# Programmars: A Revolution in Computer Language Parsing

Steven A. O'Hara, PhD
Eagle Legacy Modernization, LLC
702 Southwick Avenue
Grovetown, GA 30813
steve@eaglegacy.com

**Topical keywords**: parser, grammar, software analysis

**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.*

## 1 Introduction

Businesses around the world today collectively have billions of lines of production software written in legacy computer languages like COBOL, RPG, PL/I, Fortran and Natural. These organizations are highly motivated to modernize their software for a number of reasons, including difficulties in maintaining old, brittle code [1] and in hiring people with legacy skillsets [2]. Unfortunately the modernization process is often either prohibitively expensive or produces new software of low quality that is difficult to maintain going forward into the future [3]. Available modernization tools (e.g. [4 to 8]) tend not to be scalable enough to handle large, complex software systems that can be comprised of tens of millions of lines of code written in multiple programming languages.

For the past several decades, legacy software analysis tools have been typified by the type of parser generated by Yet-Another-Compiler-Compiler (YACC) [9]. Such a parser interprets computer program code based on a Context-Free Grammar, which is a declarative description of the syntax of a specific programming language. This parsing process relies on a separate token pre-processor (typically LEX, the Lexical Analyzer [10]) and generates an Abstract Syntax Tree (AST).

Modern programming languages also continue to evolve and require solid analysis approaches (e.g. [11 to 13]). For example, managing deprecated code often requires detailed software analysis similar to application modernization. Unfortunately, there are many one-off grammars and tools for source code analysis, but no standard or shared tools that work well across many programming languages at the same time. With our technique, we process languages as disparate as Java, HTML, CSS, DOS, XML, COBOL, Natural and RPG using a single parser.

This paper introduces a new parsing technique that embeds all required parsing information within a Java program. All of the elements needed to describe the computer programming language(s) to be parsed are embedded in the Java program as fields, classes or methods within Java classes. The focus of this work to date is on parsing languages in the context of legacy application modernization.

Grammar rules can be separated into two categories, those that depend on other rules (defining non-terminals) and those that consume characters in the input stream (terminals). Examples of terminals are string literals, comments, numbers, keywords, and punctuation.

In the Programmar approach, Java methods are used for parsing terminals, while classes are defined to enable parsing of non-terminals. The Programmar API uses Java reflection [14] to dynamically infer a grammar while parsing.

## 2 Comparison to Old Grammars

This section describes the relationship between a traditional BNF-like grammar and our new Programmar.

***Sequences***

Given a BNF production rule of the form

```
<A> ::= <X> <Y> <Z>;
```

This is represented in a Programmar as in Figure 1.

```
public class A extends TokenSequence {
     public X x;
     public Y y;
     public Z z;
}
```

**Figure 1: Sequencing Programmar example**

Note that the elements are anonymous in the BNF grammar. The BNF production rule indicates only that an instance of type A can consist of an (unnamed) instance of type X, followed by an instance of type Y, and then an instance of type Z. In Figure 1, however, the instances are named, which means these specific instances can be referred to from elsewhere in the Programmar or in associated source code analysis programs. This turns out to be extremely valuable when analyzing computer source code.

***Alternation***

Given a BNF production rule of the form

```
<A> ::= <X> | <Y> | <Z>;
```

This is represented in a Programmar as in Figure 2.

```
public class A extends TokenChooser {
     public X x;
     public Y y;
     public Z z;
}
```

**Figure 2: Alternation Programmar example**

This means that exactly one of the three elements must be present to be recognized as an instance of type A. As a convenience, anonymous inner classes can be used as well.

***Optional Terms***

Given a BNF production rule of the form

```
<A> ::= <X> [<Y>] <Z>;
```

This is represented in a Programmar as in Figure 3.

```
public class A extends TokenSequence {
     public X x;
     public @OPT Y y;
     public Z z;
}
```

**Figure 3: Optional item Programmar example**

***Repeating Terms***

Given a BNF production rule of the form

```
<A> ::= <X> <Y>* <Z>;
```

This is represented in a Programmar as in Figure 4.

```
public class A extends TokenSequence {
     public X x;
     public @OPT TokenList<Y> y;
     public Z z;
}
```

**Figure 4: Repeating term Programmar example**

The '+' BNF notation is handled in a similar manner, without the @OPT.

***Terminal Nodes***

Java code is provided by the Programmar API to assist parsing the most common terminal nodes. For example, string literals in various programming languages commonly have a number of features such as:

- Single or double quotes?
- Are pairs of quotes treated as single quote?
- What is the escape character, if any?
- Can a literal span multiple lines?

Similar routines are available for comments, numbers, punctuation, etc. By using Java code for the terminal nodes, the parsing speed is greatly improved. In our experience, writing a BNF-like grammar for a floating point number or a string literal can be challenging and time consuming.

It is not necessary to use one of the built-in methods for parsing terminal nodes. For example, Python has very strict rules for indentation. Rather than pre-processing the input stream, a Start-of-line terminal node can be used to handle the indentation logic correctly.

# 3 Motivating Example – Old Grammar

Consider the two PERFORM statements in Figure 5.

```
000160 READ-SHARED-LOCK.
000170     READ SHARED WITH LOCK.
000180     IF WS-STATUS = "00"
000190         GO TO READ-SHARED-EXIT.
000200     IF WS-STAT1 = "2" OR "3"
000210         MOVE 33 TO WS-F-ERROR
000220         PERFORM READ-ERROR.
000230     IF RECORD-LOCKED
000240         PERFORM LOCK-USERS-REC
000250             THRU LOCK-REC-EXIT
000260             WS-COUNT TIMES
000270         ADD 1 TO WS-COUNT
000280         IF WS-COUNT > 25
000290             MOVE 1 TO WS-COUNT
000300         END-IF.
```

**Figure 5: Sample COBOL code**

In a traditional grammar, the PERFORM verb in COBOL might be expressed as in Figure 6 (this is greatly simplified).

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

**Figure 6: Traditional grammar example**

Once the COBOL program has been parsed, tools can be used to analyze the AST. Typically, such tools traverse the tree looking for specific named entities, such as cPerform. These tools depend heavily on the names used in the grammar. If somebody changes the name of an element in the grammar, there is no easy way to detect that change.

Grammars used during a modernization effort tend to require significant changes when another effort begins, for a variety

of reasons. This can happen because of language variations, hardware variations, operating system variations, business-specific conventions, etc. The tools depend heavily on the terms in the grammar, and the terms are in flux, so it is very easy for grammars to get out of sync with the analysis tools.

This is the main reason why parsers and grammars do not scale well. Minor changes to grammars can have many subtle adverse effects on how parsers work, as well as the tools that perform subsequent processing tasks.

Generally, traditional grammars and parsers are successful when there are only a few people working on them, and they are working on only one or two computer languages. Most large-scale businesses deal with dozens of computer languages. Getting a single parser to handle all languages is difficult, because many languages require significant pre-processing. Also, getting some level of consistency between AST's is difficult.

Our primary goal of parsing all major computer languages in a unified manner is only possible when the grammar and the analysis tools are written in the same language. This is the basis for our claim of scalability with Programmars.

# 4 Motivating Example – Programmar

With a Programmar, the rules related to COBOL PERFORM verbs could be expressed in a language like Java as in Figure 7. Again, this is a simplified version that does not reflect all possible PERFORM variations.

```
class CobolPerform extends CobolStatement {
  CobolKeyword PERFORM =
      new CobolKeyword("PERFORM");
  CobolParagraph startPara;
  @OPT CobolPerfThrough through;
  @OPT CobolPerfTimes times;

  class CobolPerfThrough extends TokenSequence {
    CobolKeywordList THRU =
        new CobolKeywordList("THRU", "THROUGH");
    CobolParagraph endPara;
  }

  class CobolPerfTimes extends TokenSequence {
    CobolExpression count;
    CobolKeyword TIMES =
        new CobolKeyword("TIMES");
  }
}
```

**Figure 7: Simplified Programmar example**

Terms like TokenSequence are built in to the Programmar API, and terms like CobolKeyword, CobolExpression, etc. are defined in other Java classes.

The Java program representation shown serves two distinct purposes. First, the program can be considered a grammar for defining the PERFORM statement in COBOL, describing all the different ways the statement can be formed. Second, rather than creating an AST, the parsing process creates instances of this type of class. Collectively these instances form a Programmar Semantic Tree (PST). The PST can be stored as an XML file or as Java code that regenerates it.

# 5 Advantages

Although the Programmar approach is slightly more verbose than using a context-free grammar (CFG), the new approach offers four major advantages:

***Advantage #1 – Dependencies are caught automatically***

In the Programmar approach, downstream impact of a change to any part of a Progammar will be detected immediately, because the Java Programmar won't compile. If somebody were to change the name of an element in the Programmar, all references to that name would become invalid until they were updated to be consistent with the changed element. This allows projects to scale to much more significant levels. It is now possible to have dozens of developers working on the same project, processing many computer languages.

***Advantage #2 – Semantics: Entities are not just names***

In the AST version, a cParagraph is just an identifier. There is no further information attached to it. If you write a tool to analyze or transform a COBOL program, you will have to search the rest of the AST to find out what is in that other paragraph. With the PST version, the CobolParagraph instance contains within it a reference to the actual definition of that paragraph, including all of its statements, line numbers, references, etc. This greatly simplifies the task of writing analysis and conversion tools. Some of the work in connecting references to definitions is accomplished as part of the parsing process, which lessens the effort required to create tools for subsequent processing tasks.

***Advantage #3 – Context-Sensitive, not Context-Free***

The terminal nodes in a Programmar, such as CobolKeyword, are represented in Java code. When trying to parse a terminal node, the Java code has access to the current context. In Figure 8, COBOL level numbers are very important. A level number 10 following a level 05 means that the 10 should be stored in a sub-tree under the 05. But the next 10 should be stored under the same 05. In a Programmar, COBOL level number logic can examine the context to decide how to correctly build the hierarchy, without any post-processing.

```
01  INV-L7.
  05  FILLER          PIC  X(02).
  05  INV-REMKS.
    10  INV-SM        PIC  X(09).
    10  INV-SMAN      PIC  X(31).
  05  FILLER          PIC  X(05).
  05  INV-RMKS.
    10  FILLER        PIC  X(17).
    10  INV-H18       PIC  X(12).
    10  INV-TERMS.
      15  INV-ADV     PIC  Z(06)9.99-.
```

**Figure 8: Traditional grammar example**

#### *Advantage #4 – Utilize modern programming languages*

Programmars are expressed in a modern programming language such as Java, which means the development methodologies associated with such a modern language can be used when working with Programmars. We identify the five following benefits.

*Abstraction*. A traditional grammar is typically built to describe just one programming language. With the Programmar approach, the components common to all variations of a particular programming language can be placed into an Abstract Programmar class. For example, there are major variations of languages like Report Program Generator (RPG). A File specification has the same meaning across each variation, so an abstract RPGFile class can be used to define the common elements. The minor syntactic differences between RPG variations can then be represented by concrete classes that extend the abstract class.

*Inheritance*. Frequently, there are variations on a computer programming language. For example, there are both fixed width (80 column) and free-format COBOL programs. Their meanings are virtually identical, but the syntax is very different. With a traditional grammar, the whole grammar might get copied and edited for each variation. With Programmar Inheritance, only the local changes need to be considered and the rest can be inherited from the main Programmar.

*Encapsulation*. Some computer languages, such as HTML for web pages, often include other languages inside of them; Javascript, CSS or PHP in the case of HTML. In a traditional grammar, these are normally combined into a monolithic grammar covering all sub-languages. With Programmar Encapsulation, the main Programmar (e.g., HTML) can simply reference the other Programmar (e.g., Javascript).

*Logic*. With Programmar logic, the full power of the programming language used to represent the Programmar (e.g., Java) is available for complicated cases. Managing the PICTURE level numbers in COBOL is a good example where logic is needed to assist the parsing process to build the correct hierarchy.

*Shared Processing*. Most terminal tokens are somewhat similar across programming languages. Generic processors for numbers, literal strings, punctuation, etc. are all made available by the Programmar API to use when writing Programmars. For example, parsing functionality for hexadecimal (hex, base 16) numbers can generally be implemented in just a few lines of Java by extending the generic hex number processor, and simply declaring their hex prefix or suffix. Comments, floating point numbers and string literals are simpler to implement in a Programmar than a traditional grammar, because they can utilize the built-in generic methods.

# 6 Programmar Token Types

Every element in a Programmar is an AbstractToken in the representation of a programming language. The following are the main types of Abstract Tokens.

#### *TokenSequence: Sequence of Tokens*

When a Programmar class extends TokenSequence, an instance of this type is identified during parsing when all of the fields in the class are present in the order specified (unless they are marked as optional). Inner classes are a convenient way to define sub-rules for such an element.

#### *TokenChooser: Choose One Token*

Programmar classes that extend TokenChooser have both fields and inner classes, and the parser will attempt to match both. Each can be marked with @FIRST or @LAST because the parser makes three passes. The first looks only at @FIRST elements, the second looks at neither @FIRST or @LAST elements, and the third pass looks only at @LAST elements. This gives another level of control over the parser, and can be used to speed up the parser.

Care must be used in the order of the elements in a TokenChooser. Once a token matches from the list of choices, no other tokens are considered. This is done for efficiency purposes and doesn't seem to impose any restrictions other than being careful with ordering. BNF-like rules such as "<A> ::= <X> | <X> <Y>;" need to be written in the other order.

#### *TokenList: One or More Tokens*

A field may be a TokenList of another token type, typically a TokenSequence. It will match one or more instances of that type (or zero if @OPT is also present).

#### *PrecedenceToken: Operator Precedence*

This is a specialized token to help represent expressions in most computer languages. It allows a declarative specification of unary operators (like minus and not) and binary operators (like plus and times). The order of the elements in the specification determines their precedence. In most BNF-like grammars, this is a complicated process and often verbose. The Programmar approach includes a short-cut way to define operator precedence, which reduces complexity.

The parser also handles the "left-recursion" problem. A Programmar class can represent a BNF-like rule (essentially) as "expr ::= expr + expr" without getting into an infinite loop.

Typical AST's built from parsing expressions can be very long. Our parsing process eliminates needless intermediate layers, which greatly reduces the size and complexity of the resulting PST. For example, if there is no multiplication in the expression, then there is no multiplication node in the PST.

#### *UnparsedElement: Allow Parse to Continue*

In some situations, we may know there will be some statements we might not be able to parse. In that case, we can

add an @LAST UnparsedElement to a TokenChooser. This may allow parsing to continue in the event of a parse failure, resulting in a "soft" parse failure.

#### *Optional Element*

@OPT is used to indicate that a Token is optional. It is often used in conjunction with a TokenList to mean zero or more elements instead of one or more.

#### *Terminal Tokens*

There are several types of pre-defined terminal tokens, such as:

- TerminalLiteralToken
- TerminalNumberToken
- TerminalPunctuationToken
- TerminalKeywordToken
- TerminalCommentToken

Each of these provides built-in generic parsers to support most programming languages. Furthermore, each has an associated CSS style for the code inspection modules so it is easy to visually identify all the literals, numbers, comments, etc. in a computer program listing.

## 7 How Programmars Work

The central idea behind parsing with Programmars is to use Reflection to fill in the PST with the results of the parsing process. It uses a top-down approach with no look-ahead (i.e., it is an LL(0) grammar [15]). No token pre-processor is required.

The use of a Programmar differs greatly from a Recursive Descent Parser (RDP) [15] because Programmars use a declarative way of representing computer languages and rely on Java reflection to decide how to parse. Other than terminal nodes, there is no logic in a Programmar. A RDP, in contrast, uses programming logic for matching each and every node in the grammar.

The Programmar parsing process is context-sensitive for terminal nodes in the sense that the current (partial) parse tree is accessible to the terminal node parser. For example, PL/I level numbers are hierarchical like COBOL level numbers and they are parsed by looking at what is already in the parse tree.

Reflection is heavily used in a Programmar parser. The parser examines all the data fields and classes inside a given class. Depending on the type of Abstract Token, a different strategy is used. For a Token Sequence, all the elements must be present in the given order. If any one element doesn't match, the whole Token Sequence fails. Recursion is also heavily used since Abstract Tokens may contain other Abstract Tokens.

There is no grammar (other than the Java program), and there is no AST. The result is a programmatic representation of the original source program. This technique has been demonstrated to parse dozens of computer programming languages such as Assembler, Fortran, PL/I, RPG, Java, Visual Basic, Delphi, DOS, SQL, Python, C++, and many more.

One of the available outputs of the Programmar parser is a traditional grammar. In other words, given the Programmar representation of the COBOL programming language, the system can automatically generate a traditional BNF-like grammar from it.

## 8 JSON Example

Javascript Object Notation (JSON) is a simple language, so it is convenient to use for an abbreviated example, with some key parts omitted from the Programmar shown in Figure 9.

```
public class JSON_Program extends Language {
  public TokenList<JSON_Element> elements;
}

public class JSON_Element extends TokenChooser {
  public JSON_Literal literal;
  public JSON_Number number;
  public JSON_Object object;
  public JSON_Dict dictionary;
  public JSON_KeywordChoice constants = new
    JSON_KeywordChoice("null", "true", "false");
}

public class JSON_Object extends TokenSequence {
  public JSON_Punctuation leftBracket = new
    JSON_Punctuation('[');
  public @OPT JSON_Element element;
  public @OPT TokenList<JSON_MoreElements> more;
  public JSON_Punctuation rightBracket = new
    JSON_Punctuation(']');
}

public class JSON_Dict extends TokenSequence {
  public JSON_Punctuation leftBrace = new
    JSON_Punctuation('{');
  public @OPT JSON_Entry entry;
  public @OPT TokenList<JSON_MoreEntries> more;
  public JSON_Punctuation rightBrace = new
    JSON_Punctuation('}');
}

public class JSON_Entry extends TokenSequence {
  public JSON_Literal name;
  public JSON_Punctuation colon = new
    JSON_Punctuation(':');
  public JSON_Element value;
}
```

**Figure 9: JSON Programmar**

```
[
 {
   "pk": "1",
   "model": "fixtures_regress.absolute",
   "fields": {
     "name": "Load Absolute Path Test"
   }
 }
]
```

**Figure 10: Sample JSON source code**

Consider the sample JSON source code in Figure 10. The trace in Figure 11 shows the parsing process. "Next" is the current character sequence. "Pattern" is the name of the Programmar class. The leading periods show the parsing recursion depth.

```
Next    Pattern
======  ======================================
[       ? JSON_Program
[       .? JSON_Element
[       ..? JSON_Literal ()
[       .. Failed JSON_Literal ()
[       ..? JSON_Number
[       .. Failed JSON_Number
[       ..? JSON_Object
[       ...? JSON_Punctuation ([)
{       ... ***** Match JSON_Punctuation ([)
{       ...? JSON_Element
{       ....? JSON_Literal ()
{       .... Failed JSON_Literal ()
{       ....? JSON_Number
{       .... Failed JSON_Number
{       ....? JSON_Object
{       .....? JSON_Punctuation ([)
{       ..... Failed JSON_Punctuation ([)
{       .... Failed JSON_Object
{       ....? JSON_Dict
{       .....? JSON_Punctuation ({)
"pk":   ..... ***** Match JSON_Punctuation ({)
"pk":   .....? JSON_Entry
"pk":   ......? JSON_Literal ()
: "1"   ..... ***** Match JSON_Literal ("pk")
: "1"   ......? JSON_Punctuation (:)
"1",    ..... ***** Match JSON_Punctuation (:)
"1",    ......? JSON_Element
"1",    .......? JSON_Literal ()
,       ..... ***** Match JSON_Literal ("1")
,       ..... ***** Match JSON_Element
,       ..... ***** Match JSON_Entry
    (59 lines omitted)
]       ..... ***** Match JSON_Punctuation ({)
]       .... ***** Match JSON_Dict
]       ... ***** Match JSON_Element
]       ...? JSON_MoreElements
]       ....? JSON_Punctuation (,)
]       .... Failed JSON_Punctuation (,)
]       ... Failed JSON_MoreElements
]       ...? JSON_Punctuation (])
(EOF)   ... ***** Match JSON_Punctuation (])
(EOF)   .. ***** Match JSON_Object
(EOF)   . ***** Match JSON_Element
(EOF)   . ? JSON_Element
(EOF)   ***** Match JSON_Program
```

**Figure 11: Trace of the parsing process**

To parse a JSON_Program, the parser first tries to match a JSON_Element, which has to be a JSON_Literal, or a JSON_Number, etc. The parser eventually matches on a JSON_Object, at the fourth line from the bottom in Figure 11.

Each "?" in Figure 11 represents one parsing step. Each step should be considered a parsing attempt, which may or may not match the input text stream.

Figure 12 is an abbreviated version of the generated XML version of the PST. Starting and ending character and line positions are in the actual XML file for every token.

```
<Program Elapsed="1" Steps="52" Tokens="35">
 <Token TT="JSON_Program">
  <Token Name="elements" TT="List">
   <Token TT="JSON_Element">
    <Token TT="JSON_Object">
     <Token Name="leftBracket"
       TT="JSON_Punctuation" V="["/>
     <Token Name="element" TT="JSON_Element">
      <Token TT="JSON_Dict">
       <Token Name="leftBrace"
         TT="JSON_Punctuation" V="{"/>
       <Token Name="entry" TT="JSON_Entry">
        <Token Name="name"
          TT="JSON_Literal" V="pk"/>
        <Token Name="colon"
          TT="JSON_Punctuation" V=":"/>
        <Token Name="value" TT="JSON_Element">
         <Token TT="JSON_Literal" V="1"/>
        </Token>
       </Token>
       <Token Name="more" TT="List"/> (omitted)
       <Token Name="rightBrace"
         TT="JSON_Punctuation" V="}"/>
      </Token>
     </Token>
     <Token Name="rightBracket"
       TT="JSON_Punctuation" V="]"/>
    </Token>
   </Token>
  </Token>
 </Token>
</Program>
```

**Figure 12: XML representation of a PST**

According to the first line, this took approximately 1 ms to parse, with 52 parsing steps. There are 35 tokens in the final PST. Note that the "more" token was omitted from this listing.

# 9 Additional Tools

Initial versions of these code analysis tools are available.

We define a project as a (possibly large) collection of computer programs written in a variety of different programming languages. In our system, a project is a Java class that specifies what language to use for each source file, what character encoding it has, how to interpret tabs in it, etc. A project also has a very small editor built-in to repair known problems in specific files. For example, missing semicolons can be added or typos repaired. A project uses a simple regular expression matcher on pre-specified line numbers on specific files.

***Traditional Grammar Generator***

Given a Programmar, a traditional BNF-like grammar can be generated from it. However, since terminal nodes are expressed as Java methods, not as grammar rules, they cannot be generated automatically. When used on a collection of computer programs, frequency counts are also available showing how many instances of each rule are present in that project.

***Pretty Printer***

Once a computer program has been parsed, the parsing results can also be used to regenerate the original program, but in a

canonical form. Indentation can be fixed, extra spaces eliminated, capitalization standardized, etc. Elements in the Programmar can have annotations on them to assist with the output formatting, such as @NEWLINE, @NOSPACE, @INDENT, and @OUTDENT.

***Programmar Debugger***

The parser can generate a tracing output, either in plain text or html. Sometimes this output is not sufficient, such as when debugging a parse failure. A visual debugger is available for this with commands like Step-in, Step-over, Continue, etc.

***Program Inspector***

Once a program has been parsed, an html report can be created for it, with color codes for all the different kinds of terminal nodes. In addition, elements in the Programmar can be labelled with @DOC("href") to create a hyperlink to an online document describing that keyword.

***Inventory Browser***

While working on large projects, or more than one project, it is often useful to monitor parsing progress. Web-based tools are available to show all active projects and all active languages. For each project, all the files are shown with statistics like number of lines, size of the resulting parse output, parsing speed, etc. If a parse fails, a link is provided to view the details of the parse failure, including the lines around the failure.

For each language, details are shown as well for each source file such as parse success rates for that language and average parse speeds. A BNF-like grammar for that language is also viewable, with frequency counts.

***Macro Pre-Processor***

For many languages such as C, PL/I and COBOL, parsing is sometimes not possible in a single pass. We have a pre-processor available to resolve macros. Generally, not all macros need to be expanded, so controls are available to choose which macros to expand and which to leave intact.

# 10 Future Work

We plan to create an open-source repository for Programmars as well as an API for parsing programs over the web.

Work has already begun on connecting variable references to their definitions. In some cases, this can be done while parsing, but in many cases such connections must be done in a separate step because they depend on successfully parsing other files.

Ultimately, this work is intended to be helpful in application modernization, especially from legacy programming languages to more modern languages.

# 11 Conclusion

The Programmar approach builds on top of traditional parsing technologies. It greatly facilitates scalability and cross-language processing, and it is also context-sensitive (for terminal nodes). As of this writing, several dozen programming languages have Programmars built for them, with varying degrees of completeness. These Programmars have been used to parse millions of lines of code. A patent is pending on this technique.

# 12 References

[1] C. Preimesberger. Updating Legacy IT Systems While Mitigating Risks: 10 Best Practices, eWeek, Mar. 19, 2014, 7.

[2] R.L. Mitchell, M. Keefe, The COBOL Brain Drain, Computerworld, Vol. 46, Issue 10, May 2012, 18-25.

[3] A.J. McAllister, Automation-Enabled Code Conversion, Proceedings SERP '10: International Conference on Software Engineering Research and Practice, July 2010, 11-17.

[4] Modern Systems: COBOL Conversion and Migration, http://www.ateras.com/solutions/legacy-migration/cobol-migration.aspx

[5] Semantic Designs: COBOL Migration, http://www.semdesigns.com/Products/Services/COBOLMigration.html

[6] MSS International: COBOL to Java Conversion, http://www.mssint.com/sites/default/files/MSS-Cobol-to Java-conversion.pdf

[7] RES – A Pure Java Open Source COBOL To Java Translator, http://opencobol2java.sourceforge.net

[8] eranea: Modernize your core IT system toward Java, Web, Linux, http://www.eranea.com

[9] S.C. Johnson, Yacc: Yet Another Compiler-Compiler, AT&T Bell Laboratories, 1975.

[10] J.R. Levine, T. Mason, & D. Brown Lex & Yacc, O'Reilly & Associates, 1992.

[11] T.A.Wagner, S.L. Graham, Incremental Analysis of Real Programming Languages, Proceedings of the ACM Conference on Programming Language Design and Implementation, 1997, 31–43.

[12] D. Jackson, M. Rinard, Software analysis: A roadmap, Conference on The Future of Software Engineering, 2000, 135–145.

[13] Z.P. Fry, D. Shepherd, E. Hill, L. Pollock, K. Vijay-Shanker, Analysing source code: looking for useful verb–direct object pairs in all the right places, IET Software, Vol. 2, Issue 1, Feb. 2008, 27-36.

[14] I.R. Forman, N. Forman, Java Reflection in Action, Manning Publications, 2004.

[15] A.V. Aho, M.S. Lam, R. Sethi, J.D. Ullman, Compilers: Principles, Techniques, and Tools (2nd Edition), Addison Wesley, 2006.