNF_Optional: '[' BNF_Expression ']'</code></td></tr>
<tr><td>503</td><td>all all</td><td><code>BNF_Rulename: BNF_Rule_Reference [("*"|"+")]</code></td></tr>
</table>

Terminals = 3 (instances=1,103)
Tokens = 8 (instances=2,563)

(*) Statistics are shown in the same order as the Tokens.
Percentages are rounded; 'all' and 'none' mean 100% and 0% before rounding.
For lists, it shows the average number of occurrences, excluding empty lists.

Figure 49. Sample Grammar View

The statistics are useful for optimizing Programmars leading up to Transformation. Knowing how many instances of each pattern are present in the code base helps determine which patterns should be transformed automatically, rewritten, or manually transformed.

6.2.2. Programmar Instances

Clicking on a number in the Count column in Figure 49 shows all instances of that pattern throughout all the Projects, as shown in Figure 50. The "Name" column is the name of the field in the Java Programmar code. The subscript after the name is for Token Lists. SL, SC, EL and EC are starting/ending line/character in the program.

Token: /Lisp/Functions/Lisp_IfFunction

Project	File	Name	SL	SC	EL	EC
LandOfLisp	dice_of_doom_v1.lisp	body[0]	37	3	44	19
LandOfLisp	dice_of_doom_v1.lisp	exprs[2]	90	21	93	57
LandOfLisp	dice_of_doom_v1.lisp	body[1]	98	3	100	34
LandOfLisp	dice_of_doom_v1.lisp	exprs[4]	116	16	118	38
LandOfLisp	dice_of_doom_v1.lisp	exprs[2]	137	5	139	60
LandOfLisp	dice_of_doom_v1.lisp	exprs[1]	150	14	153	13
LandOfLisp	dice_of_doom_v1.lisp	exprs[2]	155	7	157	10
LandOfLisp	dice_of_doom_v1.lisp	exprs[2]	149	5	157	12
LandOfLisp	dice_of_doom_v1.lisp	exprs[2]	211	25	216	63
LandOfLisp	dice_of_doom_v2.lisp	body[0]	8	3	16	24
LandOfLisp	dice_of_doom_v2.lisp	exprs[2]	28	15	41	27
LandOfLisp	dice_of_doom_v2.lisp	exprs[2]	25	8	43	20

Figure 50. Programmar Instances

Clicking on the File link brings up the same screen as Figure 45, but with the token highlighted, as in Figure 51 for example.

LandOfLisp -- dice_of_doom_v2.lisp

Seq	Line
1.	(load "dice_of_doom_v1.lisp")
2.	(load "lazy.lisp")
3.	
4.	(defparameter *board-size* 4)
5.	(defparameter *board-hexnum* (* *board-size* *board-size*))
6.	
7.	(defun add-passing-move (board player spare-dice first-move moves)
8.	(if first-move
9.	moves
10.	(lazy-cons (list nil
11.	(game-tree (add-new-dice board player
12.	(1- spare-dice))
13.	(mod (1+ player) *num-players*)
14.	0
15.	t))
16.	moves)))
17.	
18.	(defun attacking-moves (board cur-player spare-dice)

Figure 51. Highlighted View of a Token

6.2.3. Programmar Specific Parse Results

Figure 52 shows a portion of the COBOL parsing failures, across all Projects. The most interesting columns are "PreFail" and "PostFail". These often give an indicator of where Programmars need attention.

File Name	Bytes	Date	Lines	Parsed	PreFail	PostFail
STMFMT.CBL	23,130	2017-05-10	479	No		STATEMENT
GLP054.CBL	42,061	2017-05-10	1260	No		BA16-DEBT.
STPBARC.CBL	41,508	2020-12-29	1371	No	ALUE ALL "	?.
STPENQP.CBL	12,567	2017-05-10	331	No		MOVE WS-SY
HelloWIN.cbl	22,256	2017-05-10	389	No	nuName to	null
Printl2.cbl	9,208	2017-05-10	179	No		PRT-INFO-1
printraw.cbl	11,649	2017-05-10	283	No		call "cob3
PRINTTST.CBL	21,491	2017-05-10	581	No	e 3 times	*>skip 3 l
RDPrnt2.cbl	7,625	2017-05-10	213	No	e x'1B' &	'(11U' &
TESTWIN.CBL	1,404	2017-05-10	30	No	pop-up	area is no
PRP003N.CBL	50,789	2017-05-10	1472	No	Dis	count Rate
RECSAL.CBL	37,834	2017-05-10	1007	No		ALTERNATE
SALMAN.CBL	69,186	2017-05-10	2100	No		SECTION 53
Listing16-4.cbl	2,400	2022-03-16	92	No	IS COMMON	INITIAL PR
Listing-19-3-drv.cbl	1,610	2022-03-16	63	No	c USAGE	OBJECT REF
Listing19-1.cbl	2,073	2022-03-16	63	No	ry USAGE	OBJECT REF
Listing19-3.cbl	3,101	2022-03-16	108	No	c USAGE	OBJECT REF
	359,892		*10,021*	*0*		

Figure 52. COBOL Parse Failures

6.2.4. Programmar Semantic Tree Listing

Figure 53 shows a PST listing, which is essentially just a direct representation of what is in the parse output XML file. It is equivalent to the EagleShowTree program as well as the -dump option on the command line EagleParser (see section 5.1.7). Only the first 11 of 151 lines are shown.

Grammar_EBNF.bnf

```
Project  = bnf
Language = BNF
Parsed   = Thu Oct 02 23:35:16 CDT 2025
Elapsed  = 18 ms to parse
Steps    = 333 to parse
Tokens   = 151 in parse tree
```

Seq	Depth	SL	SC	EL	EC	Type	Value
1	1	1	1	8	0	BNF_Program	
2	2	1	1	8	0	List	
3	3	1	1	2	0	BNF_Rule	
4	4	1	1	1	12	BNF_Rule_Definition	ebnf-program
5	4	1	14	1	16	BNF_Punctuation	::=
6	4	1	18	1	27	BNF_Expression	
7	5	1	18	1	27	List	
8	6	1	18	1	27	BNF_ExpressionTerm	
9	7	1	18	1	27	BNF_Rulename	
10	8	1	18	1	26	BNF_Rule_Reference	ebnf-rule
11	8	1	27	1	27	BNF_PunctuationChoice	*

Figure 53. Sample PST Listing

This screen can be accessed from the Tokens column on either the Projects screen (section 6.1.1) or the Programmar screen (section 6.2.3).

6.3. FileInventory.csv

In Figure 54, a few lines from a Project's FileInventory.csv file are shown (in Excel). This data is collected by each Project unit test, and provides most of the data for the Visualization tools in this section.

Top Dir	Directory	File Name	Suffix	Bytes	Date	Type	Lines	Parsed	Tokens	Elapsed	Steps	PreFail	PostFail
Tools	Tools/src/com/eag	COBOL_Transform_Stop.j	java	618	10/03/2015	Java	18	Yes	210	0	1845		
Tools	Tools/src/com/eag	EagleEnvironment.java	java	866	11/04/2015	Java	36	Yes	353	0	4206		
Tools	Tools/src/com/eag	EagleFile.java	java	2290	11/06/2015	Java	113	Yes	860	32	10886		
Tools	Tools/src/com/eag	EaglePath.java	java	2044	12/23/2015	Java	86	Yes	873	32	13028		
Tools	Tools/src/com/eag	EagleUtilities.java	java	2665	12/15/2015	Java	125	Yes	1621	47	20924		
Tools	Tools/src/com/eag	FileCopy.java	java	561	09/30/2015	Java	26	Yes	274	16	2688		
Tools	Tools/src/com/eag	Eagle_XRef.java	java	1805	11/06/2015	Java	79	Yes	922	32	11779		
Tools	Tools/src/com/eag	Java_XRef.java	java	10817	12/03/2015	Java	308	Yes	4532	94	47551		
Tools	Tools/test	all.bat	bat	636	11/17/2014	CMD	22	Yes	179	16	900		
Tools	Tools/test	bad_imports.bat	bat	678	09/25/2011	CMD	26	Yes	368	15	1658		
Tools	Tools/test	check_copyrights.bat	bat	176	12/26/2011	CMD	4	No				@for /r	..\\src %%i
Tools	Tools/test	COBOL_grammar.html	html	120837	09/08/2013	HTML	1910	No					. &nb
Tools	Tools/test	CommentRemover.bat	bat	171	04/12/2014	CMD	7	Yes	67	0	282		
Tools	Tools/test	CreateReports.bat	bat	328	07/26/2011	CMD	12	Yes	111	0	520		
Tools	Tools/test	CreateReports.sh	sh	240	07/06/2015								
Tools	Tools/test	CrossReference.sh	sh	164	11/06/2015								
Tools	Tools/test	DumpClass.bat	bat	206	10/15/2015	CMD	8	Yes	82	16	391		
Tools	Tools/test	DumpTree.bat	bat	156	01/03/2016	CMD	5	Yes	61	0	247		
Tools	Tools/test	DumpTree.sh	sh	132	07/20/2015								

Figure 54. FileInventory.csv Excerpt

Loading this file into Excel (or similar) allows much more flexibility for Reporting and Visualization purposes, by using Pivot (or similar) tables and charts.

6.4. Symbol Definitions and References Visualization

This section is a discussion on visualization only. Other Analysis tools are responsible for connecting References up to the corresponding Definition, and persisting those into the PST. This can be a challenging process and is still undergoing active research. In the meantime, some default rules are in place. The complexity enters from Scoping rules.

Figure 55 shows a small sample file. The symbol Definitions are highlighted in gray. Clicking on a Definition shows a screen like Figure 56.

The References are links to the Definitions. Clicking on a Reference takes you to the Definition.

Seq	Line
1.	ebnf-program ::= ebnf-rule*;
2.	ebnf-rule ::= identifier '::=' ebnf-alternation ';';
3.	ebnf-alternation ::= ebnf-expression ('|' ebnf-expression)*;
4.	ebnf-expression ::= (literal | identifier [ebnf-modifier] | ebnf-optional | ebnf-group)+;
5.	ebnf-group ::= '(' ebnf-alternation ')' [ebnf-modifier];
6.	ebnf-optional ::= '[' ebnf-alternation ']' [ebnf-modifier];
7.	ebnf-modifier ::= '+' | '*';

Figure 55. Sample BNF file

It is interesting to see an eBNF grammar for eBNF that came from a Programmar.

Scope	Name	Location	Type	References
0-7	ebnf-alternation	3/1	Rule	2/32 5/20 6/23
0-7	ebnf-expression	4/1	Rule	3/22 3/44
0-7	ebnf-group	5/1	Rule	4/78
0-7	ebnf-modifier	7/1	Rule	4/45 5/42 6/45
0-7	ebnf-optional	6/1	Rule	4/62
0-7	ebnf-program	1/1	Rule	
0-7	ebnf-rule	2/1	Rule	1/18

Figure 56. Sample Symbol Table

In Figure 56, there are five columns.

"Scope" represents a range of line numbers (zero based). Elements in a Programmar that implement the Scope Interface can maintain a list of symbols defined within that scope. For example, brackets in Java can introduce a newly scoped local variable, but an “if” statement in Java without braces cannot.

The "Name" is the Symbol Definition. Clicking on it will take you directly to the Definition, as in Figure 55. Within a given scope, Symbol Definitions should be unique. The "Location" is the line number and character position of the Symbol Definition (and possibly the file name if it is defined elsewhere).

The "Type" of a Symbol should be kept fairly simple. Values like "Function", "Variable", "Label", "Class", etc. are expected. It is not meant to be an Abstract Data Type[22].

"References" are a list of links for all the known References to each Symbol. Line number and character position (one based).

[22] en.wikipedia.org/wiki/Abstract_data_type

6.5. Functional Tests

The Visualization tools are dynamic so functional tests have been implemented to validate the embedded links. Selenium[23] is used within the context of JUnit for running the tests. The tests navigate through each of the main screens. This section shows a subset of the actual tests.

6.5.1. Test Project, no Directory

This test finds the LandOfLisp Project on the main browser page, loads the Project and verifies that there were 15 parsed files in that Project, and there are a total of 1,806 lines in the Project. Then it verifies that the Lisp program called "lazy.lisp" has 71 lines in it and that the first line is the "defmacro" shown in Figure 57.

```
    gotoProject("LandOfLisp", "15", "1,806");
    gotoProjectFile("", "lazy.lisp", "Lisp", "71");
    gotoParsed("", "lazy.lisp", "(defmacro lazy (&body body)");
    gotoListing("", "lazy.lisp", "(defmacro lazy (&body body)");
    gotoProjectFile("", "guess.lisp", "Lisp", "18");
    gotoDump("", "guess.lisp", "1", "1", "19", "0", "Lisp_Program");
```

Figure 57. Project Browser Function Test

6.5.2. Test Project, with Directory

This test is similar, but also checks directory names and verifies that the parse failure message exists.

```
    gotoProject("APAC");
    gotoProjectFile("BATCH/BATCH16", "APAC.BAT", "CMD", "45");
    gotoParsed("BATCH/BATCH16", "APAC.BAT", "copy apac\\apacmenu.cbl");
    gotoListing("BATCH/BATCH16", "APAC.BAT", "copy apac\\apacmenu.cbl");
    gotoProjectFile("BATCH/BATCH16", "APACBACK.BAT", "CMD", "283");
    gotoFailed("BATCH/BATCH16", "APACBACK.BAT");
```

Figure 58. Project Browser Directory Test

6.5.3. Test Language, no Directory

This test finds the Lisp language on the main browser page, loads the grammar for that language, verifying that there are 15 parsed Lisp files. Then it verifies that the average number of top-level expressions per Lisp program is 19.87. Then it verifies that there is a Lisp program called "lazy.lisp" with 71 lines, and that the first line is the "defmacro" shown in Figure 59.

```
    gotoLanguage("LandOfLisp", "Lisp", "15");
    gotoGrammar("Lisp", "Lisp_Program", "19.87");
    gotoProgramFile("", "lazy.lisp", "71");
    gotoParsed("", "lazy.lisp", "(defmacro lazy (&body body)");
    gotoProgramFile("", "guess.lisp", "18");
    gotoDump("", "guess.lisp", "1", "1", "19", "0", "Lisp_Program");
```

Figure 59. Lisp Browser Functional Test

6.5.4. Test Language, with Directory

This test is similar, but verifies directory names. It also verifies that a parse failure message is available for a file (APACBACK.BAT) that is known to fail parsing, as shown in Figure 60.

[23] www.seleniumhq.org

```
gotoLanguage("APAC", "CMD");
gotoProgramFile("BATCH/BATCH16", "APAC.BAT", "45");
gotoParsed("BATCH/BATCH16", "APAC.BAT", "copy apac\\apacmenu.cbl");
gotoProgramFile("BATCH/BATCH16", "APACBACK.BAT", "283");
gotoFailed("BATCH/BATCH16", "APACBACK.BAT");
```

Figure 60. CMD Browser Functional Test

6.5.5. Test Symbol References

The test in Figure 61 follows the links for a symbol to all of its references; then it follows one of the reference links back to the original program.

```
final String line1 = "ebnf-identifier ::= lower ( ['-'] lower )+;";
gotoLanguage("Eagle", "BNF");
gotoProgramFile("Tools/bnf", "Grammar_EBNF.bnf", "7");
gotoParsed("Tools/bnf", "Grammar_EBNF.bnf", line1);
gotoSymbol("Tools/bnf", "Grammar_EBNF.bnf", "ebnf-identifier");
gotoReference("Tools/bnf", "Grammar_EBNF.bnf", line1);
```

Figure 61. Symbol Browser Functional Test

6.5.6. Test Grammar Tokens

When doing transformations, it is very useful to have an idea for the frequency of Programmar rules. In Figure 49, the first column has this frequency count, and it is a link to a screen like Figure 50. The test for this is shown in Figure 62.

```
gotoLanguage("Dave", "AWK", "51");
gotoGrammar("AWK", "AWK_Program");
gotoCounts(".AWK.AWK_Program", "utilities/captions.awk");
gotoListing("utilities", "captions.awk", "/.jpg/ || /.JPG/ {");
```

Figure 62. Grammar Token Functional Test

7. Transformation

7.1. Translating Human Languages

Translating human languages, e.g., French to Japanese, is fundamentally different that transforming computer languages

- Perfection is not required and not expected. As human readers of the translated text, we can easily tolerate errors in translation. Perfection is mandatory with programming languages.
- There are massive libraries of text corpora to assist with translation, and there are significant collections of parallel documents, such as the famous Rosetta Stone[24] that might assist. We rarely have computer programs written in many languages by the same author(s).
- Computers are remarkably good at translating and interpreting human languages, especially in limited domains such as travel assistants like Google Translate[25]. There are few, if any, broad spectrum programming language translators. Even relatively simple translators, such as VB6 to VB.Net[26] are scarce.

[24] wikipedia.org/wiki/Rosetta_Stone
[25] wikipedia.org/wiki/Google_Translate
[26] Many sources, such as learn.microsoft.com/en-us/answers/questions/1639102/tool-to-convert-vb6-application-to-net-core

7.2. Manual Transformation

It is always an option to simply rebuild old code repositories into shiny new architectures. Besides the obvious challenges of cost and time, there are often subtle behaviors embedded in the old code that are difficult to extract. Typically, the business logic is intertwined with display logic and is difficult to extract. Legacy software also is typically poorly documented and has limited automated testing.

A variation on this theme is to out-source the transformation to a third-party integrator.

7.3. Fully Automatic Transformation

Attempting a completely automated transformation comes in two flavors. In one case, the new language (C# for example) implements the functionality of old language (COBOL for example) directly. Looking at the new code, you would see objects with a Picture clause on them. No real attempt is being made to update the code – it is just being moved into a new framework. This approach offers a large advantage. If you can prove the C# version of COBOL's Picture clause is correct, and the same for all the other elements in COBOL, then you can, relatively, trust the new C# program. From there, you can slowly migrate components from COBOL-ish C# to actual C#.

The other approach is much more difficult. To determine the original author(s) intentions and translate the COBOL elements to C# as if it was written directly in C#. For example, COBOL programs often have to deal with dates and times. The language itself doesn't have much support for time zones, different calendars, localization, etc. In cases like this, a few hundred lines in COBOL can be replaced by just a few lines the utilize the .Net libraries.

7.4. Hybrid Approaches

Although not clearly defined, some form of automation is likely to be helpful in the transformation process. Perhaps some cleanup before transformation, certainly some cleanup afterwards. It is reasonable to expect different organizations to have different scenarios and different approaches.

7.5. Eagle Legacy Transformation

Rather than focusing on a single pair of languages (COBOL to C# for example), we have chosen to build a broad base of simple transformation examples. Reference section 5.7 for the 29 languages of the same small test programs. For 25 of those 29 languages, we have transformed the three test programs to three different target languages. Note that 4 languages are still in active development (Bash, CMD, Lisp and Assembler). These transformations are not perfect and not intended to provide complete coverage for each of those languages. Instead, we have created a framework for transformation and made sure we covered the basic elements of each language. For each language, as applicable, we cover:

- Primitive data types such as integers and strings
- Function calls and passing parameters
- For loops and while loops
- Arithmetic and logical operators, and operator precedence
- String formatting functions, such as substring and length

The output languages, C#, Java and Python are all implementations of an abstract class that could be instantiated for languages like Rust or Powershell.

7.5.1. Transformation Example, COBOL to C#

This small example is just a snippet of the COBOL Roman Numeral program. The complete example, and many more, are available at eaglegacy.com/Romanamor/index.html.

```
      ******************************************************************
      * Depends on num, returns actualRom
          to-roman.
            IF num LESS THAN 1 OR num GREATER THAN 3999
              DISPLAY "*** Number must be in the range 1 to 3999, not ",
                      num
              MOVE "Error" TO actual-rom
            ELSE
              COMPUTE a = num / 1000
              COMPUTE b = (num - 1000*a) / 100
              COMPUTE c = (num - 1000*a - 100*b) / 10
              COMPUTE d = num - 1000*a - 100*b - 10*c
              ADD 1 TO a, b, c, d
              MOVE SPACES TO actual-rom
              STRING FUNCTION TRIM(thou(a) TRAILING)
                     FUNCTION TRIM(hund(b) TRAILING)
                     FUNCTION TRIM(ten(c) TRAILING)
                     FUNCTION TRIM(one(d) TRAILING)
                INTO actual-rom
              END-STRING
            END-IF.
```

Figure 63. Sample COBOL Paragraph (Function)

There are, of course, many ways to implement functions like this in COBOL, using Linkages for example. The full COBOL program (Figure 63 is just a snippet) has been run using ideone.com. It was also run in the Eagle Legacy Interpreter to provide dynamic analysis statistics, which in turn were used during the transformation process. Transformation to C# generates the snippet as shown in Figure 64.

```
public static void to_roman()
{
  if (num < 1 || num > 3999)
  {
    System.Console.Out.WriteLine(
        "*** Number must be in the range 1 to 3999, not " + num);
    actual_rom = "Error";
  }
  else
  {
    a = num / 1000;
    b = (num - 1000 * a) / 100;
    c = (num - 1000 * a - 100 * b) / 10;
    d = num - 1000 * a - 100 * b - 10 * c;
    a += 1;
    b += 1;
    c += 1;
    d += 1;
    actual_rom = "";
    actual_rom = thou[a - 1].Trim() + hund[b - 1].Trim() +
        ten[c - 1].Trim() + one[d - 1].Trim();
  }
}
```

Figure 64. Generated C# Method

The code in Figure 64 is unedited, except two lines were too long and manually wrapped to fit.

7.5.2. Transformation Example, C to Python

This example is meant to show the side-by-side reports. Figure 65 and Figure 66 are viewable in html next to each other. As you move the cursor over one side, the element is highlighted in red on both sides at the same time. Not all elements are directly transformed, such as class definitions which are required in Java and C# and are scarce in COBOL and C.

```
 -   17 static char *toRoman(int num) {
 -   18   char temp[100];    // Probably 16 is plenty. Longest is MMMDCCCLXXXVIII (3888)
 -   19
 6   20   if (num < 1 || num > 3999) {
 -   21     printf("*** Number must be in the range 1 to 3999, not %d\n", num);
 -   22     return "Error";
 -   23   }
 9   24   int d = num % 10;
10   25   int c = (num / 10) % 10;
11   26   int b = (num / 100) % 10;
12   27   int a = (num / 1000);
 -   28
13   29   strcpy(temp, thou[a]);
14   30   strcat(temp, hund[b]);
15   31   strcat(temp, ten[c]);
16   32   strcat(temp, one[d]);
17   33   return strdup(temp);
 -   34 }
```

Figure 65. Left side (C) of Side-by-Side Report

```
 -    5 def toRoman(num):
20    6     if num < 1 or num > 3999:
20    7         print('*** Number must be in the range 1 to 3999, not ' + str(num))
20    8         return 'Error'
24    9     d = num % 10
25   10     c = (num // 10) % 10
26   11     b = (num // 100) % 10
27   12     a = (num // 1000)
29   13     temp = thou[a]
30   14     temp += hund[b]
31   15     temp += ten[c]
32   16     temp += one[d]
33   17     return temp
```

Figure 66. Right side (Python) of Side-by-Side Report

These reports are automatically generated during the transformation process.

8. Programmars

This section includes a discussion of Programmars for some of the major programming languages. None of these Programmars can be considered "perfect", if there is such a thing. The intention of this effort is to open-source these Programmars so that experts in those languages can help solidify them.

Please note that all of these programming languages are parsed with the same parser, with no pre-processing (other than optionally macros). All generate the same type of PST so information can be shared across different programming languages.

8.1. C / C++ Language

One of the most challenging languages to parse is C (and hence C++). It is worth noting that parsing for analysis is different than compiling, see section 2.4. This means we should still be able to parse a program even if some #include files are not available. And we should try to parse if we choose not to expand some (or all) #define macros.

8.1.1. External Dependencies

Creating a completely independent C source program is rare. Normally it depends heavily on other include files, source files, environment variables, and make files. It is common for a single source file to be compiled many different ways in the same Project.

8.1.2. Macros

Macro evaluation is traditionally done with a pre-processor. But we wanted to be able to parse programs even if all the dependencies are not present or known. It was very important to keep macro processing optional.

In particular, macro functions (i.e., #define with parameters), can be very challenging. Figure 67 shows some sample code with C macro functions in it. The backslashes (\) are continuation markers, needed in C macros, but not allowed in C.

```
#define ARRAY_CLASS_INC(nam,supr,styp)          \
    MATRIX_CLASS_INC(nam,supr,styp)             \
    int (*copyArray) (struct nam *tgt, struct nam *src, int indx); \

#define MATRIX_CLASS_INC(nam,supr,styp)                         \
    COLLECT_CLASS_INC(nam,supr,styp)                            \
    struct nam  *(*fetch) (Stream_t *stream, char *label, int withbrace); \
    int (*copyMat) (struct nam *tgt, struct nam *src, icoord_t coord); \
    struct nam  *(*mul) (struct nam *mat1, struct nam *mat2); \
        /* weighted sum. */     \
    float       (*wtSum) (struct nam *matrix, struct nam *weight); \

#define COLLECT_CLASS_INC(nam,supr,styp)                        \
    OBJ_CLASS_INC(nam,supr)                                     \
    int (*read) (struct nam *collect, Stream_t *input, char *label); \
    int (*write) (struct nam *list, Stream_t *outp, char *label, int ind); \
    styp   *(*searchNOCASE) (struct nam *collect, char *name); \

#define OBJ_CLASS_INC(nam,supr)                                 \
    char        *name;          /* Class name of object. */     \
    struct supr *super;         /* Pointer to the superclass */ \
    struct nam  *(*dup) (struct nam *obj);      /* Copy everything */   \
    int         (*apply) (struct nam *obj, va_list arglist);    \

struct Array_c
{
    ARRAY_CLASS_INC (Array_t, Matrix_c, Obj_t)
};
```

Figure 67. Sample Macro Function in C (slightly condensed)

Figure 68 shows the code from Figure 67 after macro processing. Note that history of macro changes are retained for reporting purposes. The result of pre-processing is sent to the Parser, but it retains all the information needed to show all the details of the pre-processing steps.

```
struct Array_c
{
    char        *name;
    struct Matrix_c *super;
    struct Array_t  *(*dup) (struct Array_t *obj);
    int         (*apply) (struct Array_t *obj, va_list arglist);
    int (*read) (struct Array_t *collect, Stream_t *input, char *label);
    int (*write) (struct Array_t *list, Stream_t *outp, char *label, int ind);
    Obj_t   *(*searchNOCASE) (struct Array_t *collect, char *name);
    struct Array_t  *(*fetch) (Stream_t *stream, char *label, int withbrace);
    int (*copyMat) (struct Array_t *tgt, struct Array_t *src, icoord_t coord);
    struct Array_t  *(*mul) (struct Array_t *mat1, struct Array_t *mat2);
    float       (*wtSum) (struct Array_t *matrix, struct Array_t *weight);
    int (*copyArray) (struct Array_t *tgt, struct Array_t *src, int indx);
};
```

Figure 68. After Macro Processing

Comments inside macros are particularly difficult to handle because one of the goals of our Analysis work is to retain as many comments as possible while parsing. In some sense, there are two concurrent languages in C, the C language and the C macro language. Sometimes the comments are intended for C and sometimes for C macros.

The actual methodology used for macro processing is to parse the original program using the C macro language, treating the C code as just text; then interpreting the C macro program during the parse process and generating new C code dynamically. The interpretation process is based on the same implementation as unit testing (described in section 5.1.9).

As is the case with many legacy programming languages, there are many constructs in the language that have evolved over time. Function parameters in pre-ANSI C[27] were allowed to be anonymous. Functions did not have to have a return type. Programmars used 'char *' instead of 'void *'. And so forth, all of which make C one of the more difficult languages to analyze.

8.2. COBOL

COBOL has a very long history, with a reputation as stated in Figure 69[28]. A general COBOL parser is quite interesting to implement due to the many variations of the language.

> *COBOL has been criticized throughout its life for its verbosity, design process and poor support for structured programming, which resulted in monolithic and incomprehensible programs.*

Figure 69. COBOL Derision

8.2.1. Free Format versus Fixed Format COBOL

In particular, free-format COBOL is very different from fixed-format COBOL. But we want all the elements of the language stored in the same manner, whether it is fixed or free format. To accomplish this, we provide an Abstract COBOL Language class with two subclasses. So there is only one implementation of the PERFORM verb, PICTURE clauses, etc. Figure 70 shows the top-level definition of

[27] en.wikipedia.org/wiki/C_(programming_language)#K.26R_C
[28] en.wikipedia.org/wiki/COBOL

the COBOL Language(s). This is a slightly simplified version of the COBOL_Program class. The two subclasses are virtually the same as the production version.

```
public abstract class COBOL_Program extends EagleLanguage {
    public COBOL_Program(String name, COBOL_Syntax syntax) {
        super(name, syntax);
    }

    public @OPT TokenList<COBOL_Directive> directives;
    public @OPT COBOL_IdentificationDivision identificationDiv;
    public @OPT COBOL_EnvironmentDivision environmentDiv;
    public @OPT COBOL_DataDivision dataDiv;
    public COBOL_ProcedureDivision procedureDiv;
    public @OPT COBOL_EndProgram endProgram;
}
public class COBOL_Fixed_Format extends COBOL_Program {
    public COBOL_Fixed_Format() {
        super("COBOL_Fixed_Format", new COBOL_Fixed_Format_Syntax());
    }
}
public class COBOL_Free_Format extends COBOL_Program {
    public COBOL_Free_Format() {
        super("COBOL_Free_Format", new COBOL_Free_Format_Syntax());
    }
}
```

Figure 70. Different COBOL Languages (simplified)

The two subclasses of COBOL_Syntax deal with column-specific information for the fixed format version.

8.2.2. COBOL Level Numbers

COBOL Level numbers are challenging to parse in such a way that the hierarchy is maintained. The rules are fairly simple. Higher numbers are deeper in the tree, same numbers are siblings, lower numbers go higher up in the tree. In order to implement this without post-processing, it is necessary to use context sensitivity. Each instance of a Level number has to consider the preceding Level number, which means examining the partially built PST.

```
002420 01  WS-DB-LINE.
002430     03  WS-TOP-LNE.
002440         05  WS-TCHR PIC X(01) OCCURS 80.
           03  WS-T-LINE REDEFINES WS-TOP-LNE.
               05  FILLER  PIC X(01).
               05  WS-H-LINE
                           PIC X(78).
               05  FILLER  PIC X(01).
002430     03  WS-TOP-LNE2.
002440         05  WS-TCH  PIC X(01) OCCURS 80.
           03  WS-TP-LINE2 REDEFINES WS-TOP-LNE2.
               05  FILLER      PIC  X(01).
               05  WS-TOP-COMP PIC  X(40).
001430         05  FILLER      PIC  X(23).
               05  WS-WRKHD    PIC  X(11).
```

Figure 71. Sample COBOL Level Numbers

Figure 71 shows an example of Level number hierarchy. It is difficult to parse this correctly without some logic, because it depends on *relative* level numbers. In this case, all four '03's go under the '01'. And each '05' goes below the preceding '03'. The logic used for this is "semantic" in the sense that it is context

sensitive. When the COBOL_Level terminal node parser is invoked, it searches the partially constructed PST to decide if this level number should be accepted in this slot.

8.2.3. Copybooks

The COPY directive in COBOL causes another file to be included inline. Like all Macros, we sometimes want to expand it, and sometimes we just want to leave it as is. Each Project has a mechanism to decide which macros to expand or not. As of this writing, COPY / REPLACING functionality has not yet been implemented, although it closely parallels C macros in section 8.1.2.

8.3. HTML

HTML is very difficult to parse such that structure is retained. One of the highlights of our parsing approach is the simplicity of embedding Programmars inside another Programmar.

8.3.1. Embedded Languages

A typical HTML file can contain Javascript, CSS, PHP or a number of other languages. PHP is actually layered on top of Perl. These compound languages are often difficult to deal with in a traditional grammar, but relatively simple in a Programmar.

```
public class HTML_Script extends TokenChooser {
    public class HTML_ScriptWithBody extends TokenSequence {
        public HTML_StartScript startScript;
        public @SYNTAX(Javascript_Syntax.class) Javascript_Program javascript;
        public HTML_EndScript endScript;

        public class HTML_StartScript extends TokenSequence {
            public HTML_Punctuation startTag = new HTML_Punctuation('<');
            public HTML_Keyword script = new HTML_Keyword("script");
            public @OPT TokenList<HTML_Attribute> attributes;
            public HTML_Punctuation endTag = new HTML_Punctuation('>');
        }
        public class HTML_EndScript extends TokenSequence {
            public HTML_Punctuation startTag = new HTML_Punctuation("</");
            public HTML_Keyword SCRIPT = new HTML_Keyword("script");
            public HTML_Punctuation endTag = new HTML_Punctuation('>');
        }
    }

    public class HTMLScriptNoBody extends TokenSequence {
        public HTML_Punctuation startTag = new HTML_Punctuation('<');
        public HTML_Keyword script = new HTML_Keyword("script");
        public @OPT TokenList<HTML_Attribute> attributes;
        public HTML_Punctuation endTag = new HTML_Punctuation("/>");
    }
}
```

Figure 72. HTML Script Definition (condensed)

As shown in bold in Figure 72, embedding another language involves two parts. Using a reference to the other language is simple in a Programmar. Changing syntax is necessary because each language has different syntactic rules (upper / lower case, end-of-lines, comments, etc.).

8.3.2. HTML Syntax is not Strict

HTML, as opposed to XHTML, is a very forgiving language.

Tables, in particular, are difficult to parse while retaining structure. Many elements, especially those that terminate rows or data cells, are optional. Because Tables often contain other Tables, retaining this structure while parsing is very helpful for follow-on Analysis tools. The </tr> and </td> elements are often omitted. In addition, the <thead>, <tbody> and <tfoot> sections are optional.

Some tags, such as the anchor tag <a> are unusual. When used with as they are not supposed to have an tag, and don't always use the <a name=…/> notation. But when used as , they are required to have an closing tag.

Other structures like '![CDATA]' and <pre> are challenging as well, and have been implemented in the HTML Programmar.

8.3.3. Django / PHP / JSP

Languages that appear to be embedded inside HTML are often really outer wrappers around HTML and are pre-processed before the HTML parser sees them. Languages like Django, PHP and JSP are all HTML code generators, which is very similar to macro processing in C (see 8.1.2). Frequently, it is not necessary to pre-process them in advance; it is often possible to parse these compound HTML files in a single pass.

```
<tbody>
<script type="text/javascript" charset="utf-8">
{%- for p in projects %}
  row1("{{ p.name }}", {{ p.files }}, {{ p.parsed }}, {{ p.lines }});
{%- endfor %}
</script>
</tbody>
```

Figure 73. HTML Mixed with Javascript and Jinja

Figure 73 shows some typical HTML code with embedded Javascript and using Jinja (similar to Django). The 'for' loop is processed by Jinja, a Python pre-processor. The 'row1' command is Javascript. The 'tbody' and 'script' commands are HTML. In our efforts, it is vital to retain all parts intact, and not to treat any of the parts as just text to be processed elsewhere.

8.4. Java

Modern languages like Java are designed to be parsed with context-free grammars, so the language itself is fairly simple.

The only sticking point with Java was defining class constructors, which have no return type. Since the compiler knows all the possible types at compile time, and the parser does not, it was difficult to separate constructors from ordinary method definitions. The solution was to use context-sensitive parsing and to look at the surrounding class name. If the method name matched the immediate class name, then it was determined to be a constructor and not a method definition.

8.4.1. Java Printer (JavaP)

The JavaP program (part of the standard JDK) reads in a .class file and prints some useful information, as in Figure 74. Of particular interest to us are the fully qualified class names. Processes have been built into our software to automatically run the JavaP program and generate a .javap report file. We then wrote a JavaP Programmar to parse the javap report files and extract the relevant information.

```
Compiled from "DeTabberTest.java"
public class com.eagle.tests.DeTabberTest extends junit.framework.TestCase
  SourceFile: "DeTabberTest.java"
  Constant pool:
const #1 = class #2;    //  com/eagle/tests/DeTabberTest
const #2 = Asciz com/eagle/tests/DeTabberTest;
  (--similar lines omitted--)
const #44 = Asciz       SourceFile;
const #45 = Asciz       DeTabberTest.java;
{
public com.eagle.tests.DeTabberTest();
  LocalVariableTable:
   Start  Length  Slot  Name   Signature
   0      5       0    this       Lcom/eagle/tests/DeTabberTest;
  Code:
   Stack=1, Locals=1, Args_size=1
   0:   aload_0
   1:   invokespecial  #8; //Method junit/framework/TestCase."<init>":()V
   4:   return
```

Figure 74. Partial JavaP Output

Since this output is intended for human consumption, not a computer, it varies from computer to computer. But it does consistently provide fully qualified class names.

8.5. Lisp

Lisp was surprisingly simple to implement; a reflection of the mathematical notation used in its initial design back in 1958[29]. The syntax is very simple, using spaces and parentheses. The right bracket "]" used to close out all previous S-Expressions has not (yet) been implemented. It will probably be context-sensitive, looking at the partially completed PST to determine how many S-Expressions to close out. Figure 75 shows the Programmar implementation of the 'defun' function, along with a sample usage.

```
public class Lisp_DefunFunction extends TokenSequence {
    public PunctuationLeftParen leftParen;
    public Lisp_Keyword DEFUN = new Lisp_Keyword("defun");
    public @OPT PunctuationComma comma;
    public Lisp_Function_Definition name;
    public Lisp_SExpr arguments;
    public TokenList<Lisp_SExpr> body;
    public PunctuationRightParen rightParen;
}
------------------------------------------------------------------------------
(defun random-plant (left top width height)
   (let ((pos (cons (+ left (random width)) (+ top (random height)))))
        (setf (gethash pos *plants*) t)))
```

Figure 75. Lisp 'defun' Function Definition and Usage

8.6. Natural

The "Natural" programming language[30] is owned and licensed by SoftwareAG in Germany. It was designed to be a more readable version of COBOL. It is mainly used in conjunction with AdaBase, a pre-relational database management system. SoftwareAG has stopped supporting the language, but they still charge outrageous licensing fees for usage (can be millions of dollars annually).

---

[29] en.wikipedia.org/wiki/Lisp_(programming_language)

[30] www.softwareag.com/corporate/products/adabas_natural

```
DEFINE DATA LOCAL
1 MYVIEW VIEW OF EMPLOYEES
  2 CITY
  2 NAME
  2 JOB-TITLE
  2 INCOME  (1:1)
    3 CURR-CODE
    3 SALARY
    3 BONUS (1:1)
END-DEFINE
*
WRITE TITLE 'XYZ EMPLOYEE ANNUAL SALARY AND BONUS REPORT' /
READ (3) MYVIEW BY CITY STARTING FROM 'E'
  DISPLAY GIVE SYSTEM FUNCTIONS
          NAME (AL=15) JOB-TITLE (AL=15) INCOME (1)
```

Figure 76. Small Natural Code Snippet

The data level numbers are very similar to COBOL's level numbers (see section 8.2.2). The Reporting and Structured modes use slightly different syntax[31]. Reporting mode is obsolete and does not require declaration of database fields.

8.7. PHP (Perl)

It has been nicknamed "the Swiss Army chainsaw of scripting languages" because of its flexibility and power, and possibly also because of its "ugliness". In 1998, it was also referred to as the "duct tape that holds the Internet together", in reference to both its ubiquitous use as a glue language and its perceived inelegance.

Figure 77. PHP Derision

The quote[32] from Figure 77 aptly describes the state of the PHP language, as well as its base language, Perl. Figure 78 shows a small snippet of PHP, which is still used for many dynamic websites.

```
function date3($year, $month, $day) {
   $dateStr = "";
   if ($year and $year > 0) {
      $dateStr = $year;

      if ($month and $month > 0) {
         if ($day and $day > 0) {
            $dateStr = $month . "/" . $day . "/" . $year;
         } else {
            $dateStr = $month . "/" . $year;
         }
      }
   }
   return $dateStr;
}
```

Figure 78. Snippet of PHP Code

8.8. PL/I, Programming Language One

PL/I is a language created by a committee, in the 1960's. It was meant to provide the best features of both COBOL and Fortran, but ended up as a very large complicated language, as in Figure 79[33].

[31] See www.silk.nih.gov/dbtek/viewdoc?vdsn=VPOD.D8329870.C040729.PDF, for example
[32] en.wikipedia.org/wiki/Perl

Speaking as someone who has delved into the intricacies of PL/I, I am sure that only Real Men could have written such a machine-hogging, cycle-grabbing, all-encompassing monster. Allocate an array and free the middle third? Sure! Why not? Multiply a character string times a bit string and assign the result to a float decimal? Go ahead! Free a controlled variable procedure parameter and reallocate it before passing it back? Overlay three different types of variable on the same memory location? Anything you say! Write a recursive macro? Well, no, but Real Men use rescan. How could a language so obviously designed and written by Real Men not be intended for Real Man use?

Figure 79. PL/I Derision

It shares the level number issues of COBOL (see 8.2.2) and the macro pre-processing of C (see 8.1.2). Like COBOL, it is very verbose with a very long syntactic description. It uses many keywords, but they are all context-sensitive keywords. For example, you can have variables called IF, THEN and ELSE, and use them inside an IF / THEN / ELSE block. Figure 80 shows a small snippet of PL/I.

Note the usage of % commands. These are all macros and are pre-processed by the PL/I compiler.

```
GAMMA:
   PROCEDURE (Z)  OPTIONS (REORDER) RECURSIVE RETURNS (FLOAT DECIMAL (15));
      DECLARE Z                    FLOAT DECIMAL (15);
 %INCLUDE MACFACT;
      DECLARE Y                    FLOAT DECIMAL (15);
      %DECLARE N                   FIXED;
      %N = 30;
      IF Z = 0 THEN SIGNAL ERROR;
      IF Z < 1 THEN
         RETURN (GAMMA(Z+1) / Z);
      IF (Z >= 1) & (Z <= N+1) THEN
         DO;
            IF Z = TRUNC(Z) THEN
               RETURN (FACTORIALS (Z-1));
         END;
      %DEACTIVATE N;
```

Figure 80. Snippet of PL/I Code

8.9. Python

Critical to Python programming is indentation. Loops are defined by indentation (white space) rather than keywords. Most traditional grammars do not consider white space, so they must pre-process Python programs before attempting to parse. In our Programmar system, the special terminal token called 'Python_StartOfLine' takes care of all that complexity without pre-processing. Caution must be used as well because the semicolon (;) is a statement separator, and the use of parentheses (or braces or brackets) allows a line to continue onto multiple lines, without any end-of-line markers. Figure 81 shows a snippet of Python code.

[33] en.wikipedia.org/wiki/PL/I

```
def solve(n):
    if n == 0:
        return [[]]
    smaller_solutions = solve(n - 1)
    return [solution+[(n,i+1)]
        for i in xrange(BOARD_SIZE)
            for solution in smaller_solutions
                if not under_attack(i+1, solution)]
for answer in solve(BOARD_SIZE):
    print answer,
```

Figure 81. Snippet of Python Code

Figure 82 shows the Programmar definition for the Python 'print' statement. Note the use of the "@CURIOUS()" Java annotation to indicate a token that we should accept, but we think it might be worth looking at later. "Why is there an extra comma at the end of the 'print' statement?"

```
public class Python_PrintStatement extends TokenSequence {
    public Python_Keyword PRINT = new Python_Keyword("print");
    public @OPT Python_Punctuation greater = new Python_Punctuation(">>");
    public @OPT SeparatedList<Python_Expression,PunctuationComma> exprs;
    public @OPT @CURIOUS("Extra comma") PunctuationComma comma;
}
```

Figure 82. Python 'print' Statement

8.10. Report Program Generator (RPG)

RPG[34] was created in the 1950's, so it is a very old language. It has also evolved over time. The original version was all fixed columns, as in Figure 83.

```
FINVHDR  IP  F      62            DISK
     FPRINTER O   F     132      OF     PRINTER
     IINVHDR  NS  01   1NC
     I                                       1   3 STORE L1
     I                                       4  13 INVNO
     I                                      14  20 CUSTNO
     I                                      21  45 STNAM
     I                                      46  53 INVDAT
     I                                      54  622TOTINV
     I           UDS
     I                                       1   8 RPTDAT
     C   01      INVDAT    COMP RPTDAT                 11
     C   01 11             ADD  TOTINV    L1TOT   92
     C   01 11             ADD  TOTINV    LRTOT   92
     OPRINTER H  101   1P
     O       OR        OF
     O                        PAGE  Z  106
     O                                 102 'PAGE'
     O                                  59 'VERY BIG'
     O                                  72 'STORES, INC.'
```

Figure 83. Sample RPG Snippet

There were two major versions of RPG, narrow and wide specification columns (see Figure 84). The first two parameters in those functions are the starting and ending columns for each field. The remaining parameters are allowed values.

[34] en.wikipedia.org/wiki/IBM_RPG

Note that the fields are the same for each, only the column numbers vary. For example, the fileName field has the exact same semantics for both cases, it just can be a little longer in the wider version.

```
public RPG_F_File_Specification_Narrow() {
    fileName = new RPG_Literal(7, 14);
    fileType = new RPG_KeywordChoice(15, 15, "I", "O", "U", "C");
    endOfFile = new RPG_Keyword(17, 17, "E");
    sequence = new RPG_KeywordChoice(18, 18, "A", "D");
    (etc.)

public RPG_F_File_Specification_Wide() {
    fileName = new RPG_Literal(7, 16);
    fileType = new RPG_KeywordChoice(17, 17, "I", "O", "U", "C");
    endOfFile = new RPG_Keyword(19, 19, "E");
    sequence = new RPG_KeywordChoice(21, 21, "A", "D");
    (etc.)
```

Figure 84. RPG Wide and Narrow Formats

A newer version of RPG (called RPG IV or RPG ILE) allows some free format programs. All versions of RPG have similar capabilities, just different syntax.

8.11. Other Languages

Programmars are being written for many other languages, with varying degrees of completeness, such as C#, Visual Basic, SQL, Delphi, Assembler (IBM and Intel), AWK, BNF, JSON, TCL, Gupta, Javascript, CMD, and others.

Because the Programmars are all open-sourced, it is our hope and expectation that a robust set of standard Programmars will evolve, covering major languages and their primary variations.

Note that there are two approaches to Programmar development. One is to find a BNF-like syntactic description and convert it. A BNF-to-Programmar transformation system is under active development. The other approach is empirical. Work from the collection of available programs and build up the Programmar.

For Analysis and Transformation purposes, we don't want to unconditionally do either of those approaches. A mix of the two is preferred, because

- There are many variations of most languages. It is rare to find a single BNF grammar that encompasses all the variations.
- It is often necessary to handle tokens in a different manner than a compiler. For example, a compiler can pre-process everything, and throw away all the comments.
- Terminal nodes are handled directly in Java code in a Programmar. In most traditional grammars, they are handled through an awkward set of production rules.
- Expression precedence is handled via the PrecedenceChooser, which greatly simplifies the sequencing of operations.

9. Parsing Patent

U.S. Patent #9,710,243 B2 was officially awarded in July 18, 2017. The brief title is PARSER, and the full title is "*Parser That Uses a Reflection Technique to Build a Program Semantic Tree*". The two authors are Steven A. O'Hara and Jeffrey A. Wilkinson. Both have assigned their rights over to Eagle Legacy Modernization, LLC. Note that many key parts of the patent were published in the paper in section 10.1.

> *A grammar of a first programming language is represented in member fields and data types of object-oriented classes of a second programming language as an empty program semantic tree. A parser builds a new program semantic tree that represents source code written in the first programming language. The new program semantic tree is built by a reflection technique in which the member fields and data types of the object-oriented classes of the second programming language as set out in the empty program semantic tree are modified during the building of the new program semantic tree.*

10. Publications

The following abstracts describe the peer-reviewed publications related to Eagle Legacy.

10.1. Paper #1: "Programmars: A Revolution in Computer Language Parsing"

We chose "*Software Engineering Research and Practice*" SERP 2015[35] and the paper was accepted for publication with very positive reviewer comments. The paper was authored by Dr. Steven A. O'Hara, and presented at the World Comp Conference in Las Vegas, July 30, 2015. Here is the abstract.

> *This paper presents a revolutionary way to parse computer programming languages without a traditional grammar. The motivation behind this approach is to dramatically increase scalability. The intention is to be able to parse and analyze billions of lines of code written in hundreds of programming languages. To achieve that goal, it is advantageous to have sharable, open-source, modular ways for defining the syntax and semantics of programming languages. The new parsing technique replaces a traditional grammar with a computer program, referred to as a Programmar (short for program and grammar). All the basic operations in BNF (sequencing, alternation, optional terms, repeating and grouping) are supported, and the Java code is both sharable and modular. This parsing approach enables dozens or even hundreds of developers to work on computer program analysis concurrently, while avoiding many of the consistency issues encountered when building grammars and associated code analysis tools.*

10.2. Paper #2: "Toward Effective Management of Large-Scale Software"

A second paper was presented at "*Software Engineering Research and Industrial Practice*" SER&IP 2016[36], which was held in Austin, Texas. This paper was co-authored by Dr. Andrew J. McAllister and Dr. O'Hara. The focus of the paper is a three-to-five year perspective on how Eagle Legacy's technologies can improve the world of software analysis and transformation. The abstract follows.

[35] world-academy-of-science.org/worldcomp15/ws/conferences/serp15
[36] sites.google.com/site/serip2016/

This paper outlines challenges the authors have faced over decades of experience with large-scale software analysis and maintenance projects (especially legacy modernization) for multiple organizations where millions of lines of source code are involved. Such projects require large teams cooperating on parsing, analyzing, and manipulating source code. In this context the use of traditional parsing techniques based on context-free grammars has proven problematic. We introduce the Programmar API, which supports a novel parsing approach designed to overcome these problems. The Programmar approach enables large teams to effectively extract complete, accurate, up-to-date information from application source code, and to provide this information as the basis for a wide variety of software management tools and activities. We present a framework that relates various types of such activities, and describe a vision for how the Programmar approach can provide significant benefits for the software industry in the future via an open-source distribution approach.

10.3. Paper #3: "Improving Programming Language Transformation"

Presented at the "*International Conference on Software Engineering Research and Practice*", SERP '18[37] on July 30, 2018 in Las Vegas, Nevada.

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.

10.4. Paper #4: "Modernizing Parsing Tools"

Presented at the "*ACM SIGPLAN International Workshop on the State of the Art in Program Analysis*", SOAP '19[38], June 22, 2019 in Phoenix, Arizona. Co-authored by Dr Rocky Slavin.

Software Engineering tools today are hampered by weaknesses in parsing and analysis tools. For example, there are no standard repositories of grammars for the most popular programming languages. If an organization has software written in Python, Java, Bash, SQL, HTML, CSS, JavaScript and so on, there is no readily available mechanism to parse and analyze all of the software in a unified manner. This paper describes a collection of tools for parsing and analyzing many different languages, including legacy languages like COBOL and Fortran. The primary goal is scalability; dealing with a single programming language and a limited number of programs is far simpler than dealing with millions of lines of code written in many different languages.

[37] americancse.org/events/csce2018/conferences/serp18
[38] pldi19.sigplan.org/home/SOAP-2019

10.5. Unpublished Research on Cross-Language Data Flow

One of Dr. McAllister's PhD students (David Kell) at the University of New Brunswick (in Fredericton, Canada) did some research on the long-range problem of data flow across heterogeneous systems. For example, a screen input value gets manipulated then stored in a database. Another program reads the database and writes to a file. That file gets ftp'd to another computer. A program reads the file and saves the contents as a csv file. A spreadsheet is opened from that csv file and values are extracted and displayed to a user. This type of dataflow is very challenging and requires a very solid foundation.

11. Biography

Steven A. O'Hara has a Ph.D. in Computer Science from the University of Texas at San Antonio (UTSA), awarded in 2014. Recently, he retired from UTSA as an Assistant Professor in the Computer Science Department, where he taught classes in Artificial Intelligence, Cyber Security, Programming Languages and others. He has worked with computers since the early 1970's, including at Google, working in the Search team. During his tenure as Chief Technical Officer at a Software Modernization company, he became acutely aware of the many difficulties involved in large-scale analysis and transformation.