metaware.

the complete IT modernization solution

Parsing Technology and its role in Legacy Modernization

A Metaware White Paper

1 INTRODUCTION

In the two last decades there has been an explosion of interest in software tools that can automate key tasks in modernizing legacy systems such as:

- Replatforming applications to escape from the high cost of mainframes,
- Migrating applications from obsolete data storage to relational databases,
- Refactoring to simplify code structure and reduce maintenance costs, and
- Web-enabling applications to make them accessible on online.

Most automated software modernization tools rely upon a technology called *parsing* to convert software code from text files into a form that facilitates automated analysis and transformation of the legacy system.

This paper is intended to:

- Give an overview of parsing to illustrate its purpose and function
- Explain the role of parsing in automated software modernization tools
- Survey common parsing techniques used these tools
- Describe advances in parsing that enable a dramatic improvement in the power and accuracy of automated software modernization tools, while also reducing tool development time

2 OVERVIEW OF PARSING

In software engineering, *parsing* is defined as the process of analyzing the source code of a program in order to determine its grammatical structure. A *parser* for a programming language (e.g., COBOL) is a software tool that performs parsing of source code in that language.

Parsers were originally invented for use in compilers and interpreters for programming languages. However, they have proven to be extremely valuable for use in other tools that automatically process source code, including tools for legacy system modernization.

The input to a parser is usually one or more text files containing source code. The output of a parser is typically a tree structure that represents the grammatical structure of the code according to the syntax rules of its programming language.

2.1 Parsers Create Trees

The tree structure produced by a parser is analogous to the tree diagrams that are sometimes used in schools to teach the grammar rules of natural languages such as English or French.

For example, a tree diagram for the sentence "Bob threw the red ball" could be drawn as:

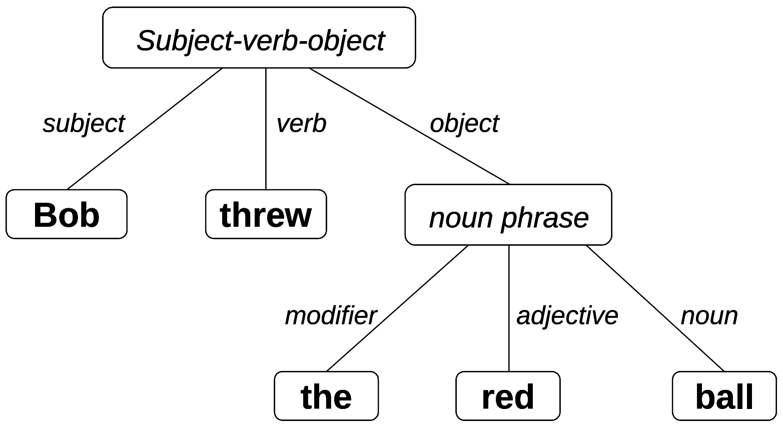

Figure 1: Tree diagram for "Bob threw the red ball"

In the diagram above, the tree is drawn upside down, with its root at the top and the "leaves" at the bottom. This is a common convention when discussing parsers and we follow it in the rest of this paper.

The tree shown above describes the structure of the sentence according to the grammatical rules of the English language. The entire sentence is a subject-verb-object phrase, so the root of the tree is labeled "subject-verb-object". The subject of the sentence is "Bob", and the verb is "threw". The object of the sentence is a noun phrase that itself has three parts: a modifier ("the"), an adjective ("red"), and a noun ("ball").

2.2 Trees to Represent Software Source Code

Parsers for programming languages typically create a tree structure called an *abstract syntax tree* (AST) to represent the grammatical structure of the parsed source code.

For example, the Java assignment statement:

```
a = a + 10;
```

could be represented by the AST below.

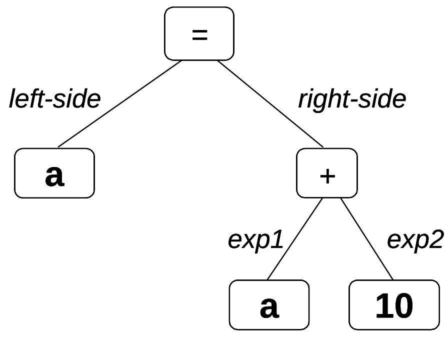

Figure 2: AST to represent "a = a + 10;"

The AST shown in Figure 2 above represents the structure of the source code **`a = a + 10;`** according to the grammatical rules of the Java language. The grammatical rules of many programming languages are the same as Java for assignment statements, but some other languages use a different symbol instead of "=".

When parsing a program with three files, the output of the parser will be an AST whose root looks like:

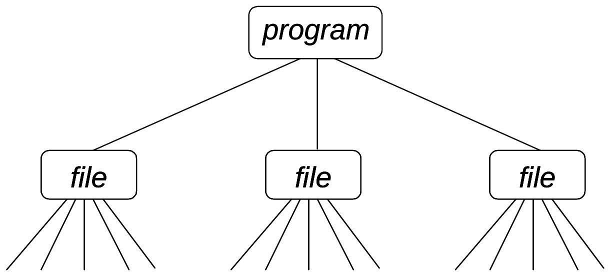

Figure 3: Root of a tree for a program with three files

Under each object labeled "file" would be tree nodes to represent the variables, functions and other language elements contained in the file.

When parsing a legacy application with millions of lines of source code, the resulting tree can have several million objects, and can reach depths of 20 levels or more between the AST root and the leaves (nodes at the bottom of the AST).

2.3 Grammars

The grammatical rules for a programming language, also known as the *syntax rules* or *syntax* of the language, are often formalized in a description called a *grammar*.

For each element of the language—variable declarations, assignment, addition etc—the grammar contains one or more *production rules* that specify the sub-parts of the language element, the order in which they appear, and *keywords* that separate the sub-parts.

For example, a production rule for the an assignment statement might look like:

```
assignment => variable "=" expression;
```

The above production rule can be read as saying "an assignment consists of a variable, followed by the keyword "=", followed by an expression.

Reference manuals for a programming language often include a grammar for the language to exactly specify the syntax rules for the language.

There are several notations for writing grammars. One of the most common notations is named *Backus-Naur form*, also known as BNF.

2.4 Parser Generators

In the early days of software engineering, parsers were usually written manually, and the development of a parser was a time-consuming and error-prone process. Even today, many programmers will write parsers for simple languages "by hand": the programmer will study a grammar for the language and write a corresponding parser.

A great advance in software engineering was the invention of *parser generators*, which take as input a grammar for a programming language, and produce as output a parser for the language. Parser generators greatly reduce the amount of work required to create a parser. Parser generators also provide additional benefits, including the ability to determine if a grammar has problems such as ambigous syntax rules that allow alternative parsings of a program into two or more different ASTs.

Today, most parsers for programming languages are created using parser generators, in order to reduce development time and prevent errors in the grammar and parser. The parsers created by parser generators are generally about as fast as hand-written parsers.

3 THE ROLE OF PARSING IN LEGACY MODERNIZATION

The parsing process creates AST representation of source code that enables an automated tool to perform accurate analysis and transformation of the code.

As mentioned previously, the original purpose of parsers was to create a representation of source code that allowed a compiler to transform the source code into machine code for execution. A new breed of automated tools has exploits parsers to enable the analysis and translation of source code for legacy modernization purposes, as shown below:

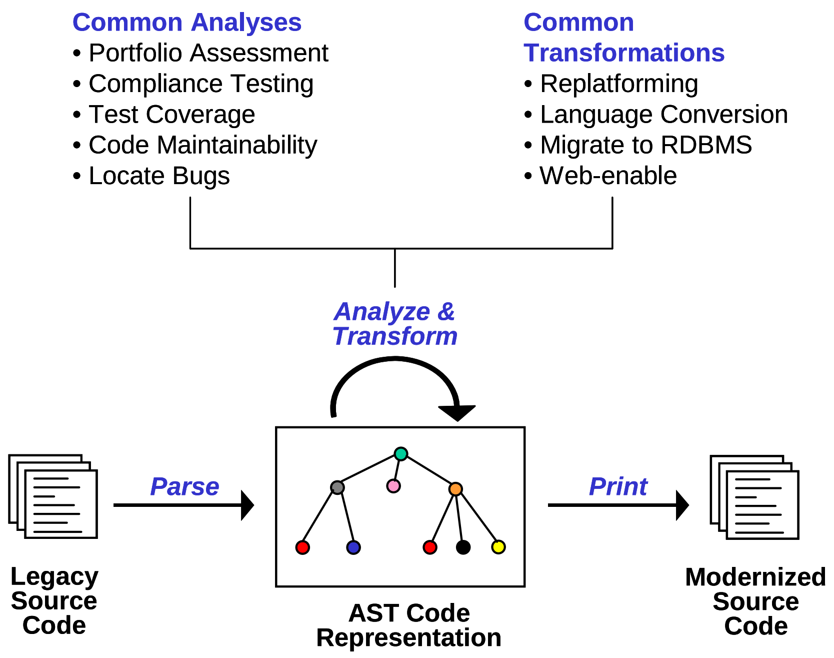

Figure 4: Parsing enables legacy system modernization by analysis and transformation

3.1 Code Representation Requirements for Analysis and Transformation

Why is it necessary to parse code first in order to analyze and transform it? Why can't a tool directly analyze and transform the text in the source files?

The reason is that in order to accurately analyze and transform code, a tool must have a model of the *relationships* between parts of source code. Examples of relationships include:

- The hierarchical relationship between the nodes in an AST. For example, an assignment statement has a parent-child relationship with the left-side and right-side of the assignment.
- The code that declares a variable or function is related to each part of the code that references that variable or function.

- For each statement, there is a set of statements that could be executed immediately before or after that statement.

3.2 The Importance of Relationships between Parts of Source Code

Consider a simple task: changing the name of a variable from "x" to "index". To automate this change, a tool must change "x" to "index" in the variable declaration, and also change every reference to that variable from "x" to "index".

This is harder than it might at first appear. The tool cannot simply find all occurrences of the symbol "x" in the program and change them to "index"; doing so would very likely introduce severe bugs into the program. This is because some of the occurences of "x" may refer to other, different variables that are also named "x", or may refer to other kinds of object also named "x", such as a function or data type.

If the AST stores the relationship between each variable declaration and the corresponding references, automatically renaming variables becomes a simple task to implement—simply rename the declaration and each reference from "x" to "index".

In practice, any useful analysis or transformation of source code will rely upon a knowledge of a comprehensive set of relationships between parts of source code.

3.3 Parsing and Static Semantic Analysis

A parser creates an AST that models the hierarchical relationships between parts of a program. The AST serves as a base source code model from which additional useful relationships can be computed, such as the relationship between variable declarations and references, and the data types of expressions. These additional relationships, strictly speaking, are computed and stored by what is sometimes called *static semantic analysis* of the AST.

The additional relationships determined by static semantic analysis usually connect parts of the AST in a "sideways" manner rather than the parent-child connections created by the parser. For example, a reference of a variable is not nested underneath its declaration—it occurs in a different section of code, perhaps even in a different file. For this reason, we refer to these connections as *annotations* and refer to the result of static semantic analysis as an *annotated AST*.

The basic relationships created by static semantic analyzer serve as a prerequisite for the more advanced types of analysis and transformations performed by modernization tools. For this reason, it is beneficial to include static semantic analysis as part of the parsing process, so that the AST produced by the parser is immediately ready for the next stage of a modernization process.

4 TYPES OF PARSING AND PARSERS

Now that we have described the function of parsing and its role in modernization tools, we will examine some parsing technologies used in legacy modernization projects.

4.1 Text Searching

Text searching refers to the processing of source files directly, using simple search or search-and-replace operators. Text searching tools for software are often based on the use of a technology called *regular expressions*. Examples of technologies for text searching include the Unix tools sed and awk, the Perl scripting language, various shell scripting languages, and the search-and-replace functions of advanced text editors.

The key difference between text searching and parsing is that text searching does not build a comprehensive tree model of the source code. Instead, simple text patterns are specified using regular expressions, and some action is performed each time an occurrence of the text pattern is found. For example, if the tool is intended to count the number of lines in a file, it can search for all occurences of the carriage return or line feed characters and count them.

Text searching tools are generally much faster than tools which use parsing. However, it is theoretically impossible to use regular-expression-based text searching tools to parse conventional programming languages and create an AST model. Because text searching tools cannot build an AST model, they cannot understand the relationships between parts of software.

Therefore, text searching tools are limited to only the simplest kinds of analysis, such as line counting. Some organizations have attempted to use text searching to perform more complex analyses, such as cross-referencing variable declarations and references, but this results in analyzers that produce inaccurate results due to the lack of a model of the relationships in the source code.

4.2 Hand-Written Parsers

Given a description of the syntax rules of a programming language, it is possible to manually write a parser for the language. However, this is a complex task for all but the simplest languages, and there are a variety of parser generators that can be used to automate the creation of a parser from a suitable grammar.

Most hand-written parsers use a technique called *recursive descent* parsing, which is powerful enough to handle common programming languages.

The advantages of hand-written parsers include:

- There is no requirement to obtain and use a parser generator
- The developer has complete control over the actions performed by the parser, including the creation of an AST and possibly other data structures.

The disadvantages of hand-written parsers include:

- They require much more coding and hence more development time
- A hand-written parser for a typical programming language such as C, COBOL or PL/1 is very complex and difficult to maintain
- There is no automatic checking to determine if the parser has bugs.

4.3 Parser Generators

Most parsers for programming languages are created using parser generators, which take a grammar as input and produce a parser as output.

The advantages of using a parser generator include:

- Development time is reduced because the developer only has to create (or find) a grammar, the parser is created automatically from the grammar.
- Maintaining a grammar is much easier than maintaining a hand-written parser.
- The parser generator can check for errors in the grammar that might otherwise turn into bugs in a hand-written parser.

4.4 Types of Parser Generators

There are several parser generator algorithms and an in-depth discussion of parser generators is beyond the scope of this paper.

The two main types of parser generators currently in use are called *recursive descent* parser generators and *LALR* parser-generators. A recursive descent parser generator translates a grammar into a "top-down" parser whose functions closely resemble the production rules in the grammar. An LALR parser generator translates a grammar into a "bottom-up" parser that is driven by a set of tables called parse tables.

While the parsers produced by recursive descent and LALR parser generators have comparable speed, LALR parser generators have some advantages that make them preferable in many or most cases:

- LALR parser generators can process a wider variety of grammars than recursive descent parser generators. Recursive descent parser generators impose some important restrictions on the grammar that force grammar developers to modify grammars in somewhat unnatural ways to meet the rules of the parser generator
- LALR parser generators generate parsers that can provide better error messages when parsing a source code file with a syntax error. This is because LALR parsers detect syntax errors as early as possible in the file.
- LALR parser generators can also provide better error messages when compiling a grammar that contains an error such as an ambiguity.

5 ADVANCED PARSING FEATURES FOR MODERNIZATION TOOLS

This section describes some novel and advanced parsing features that have been demonstrated to dramatically reduce the cost and schedule for legacy modernization projects.

5.1 Parser Development in Legacy System Modernization Projects

In the installed base of business-critical legacy applications worldwide, hundreds of different programming languages are in use, and some—such as COBOL—have dozens of distinct dialects. A single large legacy system might use 10 or more different programming languages in different system modules. Some organizations are running legacy systems that still use obsolete "in-house" languages that were created by the organization itself, often decades ago.

It is often necessary, at the beginning of a modernization project, to create or modify parsers to handle the programming languages used in the legacy system. The time and cost required to develop necessary parsers is an important factor in determining overall project costs and schedules. In addition, the completeness and usability of the ASTs produced by the parsers will strongly affect the cost and time to develop the automated analysis and transformation tools needed for the project.

Therefore, the power of the parsing system used in a modernization project is often a key factor in determing the success or failure of the project. In the sections below, we describe new parser generator features that lead to dramatic improvements in these projects.

5.2 Higher-Level Grammar Languages

BNF (Backus-Naur form) notation has long been the standard way to write grammars, and many parser generators require as input grammars written in BNF or a notation very similar to BNF.

However, some advanced parser generators provide higher-level grammar languages that make grammars much shorter and easier to read. This reduces the development time for creating parsers, and makes the grammars easier to understand and maintain. The effect is similar to the benefit of a high-level language such as Java compared to assembler code.

Some of the features provided by these higher-level grammar languages include:

Regular-Right-Part Productions. This feature allows the developer to write productions that use regular expressions to describe common syntax features of programming languages such as "a + b + c", in which a sequence of elements are separated by the same keyword—in this case, the "+" keyword.

Automated Construction of AST. In some parser generators, it is necessary to write code in grammar productions to build an AST node whenever the parser uses that production. If the parser generator supports the automatic construction of ASTs, the parser generator will automatically generate that code, which frees the programmer from this extra work.

Grammar Inheritance. An advantage of object-oriented programming is that behavior common to a set of classes can be put into a common "parent class" and then be *inherited* by each "subclass". This inheritance feature reduces development time, avoids code duplication, and makes programs more maintainable.

An advanced parsing system can provide an analogous *grammar inheritance* feature. This allows a grammar developer to factor out the syntax rules common to a set of grammars, and put those rules into a grammar from which the other grammars inherit.

Grammar inheritance is particularly useful for when developing grammars for multiple dialects of a language such as COBOL. The common syntax rules can be put into a *BaseCobol* grammar, and then more specialized grammars such as *BullCobol*, *TandemCobol*, and *MicroFocusCobol* can each inherit from the *BaseCobol* grammar.

Operator Precedence Tables. The *operator precedence* rules for a programming language eliminate ambiguity regarding the order of evaluation in expressions when parentheses are not used. For example, when the programmer writes an expression such as "a + b * c", in most languages the multiplication "b * c" should be performed first, and the result added to "a".

In many parser generators, the developer must "encode" the operator precedence rules in extra productions in a way that makes the grammar larger and more complex. Operator precedence tables are a much simpler mechanism in which the operators of the language ("+", "*", "-" etc) are listed in a table, in order of their precedence.

5.3 Capturing Code Formatting and Comments in the AST

When creating an AST, most parsers discard information about the code formatting (eg, indentation) and the documentation (comments) that is intermixed with the code.

However, for legacy system modernization, the comments and formatting are vital information that must be preserved in the AST. For example, most customers will want the comments to be preserved when an application is modernized. And if a program is being refactored while staying in the same language, customers prefer that the formatting of unmodified code be preserved.

5.4 Advanced Object-Oriented AST Representation

Closely related to the parsing system is the data representation used to store and manipulate the ASTs produced by the parsers. A powerful, modern approach is to use object-oriented design to represent the elements (variables, functions, statements, expressions etc) of a programming language.

In this approach, each programming language element has a corresponding object class to represent instances of that element. For example, a class named *Assign* could be used to represent all assignment statements, and the *Assign* class would inherit from a parent class named *Statement*. The diagram below shows elements of such an object-oriented AST representation for a toy language:

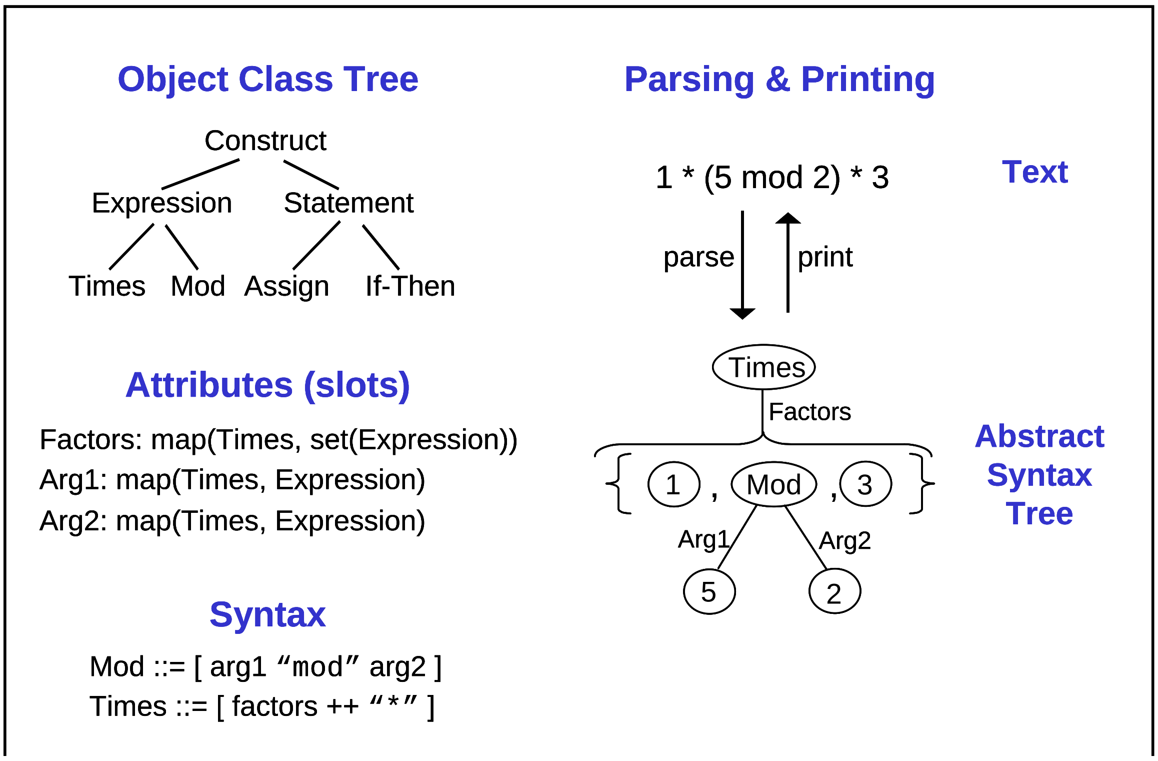

Figure 5: Object-oriented AST representation for a simple language

The object-oriented data repository used in this approach can provide many valuable features for source code analysis and transformation, including:

- Persistent storage of object data to allow ASTs to be saved between sessions.
- A library of reusable functions for performing common operations on ASTs such as copying, comparison, search, and transformation.
- The ability to automatically maintain an relationship defined as the *inverse* of an existing relationship. For example, if a relationship named *References* links a variable declaration to all references to the variable, then a relationship named *ReferenceTo* defined as an inverse will automatically link each reference back to the corresponding variable declaration.

5.5 Automated Printer Generation

Automated printer generation allows a parser generator to generate a printer that converts ASTs back into text files. This is necessary so that ASTs that have been modified (for example, translated into a different language) can be printed to generate the modernized source files.

5.6 Syntax-Based Pattern-Matching Support

This enables writing concise code transformation rules by providing a description of the code before and after the transformation. The benefit is that these "before" and "after" descriptions can be written in the source and target languages of the transformation. For example, a COBOL-to-Java transformation rule could describe the "before" using COBOL syntax and the "after" using Java syntax.

6 SUMMARY

Parsing is a key technology for legacy modernization and especially for tools automating legacy modernization. Because of the wide range of languages encountered in legacy systems, some parser development is often required in modernization projects that use automated tools. The need for parser development, along with the key role played by the ASTs that parsers produce, make the choice of parsing technology an important factor that heavily influences the cost and schedule for these modernization projects.

The use of an advanced parsing technology based on a parser generator will yield the best results for most projects. LALR parser generators are well-suited to the technical challenges presented by legacy software modernization.

Novel features of parsing systems including higher-level grammar languages, object-oriented AST representation, the ability to capture formatting and comments in ASTs, automated printer generation, and syntax-based pattern-matching support provide a dramatic reduction in cost, schedule and risk for legacy modernization projects.

metaware.
the complete IT modernization solution

www.metaware.fr
Tel. +33 1 30 15 60 00
Fax. +33 1 30 15 06 71

Global Headquarter:
1. Parc des Grillons
60, rte de Sartrouville
78230 Le Pecq Cedex – FRANCE