Attribute Grammars and Parsers
for Chunk-Based Binary File Formats

William Underwood
Sandra Laib

ICL/ITDSD Technical Report 12-01

December 2012

Information Technology and Decision Support Division
Information and Communications Laboratory
Georgia Tech Research Institute
Atlanta, Georgia

This research was sponsored by the Army Research Laboratory (ARL) and the Applied Research Division of the National Archives and Records Administration (NARA) and was accomplished under Cooperative Agreement Number W911NF-10-2-0030. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Army Research Laboratory, NARA, or the U.S. Government. The U.S. government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation hereon.
ABSTRACT

The capability to validate and view or play binary file formats, as well as to convert binary file formats to standard or current file formats, is critically important to the preservation of digital data and records. This report describes the extension of context-free grammars from strings to binary arrays. Binary files are arrays of data types such as long and short integers, floating-point numbers, and pointers as well as characters. The concept of an attribute grammar is extended to these context-free binary array grammars. Binary array attribute grammars have been used to define a number of chunk-based file formats. These grammars consist of grammar rules for specifying the structure or layout of a file format and lexical rules for defining the binary data types of the file format. A parser generator has been used with some of these grammars to generate parsers for binary file formats. The capability has been developed to display the parse trees of binary array attribute grammars as pretty printed nested lists.

The significance of success in this research task is that if binary file formats can be specified with binary array attribute grammars, then only one parsing generator is needed to generate the parsers for verifying that a file conforms to a particular format. Similarly, the same parsing generator can possibly be used for conversion of legacy file formats to current or standard formats. Finally, the same parser generator can possibly be used for generating viewers/players for most file formats. This would increase the likelihood of preserving and making available into the indefinite future those digital records encoded in binary file formats.
TABLE OF CONTENTS

1 INTRODUCTION ......................................................................................................................... 1
   1.1 BACKGROUND ...................................................................................................................... 1
   1.2 PURPOSE ............................................................................................................................ 1
   1.3 SCOPE .................................................................................................................................. 1

2 SPECIFICATION OF BINARY FILE FORMATS ........................................................................ 1

3 CONTEXT-FREE GRAMMARS AND ATTRIBUTE GRAMMARS .............................................. 3
   3.1 CONTEXT-FREE GRAMMARS FOR ARRAYS OF DATA TYPES ........................................... 4
   3.2 ATTRIBUTE GRAMMARS ...................................................................................................... 4

4. GRAMMARS FOR CHUNK-BASED BINARY FILE FORMATS ............................................ 5
   4.1 INTERLEAVED BITMAP (ILBM) ............................................................................................ 6
   4.2 RIFF WAVEFORM PCM AUDIO FILE FORMAT (WAVE) ................................................... 9

5 RECURSIVE DESCENT PARSERS FOR BINARY FILE GRAMMARS ..................................... 10

6. A PARSER GENERATOR FOR BINARY FILE FORMATS .................................................... 12
   6.1 ANTLR ................................................................................................................................. 12
   6.2 USING ANTLR TO GENERATE A PARSER FOR A BINARY FILE FORMAT ............................. 12
       6.2.1 An ANTLR Grammar for an ILBM Chunk-based File Format ....................................... 13
       6.2.2 The Lexer and Parser .................................................................................................. 17
   6.3 IMPROVEMENTS TO THE LEXICAL ANALYSIS OF DATA TYPES .................................... 18
   6.4 GENERATING PARSE TREES FOR BINARY ARRAY GRAMMARS .................................... 33
       6.4.1 How ANTLR4 Generates Parse Trees for String Attribute Grammars ............................ 33
       6.4.2 Using ANTLR4 to Generate Parse Trees for Binary Attribute Grammars ..................... 34
       6.4.3 Extensions to ANTLR4 to Generate Pretty Printed Parse Trees with Data Type Values ...... 35
       6.4.4 How ANTLR4’s Attribute Grammars, Semantic Predicates and Action Clauses Help with Parsing and Validation ................................................................. 38

7 RELATED RESEARCH ............................................................................................................. 40

8. CONCLUSION .......................................................................................................................... 41

REFERENCES .............................................................................................................................. 43

APPENDIX A: BINARY ARRAY GRAMMARS FOR CHUNK-BASED BINARY FILE FORMATS ....... 53
   A.1 ELECTRONIC ARTS INTERCHANGE FILE FORMAT (IFF) .................................................... 53
       A.1.1 Interleaved Bitmap (ILBM) ............................................................................................ 53
       A.1.2 8-bit Sampled Voice (BSVX) ......................................................................................... 53
   A.2 STANDARD MIDI FILE FORMAT ......................................................................................... 54
   A.3 AUDIO INTERCHANGE FILE FORMAT ................................................................................ 54
   A.4 RESOURCE INTERCHANGE FILE FORMATS ....................................................................... 56
       A.4.1 RIFF-Waveform (WAVE) .............................................................................................. 56
       A.4.2 WAVEFORMATEX ........................................................................................................ 56
   A.5 JPEG .................................................................................................................................... 56
List of Figures

Figure 1. File format (Layout) of a RIFF container for a Webp Image.................................................. 2
Figure 2. ILBM File ham.pic Displayed .................................................................................................. 6
Figure 3. An Attribute Grammar for an ILBM Chunk-based File Format ........................................... 7
Figure 4. A parse tree for the ILBM file ham.pic ................................................................................. 8
Figure 5. Binary Array Grammar for RIFF WAVEFORM PCM File Format ........................................ 10
Figure 6. Top-Level Grammar Rule ...................................................................................................... 10
Figure 7. Top-level Java program of a recursive descent parser ......................................................... 11
Figure 8. An ANTLR Grammar for the ILBM File Format................................................................. 16
Figure 9. A Java Application for Parsing an ILBM File ..................................................................... 17
Figure 10. Output of the ILBM Parser Applied to the File BLK.IFF .................................................. 18
Figure 11. Lexer for all Context-free Binary Array Grammars .............................................................. 19
Figure 12. Data Translation and Other Grammar Utilities ................................................................. 21
Figure 13. Datatype rules used by all binary array attribute grammars........................................... 23
Figure 14. The ILBM File Format Grammar ....................................................................................... 26
Figure 15. The Wave_Form File Format Grammar ........................................................................... 32
Figure 16. Parse tree shown as a nested list ....................................................................................... 33
Figure 17. Parse tree shown as a graph ............................................................................................... 33
Figure 18. ILBM File Docking.bru ....................................................................................................... 34
Figure 19. Parse Tree Created by ANTLR4 ....................................................................................... 35
Figure 20. Pretty Printed Parse Tree of ILBM Formated File Docking.bru ......................................... 38
1 Introduction

1.1 Background
Automated tools are required for identifying and validating the formats of the huge number of files ingested into digital data and record archives. Automated tools are also needed for viewing/playing text and binary file formats, and for converting legacy and obsolete file formats to standard or current formats. Technologies such as data description languages have emerged to address these preservation challenges [Dunckley et al 2007].

Context-free grammars have been used to specify the syntax of programming languages. Attribute grammars provide a framework for formally specifying the semantics of a language based on its context-free grammar and for addressing the mildly context-sensitive features of programming languages such as agreement of the data types of variables in expressions with data type declarations. Such grammars have parsing/translation algorithms that can be used for syntax-checking and interpretation or translation of the languages the grammars define.

1.2 Purpose
The research question addressed by this research is whether it is possible to extend the context-free grammars used to specify the syntax of programming languages to the specification of binary file formats and to use these grammars with parsers for validating the file formats of binary files.

1.3 Scope
The next section discusses the traditional approach to specifying file formats and two of the major families of binary file formats. In section 3, extensions to the concepts of context-free grammars and attribute grammars are described that enable the specification of binary file formats. In section 4, examples of binary array attribute grammars for a chunk-based binary file formats are presented. In section 5, recursive descent parsers for these kinds of grammars are described. In section 6, experience in using ANTLR, a parser generator for LL(k) string grammars, in generating parsers for recognizing the formats of binary files is discussed. This includes the extension of ANTLR to work with grammars with data types other than strings and the creation of parse trees represented as pretty printed nested lists. In section 7, related research is described. Section 8 summarizes the results.

2 Specification of Binary File Formats
Traditionally, a file (or record) layout was a form showing how fields were positioned in a file (or record). The fields are named and have as attributes a data type, length and sometimes a constant value or range of values. Fields are offset at addresses relative to the beginning of a file.
File formats are still specified in this manner. For instance Figure 1 shows the format layout of a RIFF container for the VP8 codec of a Webp image [Google 2010].

<table>
<thead>
<tr>
<th>Offset</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>&quot;RIFF&quot; 4-byte tag</td>
</tr>
<tr>
<td>4</td>
<td>long integer: size of data chunk starting at offset 8</td>
</tr>
<tr>
<td>8</td>
<td>&quot;WEBP&quot; the form-type signature</td>
</tr>
<tr>
<td>12</td>
<td>&quot;VP8&quot; 4-bytes tag, describing the raw video format used</td>
</tr>
<tr>
<td>16</td>
<td>long integer: size of the raw VP8 image data chunk, starting at offset 20</td>
</tr>
<tr>
<td>20</td>
<td>the VP8 image data</td>
</tr>
</tbody>
</table>

Figure 1. File format (Layout) of a RIFF container for a Webp Image.

Many binary file formats are specified using pseudo-regular expressions or pseudo-EBNF notation. EBNF (Extended Backus-Naur Form) is a notation for expressing context-free grammars in a compact, human readable way. Pseudo-EBNF is similar in concept to pseudocode, which is a high-level description of a computer algorithm that is intended for human understanding, but that omits details that would be necessary for computer execution. The pseudo-EBNF (or regular expression) specification of a file format is augmented with a natural language description of the details that are not actually expressible in the EBNF (or regular expression) notation. These details are often the context-sensitive relationships of the declared size of an array to its actual length, or the relationship of an address pointer to the actual location of the data pointed to in the file. These context-sensitive relationships are not expressible in a context-free grammar (or EBNF notation). It is a goal of this research to extend the EBNF notation and concept of a context-free grammar to include the details that are necessary to precisely specify binary file formats that include these context-sensitive features.

Binary file formats with a similar file structure are referred to as a family of file formats. There are two families of binary file formats that are readily distinguished, the chunk-based and the directory-based file formats.

Chunk-based file formats were created by Electronic Arts and Commodore-Amiga as the Interchange File Format (IFF) [Morrison 1985]. A chunk consists of an ID tag, a size, and size bytes of data. The data may include chunks called subchunks. Subchunks have the same structure as chunks. Subchunks can have subchunks. Chunk-based file formats are primarily used as containers for multimedia, but have also been used for word processing files including pictures and other figures. File format specifications for chunk-based formats may use other terms than
chunks in describing the format, for instance, "atoms" in QuickTime/MP4, "segments" in JPEG, “boxes” in JPEG 2000, and “tagged data representations” in AutoCAD DXF.

The Audio Interchange File Format (AIFF) developed by Apple computer in 1988 was based on Electronic Arts Interchange File Format. Apple Core Audio Format (CAF), Apple’s replacement format for AIFF, remains a chunk-based format [Apple Computer 2005]. The Resource Interchange File Format (RIFF) introduced by IBM and Microsoft [1991] is also based on Electronic Arts’ Interchange File Format. The Microsoft container formats like Audio-Video Interleave (AVI) and Waveform PCM (WAV) use RIFF as their basis. WebP, a picture format recently introduced by Google [2010], also uses RIFF as a container.

Microsoft’s Advanced Systems Format (ASF) [Microsoft 2004] is also a chunk-based container format. The most common file formats contained within an ASF file are Windows Media Audio (WMA) and Windows Media Video (WMV). Microsoft’s Binary Interchange File Format (Microsoft Excel’s File Format) is chunk-based [Rentz 2008].

Another family of binary file formats is directory-based. A directory-based file format consists of one or more directory tables which contain one or more directory entries. A directory entry specifies where the actual data is located. This is the scheme used in TIFF files, OLE (Microsoft Object Linking and Embedding) files, OASIS OpenDocument and Microsoft Open Office files.

3 Context-Free Grammars and Attribute Grammars

Context-free grammars have been widely used to define programming languages. A context-free grammar G is a quadruple <N, Σ, S, P> where:

- N is a finite set of non-terminal symbols,
- Σ is a finite set of terminal symbols,
- S ∈ N is the start symbol,
- P is a set of production rules of the form A → {N ∪ Σ}* where A ∈ N

The set of all strings over a vocabulary Σ is denoted by Σ*. Formally, a string γ is said to directly derive in G a string τ, denoted γ⇒Gτ, if τ can be obtained from γ by replacing a substring α with β, where α⇒β is a production rule in G. A string γ is said to derive τ in G, denoted γ⇒G*τ, if γ0⇒Gγ1⇒G...⇒Gγn for some γ0, . . . , γn such that γ0 = γ and γn = τ.

The language generated by a context-free grammar G is the set L(G) = {w: w ∈ Σ* and S ⇒G* w}.
3.1 Context-Free Grammars for Arrays of Data Types

An array is a data structure consisting of a collection of elements (values or variables), each identified by an index. An array is stored so that the position of each element can be computed from its index. For example, an array of 10 integer variables, with indices 0 through 9, may be stored as 10 (4 byte) words at file addresses 0, 4, 8, …36, so that the element with index i has the address $4 \times i$.

Arrays are used to implement many other data structures, such as lists and strings. In most modern computers, the internal memory and external memory of storage devices (and files on those devices) is a one-dimensional array of data types, whose indices are their addresses.

A data type is a classification of one of various types of data, such as floating-point, integer, character, pointer, or Boolean, that determines the possible values for that type, the operations that can be performed on values of that type, and the way values of that type can be stored. Data types have names such as int16, int32, float, char, bool, ptr. We define a binary file to be an array of values of data of various types.

We define a context-free binary array grammar BAG as a quintuple $<N, D, \Sigma, S, P>$ where:

- $N$ is a finite set of non-terminal symbols,
- $D$ is a set of data types,
- $\Sigma$ is a finite set of binary values of data types D called terminals,
- $S \in N$ is the start symbol,
- $P$ is a set of production rules of the form $A \rightarrow N^*$ and $A \rightarrow t$ where $A \in N$ and $t \in \Sigma$. The first set of productions are called structural rules and the second are called lexical rules.

The set of all binary arrays is $\text{BinaryArrays} = [\text{Indices} \rightarrow \Sigma]$, where $\text{Indices} = \{1, \ldots, \infty\}$.

Formally, a string $\gamma$ is said to directly derive in BAG a string $\gamma'$, denoted $\gamma \Rightarrow_{\text{BAG}} \gamma'$, if $\gamma$ can be obtained from $\gamma$ by replacing a substring $\alpha$ with $\beta$, where $\alpha \Rightarrow \beta$ is a production rule in $G$. A string $\gamma$ is said to derive $\gamma'$ in BAG, denoted $\gamma \Rightarrow_{\text{BAG}}^* \gamma'$, if $\gamma_0 \Rightarrow_{G} \gamma_1 \Rightarrow_{G} \cdots \Rightarrow_{G} \gamma_n$ for some $\gamma_0, \ldots, \gamma_n$ such that $\gamma_0 = \gamma$ and $\gamma_n = \gamma'$.

The language generated by a context-free binary array grammar BAG is the set $L(\text{BAG}) = \{ w : w \in \text{BinaryArrays} \text{ and } S \Rightarrow_{\text{BAG}}^* w \}$.

3.2 Attribute Grammars

Context-free grammars cannot represent context-sensitive aspects of programming languages such as: (1) enforcing the constraint that all variables are declared before they are used, or (2)
checking the number of parameters in a function call against the number in the function’s declaration. Context-free grammars have also been used to define the syntax of English and other natural languages, but they cannot define such context-sensitivity as subject verb agreement in English sentences. A number of extensions of context-free grammars have been proposed to address the context-sensitivity of programming languages and natural languages. These include indexed [Aho 1968], recording [Barth 1979], affix [Koster 1971], VW [van Wijngaarden 1974] and attribute grammars [Knuth 1969, 1971].

Context-free grammars also cannot represent the semantics of programming languages. Knuth [1968, 1971] proposed an extension of context-free grammars termed attribute grammars that addresses the semantics as well as the context-sensitivity of programming languages.

An *attribute grammar* AG is a triple <G, A, AR>, where G is a context-free grammar for the language, A associates each grammar symbol X ∈ (N ∪ Σ) with a set of attributes, and AR associates each production R ∈ P with a set of attribute computation and/or condition rules. A(X), where X ∈ (N ∪ Σ), can be further partitioned into two sets: synthesized attributes S(X) and inherited attributes I(X). AR(R), where R ∈ P, contains rules for computing inherited and synthesized attributes associated with the symbols in the production R. Synthesized attributes are evaluated and passed up from deeper terminals and non-terminals while inherited attributes are passed down from upper non-terminals.

An *L-attributed grammar* is an attribute grammar that uses:

i) synthesized attributes, and

ii) inherited attributes where the inherited attribute depends only on:

a) attributes of the other child nodes to its left

b) inherited attributes of P itself.

L-attributed grammars allow the attributes to be evaluated in one left-to-right traversal of the abstract syntax tree. As a result, attribute evaluation in L-attributed grammars can be incorporated conveniently into top-down parsing.

**4. Grammars for Chunk-based Binary File Formats**

The attribute grammars that we present are a combination of *synthesized* semantic rules (attribute values of the left side of a production are calculated from attribute values of symbols on the right side) and *inherited* rules (attribute values of symbols on the right side of a production are calculated from attribute values of the symbol on the left side).
4.1 InterLeaved BitMap (ILBM)

The author of the Interleaved Bitmap (ILBM) file format recognized that the structure of the file format could be specified using pseudo EBNF rules and C data structure declarations [Morrison 1986]. In this section, the “grammar” of the ILBM file format specification is represented as a context-free binary array grammar with synthesized and inherited attributes.

Figure 2 shows a display of an ILBM file ham.pic. Figure 3 shows an attribute grammar for its chunk-based binary file format. The attribute grammar is a combination of synthesized semantic rules (attribute values of the left side of a production are calculated from attribute values of symbols on the right side) and inherited rules (attribute values of symbols on the right side of a production are calculated from attribute values of the symbol on the left side). The attribute grammar is also an L-attributed grammar. Figure 4 shows a manually generated parse tree for the file ham.pic.
Figure 3. An Attribute Grammar for an ILBM Chunk-based File Format
Figure 4. A parse tree for the ILBM file ham.pic
4.2 RIFF Waveform PCM Audio File Format (WAVE)

The waveform PCM File format was originally specified in an extended BNF notation [IBM & Microsoft 1991]. The chunk size was omitted from the EBNF rules. Figure 5 shows the EBNF rules with the cksize inserted. To convert this grammar into a context-free binary array grammar, data types are added to non-terminals which would be replaced by a binary value were an additional lexical production rule applied.

```xml
<WAVE-form> -> “RIFF” <cksize UINT32> “WAVE”
  <fmt-ck>
    [<fact-ck>]
    [<cue-ck>]
    [<playlist-ck>]
    [<assoc-data-list>]
  <wave-data>
<fmt-ck> -> “fmt” <cksize UINT32> <common-fields> <wBitsPerSample UINT16>
<common-fields> -> <wFormatTag UINT16 value=0x0001>
  <wChannels UINT16>
  <dwSamplesPerSec UINT32>
  <dwAvgBytesPerSec UINT32>
  <wBlockAlign UINT16>
<wave-data> -> {<data-ck> | <wave-list>}
<data-ck> -> data <cksize UINT32> <wave-data>
<wave-list> -> LIST <cksize UINT32> “wavl” {<data-ck> | <silence-ck>}
<silence-ck> -> slnt <cksize UINT32> <dwSamples UINT32>
<fact-ck> -> factxksize <dwFileSize UINT32>
<cue-ck> -> cue <cksize UINT32> <dwCuePoints UINT32> <cue-point>+<cue-point> -> <dwName UINT32>
  <dwPosition UINT32>
  <fccChunk CHAR[4]>
  <dwChunkStart UINT32>
  <dwBlockStart UINT32>
  <dwSampleOffset UINT32>
<playlist-ck> -> “plst” <cksize UINT32> <dwSegments UINT32> <play-segment>+<play-segment> -> <dwName UINT32>
  <dwLength UINT32>
  <dwLoops UINT32>
<assoc-data-list> -> “LIST” <cksize UINT32> “adtl” <lab1-ck><note-ck><ltxt-ck><file-ck>
```
5 Recursive Descent Parsers for Binary File Grammars

A recursive descent parser is a top-down parser built from a set of recursive procedures where each procedure implements one of the production rules of the grammar. A predictive parser is a recursive descent parser that does not require backtracking. Predictive parsing is possible only for the class of LL(k) grammars, which are the context-free grammars for which there exists some positive integer k that allows a recursive descent parser to decide which production to use by examining only the next k tokens of input.

Figure 6 shows the top-level grammar rule of the binary array attribute grammar for the ILBM file format. The interpretation of this rule is: The first 4 bytes of the file format contain the characters “FORM”. Next is a 4-byte (32-bit) unsigned integer indicating a chunk size that is the size of the remainder of the file. The next 4 bytes contain the characters “ILBM”. This must be followed by a propertyChunk, a dataChunk and a BODY chunk.
Figure 7 shows the top-level Java program named ilbm in a top-down recursive descent parser for the ILBM grammar. It is manually constructed from the top-level grammar rule. There are also Java programs corresponding to <propertyChunk>, <dataChunk> and <BODY>.

The property chunks of an ILBM file must contain a BitmapHeader chunk. The attribute foundBMHD() is passed back to the program with a value true or false, indicating whether a BitmapHeader (BMHD) was found.

The recursive descent parsers for the binary array grammars defined in this paper do not require a separate lexical analyzer (lexer) to identify the data of a file prior to parsing. The data types are predicted by the grammar rules. The top-down parser defines which data types are expected next so that the tokens (data type values) are identified while parsing.

```java
private void ilbm()
{
    try
        {
            if(nextTerminal(STRING, 4).equals("FORM"))
                {
                    long cKsize = (long)nextTerminal(INT32).value();
                    if(nextTerminal(STRING, 4).equals("ILBM"))
                        {
                            if(propertyChunks().foundBMHD())
                                {
                                    dataChunk();
                                    body();
                                }
                            else
                                {
                                    throw new Exception("BMHD not found");
                                }
                        }
                    else
                        {
                            throw new Exception("ILBM not found");
                        }
                }
            else
                {
                    throw new Exception("PGM not found");
                }
            }
        catch (Exception e)
            {
                e.printStackTrace();
            }
}
```

*Figure 7. Top-level Java program of a recursive descent parser*

The semantic rules of an L-attributed grammar can be included in the recursive descent parser to check for context-sensitive aspects of the grammar such as chunk size or image dimensions and to use these values in parsing the chunk data or images. The pointers to file addresses that occur in the file formats might be handled by pushing the current address in the file onto a pushdown
stack. When the parser exhausts the procedures implementing production rules corresponding to
the pointed to location, the topmost address on the pushdown stack is popped and the procedure
responding to the nonterminal at that position is executed.

Recursive descent parsers with a pushdown stack can be used with binary array grammars to
create file format recognizers. A recognizer (syntax checker or validator) is a parser that reads a
file and generates an error if the file does not conform to the syntax specified by the grammar.

6. A Parser Generator for Binary File Formats

What we want is a parser generator whose input is a binary array attribute grammar for a
particular binary file format, and whose generated output is the Java source code of a recursive
descent parser for the class of binary file formats specified by the grammar. Such a parser
generator does not yet exist, but there is a widely used parser generator for attribute grammars
for string-based languages.

6.1 ANTLR

ANTLR (ANother Tool for Language Recognition) is a parser generator that uses LL(k) parsing
[Parr & Quong 1995; Parr 2007]. ANTLR takes as input an attribute grammar that specifies a
language and generates as output source code for a parser for that language. ANTLR supports
generating code in a number of the programming languages including C, Java, JavaScript and
Python.

ANTLR also generates a Lexical analyzer (lexer) from lexical rules in the grammar. A lexer is a
program that converts a sequence of characters into tokens. A lexer is not needed to parse binary
file formats, because the data types predicted by the parser are the tokens of the grammar.

6.2 Using ANTLR to Generate a Parser for a Binary File Format

It is possible to use ANTLR to test our binary array grammars in recognizing the chunk-based
and directory-based binary file formats. This is accomplished by writing a lexer that treats each
input byte in a file as a character token. Then functions are created for each data type in the
binary file grammar that convert the appropriate number of character tokens to binary data types,
e.g., int 16, int32, etc. This is effective, if someone clumsy, and has allowed us to test our
attribute grammars for binary file formats as well as to demonstrate the feasibility of creating a
parser generator for binary array grammars.
6.2.1 An ANTLR Grammar for an ILBM Chunk-based File Format

Figure 8 shows a specification for the ILBM file format that is input to ANTLR. The output of ANTLR is a parser for ILBM files.

```
grammar iblm;
options {
  language = Java;       // The programming language of the output parser
  TokenLabelType = CommonToken;
}
// The text contained in @header is placed at the top of the parser file.
@header {
  package gtri.org.grammars.chunk;
  import java.io.UnsupportedEncodingException;
}
// The text in @lexer::header is placed at the top of the lexer file.
@lexer::header {
  package gtri.org.grammars.chunk;
  import java.io.UnsupportedEncodingException;
}
// The text in the @members section is placed inside the parser class.
@members {
  // getUByteValue inputs the token of a Byte terminal and returns an
  // integer value.
  public int getUByteValue(Token tok) throws UnsupportedEncodingException {
    int intValue = 0;
    char charValue = tok.getText().charAt(0);
    intValue = (int) charValue;
    intValue = intValue & 0x00ff;
    return intValue;
  }
}
// Grammar Rules used to create the Parser. Start symbol iblm
iblm returns [boolean result] :
  FORM ckSize ILBM
  // Print out ILBM chunksize.
  { System.out.println("ID: ILBM Size: "+$ckSize.value); }
  // Semantic predicate that ensures that at least on BMHD is found.
  propertyChunks {$propertyChunks.foundBMHD == true}?
    dataChunks?
      body?
      { $result = true; }
  // The bmhd, cmap and camg property chunks can occur in any order.
  // At least one BMHD data chunk must be present.
  propertyChunks returns [boolean foundBMHD] :
    (bmhd { $foundBMHD = true; } | cmap | camg)+;
  bmhd :
    BMHD ckSize
    { System.out.println("ID: BMHD Size: " +$ckSize.value); }
    bitmapHeader;
  bitmapHeader :
    width = uWordValue
```
height = uWordValue
xPosition = wordValue
yPosition = wordValue
nPanes = uByteValue
masking = uByteValue
compression = uByteValue
reserved1 = uByteValue
transparentColor = uWordValue
xAspect = uByteValue
yAspect = uByteValue
pageWidth = wordValue
pageHeight = wordValue

// Print out values in the bitmap structure.
{
    System.out.println("         Width: " + $width.value);
    System.out.println("         Height: " + $height.value);
    System.out.println("         X Position: " + $xPosition.value);
    System.out.println("         Y Position: " + $yPosition.value);
    System.out.println("         Number of Panes: " + $nPanes.value);
    System.out.println("         Masking:" + $masking.value);
    System.out.println("         Compression:" + $compression.value);
    System.out.println("         Reserved1:" + $reserved1.value);
    System.out.println("         Transparent Color:" + $transparentColor.value);
    System.out.println("         X Aspect:" + $xAspect.value);
    System.out.println("         Y Aspect:" + $yAspect.value);
    System.out.println("         Page Width:" + $pageWidth.value);
    System.out.println("         Page Height:" + $pageHeight.value);
}

cmap returns [long value]
// Initialize the index value i in the cmap function.
@init {  int i = 0; }
:
    CMAP ckSize
    // Print out chunk size.
    {  $value = $ckSize.value;
        System.out.println("ID: CMAP Size: " + $ckSize.value);
    }
    (colorRegister
    // Print out color values.
    {  System.out.println("         Color[" + i++ + "]: {" + $colorRegister.redValue + ", " + $colorRegister.greenValue + ", " + $colorRegister.blueValue + "}");
    })+
    colorRegister returns [int redValue, int greenValue, int blueValue]
    : red = uByteValue green = uByteValue blue = uByteValue
    {  $redValue = $red.value;
        $greenValue = $green.value;
        $blueValue = $blue.value;
    }
    camg
    :  CAMG ckSize longValue
    {  System.out.println("ID: CAMG Size: " + $ckSize.value);
System.out.println("  View Modes: 0x" +
Long.toHexString($longValue.value));
}
// The crng and ccrt data chunks
dataChunks:
  | crng
  | ccrt;
crng:
  CRNG ckSize
  { System.out.println("ID: CRNG Size: " + $ckSize.value); }
cRange:
  pad1 = wordValue {pad1.value == 0}?
  rate = wordValue
  active = wordValue
  low = uByteValue
  high = uByteValue;
ccrt:
  CCRT ckSize
  { System.out.println("ID: CCRT Size: " + $ckSize.value); }
cycleInfo:
  direction = wordValue
  start = uByteValue
  end = uByteValue
  seconds = longValue
  microseconds = longValue
  pad = wordValue {pad.value == 0}? ;
body:
  BODY ckSize
  { System.out.println("ID: BODY Size: " + $ckSize.value); }
  { System.out.println("Number of bytes: " + $data.value); }
  {$data.value == $ckSize.value}?;
// defines the data chunk of the body chunk
data returns [long value]
  : (b+=BYTE)+ {b.size() > 0}?
    { $value = $b.size(); };
// The cksize is the value of a long integer.
ckSize returns [long value]
  : longValue
    {$value = $longValue.value;};
// defines the longValue data type in terms of BYTE terminals
longValue returns [long value] throws UnsupportedEncodingException
  : h1 = BYTE h2 = BYTE l1= BYTE l2 = BYTE
    {$value = (long)
      getUByteValue($h1) << 24 |
      getUByteValue($h2) << 16 |
      getUByteValue($l1) << 8  |
      getUByteValue($l2));
    }
catch[RecognitionException re]{
    reportError(re);
    recover(input,re);  }
catch[UnsupportedEncodingException uue]{

System.out.println("Unsupported Encoding Exception");
// defines the unsigned word data type
uWordValue returns [int value]
: b1 = BYTE b2 = BYTE
{$value = (int)
( getUByteValue($b1) << 8 | getUByteValue($b2));
};
catch[RecognitionException re]{
reportError(re);
recover(input,re);
}
catch[UnsupportedEncodingException uee]{
System.out.println("Unsupported Encoding Exception");
}
// defines the signed word data type
wordValue returns [int value]
: b1 = BYTE b2 = BYTE
{$value = (short)
( getUByteValue($b1) << 8 | getUByteValue($b2));
};
catch[RecognitionException re]{
reportError(re);
recover(input,re);
}
catch[UnsupportedEncodingException uee]{
System.out.println("Unsupported Encoding Exception");
}
// defines the unsigned word data type
uByteValue returns [int value]
: BYTE
// calls the getUByteValue from a BYTE token
{$value = (int) getUByteValue($BYTE);
};
catch[RecognitionException re]{
reportError(re);
recover(input,re);
}
catch[UnsupportedEncodingException uee]{
System.out.println("Unsupported Encoding Exception");
}
// Terminal rules used to create the Lexer
FORM : 'FORM' ;
ILBM : 'ILBM' ;
BMHD : 'BMHD' ;
CMAP : 'CMAP' ;
CAMG : 'CAMG' ;
CRNG : 'CRNG' ;
CCRT : 'CCRT' ;
BODY : 'BODY' ;
BYTE : . ;

Figure 8. An ANTLR Grammar for the ILBM File Format.
6.2.2 The Lexer and Parser

When provided the described in the previous section, ANTLR generates the `ilbmlexer` and `ilbmparser` Java programs. The Java application shown in Figure 9 uses these programs to parse the ILBM file BLK.IFF (a black rectangle) and extract the metadata shown in Figure 10.

```java
package org.gtri.grammars.chunk;
import java.io.IOException;
import org.antlr.runtime.ANTLRFileStream;
import org.antlr.runtime.ANTLRFileStream;
import org.antlr.runtime.ANTLRStringStream;
import org.antlr.runtime.CharStream;
import org.antlr.runtime.CommonTokenStream;
import org.antlr.runtime.RecognitionException;
import org.antlr.runtime.TokenStream;
import gtri.org.grammars.chunk.iblmLexer;
import gtri.org.grammars.chunk.iblmParser;

public class Test {
    static String filePath = "C:\Users\sl170\Pictures\IFF-ilbm\BLK.IFF";
    public static void main(String[] args) throws RecognitionException, IOException {
        ANTLRFileStream stream = new ANTLRFileStream(filePath, "ISO-8859-1");
        iblmLexer lexer = new iblmLexer(stream);
        TokenStream tokenStream = new CommonTokenStream(lexer);
        iblmParser parser = new iblmParser(tokenStream);
        System.out.println("FILE: " + filePath);
        System.out.println("Successfully parsed = " + parser.iblm());
    }
}
```

Figure 9. A Java Application for Parsing an ILBM File
Figure 10. Output of the ILBM Parser Applied to the File BLK.IFF

6.3 Improvements to the Lexical Analysis of Data Types

The ANTLR Lexer returns tokens made up of characters. For the simple sentence “This is a simple string.” could return ten tokens, one token for each word, one token for each space and one token for the period. The number of tokens the lexer returned would depend on the grammar it was built from. The previous version of the ilbm grammar also created a lexer that groups bytes intermixed with certain strings. Examples of some of the strings that were returned were “FORM”, “ILBM”, “CAMG”, “BODY”. In some cases this worked fine. But in other cases it caused problems. Whenever the lexer encountered a ‘C’ it expected the next byte to be an ‘M’, ‘A’, ‘R’, or another ‘C’. The lexer returned an error if the ASCII equivalent of the next byte was
not what it expected. This error would state what it was expecting and what it encountered instead. The problem occurred because the same byte that can represent the ASCII ‘C’ could also occur as part of a set of bytes that were supposed to represent a number. Another problem that occurred with the previous grammar, was that the specification for this file type allowed for unknown chunks as long as they had a chunk id that consisted of four capital letters followed by the size of the chunk and the data in the chunk.

The lexer created from a chunk-based binary array attribute grammar, needs to simply return an array of bytes. It needs to leave it up to the parser as to whether a group of bytes represents an ASCII string or a number. Figure 11 shows a grammar that ANTLR uses to create a Lexer that returns an array of bytes.

```
lexer grammar binaryArray;

//Terminal rules used to create the Lexer

BYTE
  : . // The period is used to match any byte
    ;
```

Figure 11. Lexer for all Context-free Binary Array Grammars

Applications store data to a file using groups of bytes. Some applications store the most significant byte of numeric data first. This is called big-endian representation. Other applications store the least significant byte first. This is called little-endian representation. The applications that store data in ILBM files use little-endian representation. To make use of this stored numeric data, the data types that the bytes represent have to be translated into the appropriate numeric representation.

A set of JAVA utilities have been created that can be referenced by binary attribute grammar rules to return values of these big and little-endian data types. There are also data types that return a string of characters and one that returns the hex representation of the bytes as they were stored in the file. Many chunk-based file specifications allow for the addition of unknown chunks. These chunks of data may contain data that is used by one application but can be ignored by other applications. The data types of the data in these chunks are not known, but the size of the array of data that they contain is known. In this case, if the data contains only ASCII printable characters, the ASCII representation is returned. In the case that the data contain non-printable characters, the hex representation of the data is returned. There is a utility for checking whether a number is odd or even. This utility is needed because odd-sized chunks in chunk-based files often have a pad byte added. These functions are shown in Figure 12.
public static boolean isOdd(long value) {
    return ((value % 2) != 0);
}

public static boolean isPrintable(String str) {
    if (str == null) {
        return false;
    }
    int sz = str.length();
    if (str.charAt(sz - 1) == 0)
        sz--;
    for (int i = 0; i < sz; i++) {
        if (isPrintable(str.charAt(i)) == false) {
            return false;
        }
    }
    return true;
}

public static boolean isPrintable(char ch) {
    return ((ch >= 32 && ch <= 127) ||
            (ch >= 8 && ch <= 10) ||
            (ch >= 12 && ch <= 13));
}

public static boolean isValidZString(String str) {
    if (str == null) {
        return false;
    }
    int sz = str.length();
    return (str.charAt(sz - 1) == 0);
}

public static String uBytesToHex(String string) {
    byte[] a = string.getBytes();
    StringBuilder sb = new StringBuilder();
    sb.append("0x");
    for (byte b : a) {
        sb.append(String.format("%02x", b & 0xff));
    }
    return sb.toString();
}

public static long uBytesToLong(String str, String byteOrder) {
    if (byteOrder.toUpperCase().equals("LE")) {
        str = new StringBuffer(str).reverse().toString();
    }
    byte[] b = str.getBytes();
    return (long) (uByte(b[0]) << 24 |
              uByte(b[1]) << 16 |
              uByte(b[2]) << 8 |
              uByte(b[3]));
}
public static long uBytesToDWord(String str, String byteOrder) {
    if (byteOrder.toUpperCase().equals("LE")) {
        str = new StringBuffer(str).reverse().toString();
    }
    byte[] b = str.getBytes();
    return (long) (uByte(b[0]) << 24 | uByte(b[1]) << 16 | uByte(b[2]) << 8 | uByte(b[3]));
}

public static int uBytesToUWord(String str, String byteOrder) {
    if (byteOrder.toUpperCase().equals("LE")) {
        str = new StringBuffer(str).reverse().toString();
    }
    byte[] b = str.getBytes();
    return (int) (uByte(b[0]) << 8 | uByte(b[1]));
}

public static int uBytesToWord(String str, String byteOrder) {
    if (byteOrder.toUpperCase().equals("LE")) {
        str = new StringBuffer(str).reverse().toString();
    }
    byte[] b = str.getBytes();
    return (short) (uByte(b[0]) << 8 | uByte(b[1]));
}

public static int uByte(byte b) {
    return ((int) b & 0x00ff);
}

public static int uByte(String str) {
    byte[] b = str.getBytes();
    return ((int) b[0] & 0x00ff);
}

Figure 12. Data Translation and Other Grammar Utilities

ANTLR was designed to create parsers for textual languages. It has one data type. This is the text data type that is returned by tokens. There are special grammar rules that use some of the utilities in Figure 12 to allow binary attribute grammars to read a sequence of bytes and return a data type other than text. These rules are shown in Figure 13. Together with the utilities in Figure 12, these grammar rules act as a work around that simulates a lexer that has bytes as input and returns data types other than text.
Each of the rules for a numeric data type take a parameter \textit{byteOrder}. This parameter is then passed on to the translation utilities. This allows the bytes of numeric data to be correctly translated. The \textit{byteOrder} is either big-endian or little-endian depending on how the data was originally stored in the file. For example, ilbm files are stored in big-endian order while wave files are stored in little-endian order.

As long as chunk-based binary array grammars are built on top of the simplified lexer grammar and the data type grammar, ANTLR 4 can be used to automatically create a lexer and parser that can be used with chunk-based binary array grammars.

```
grammar dataType;
// The binaryArray grammar is a simple lexer grammar with BYTE as the only
// Token type.
import binaryArray;
// The following rules are a work around.
// data types instead of Antlr4's text only data type.
// They allow other binary grammars read and return
// data types instead of Antlr4's text only Token type.
// These parser and lexer rules simulate a lexer that
// inputs bytes and returns data type Token on demand.
// This rule inputs byte array of a given size. It uses a
// long as the size of the array to input and returns
// a string of all printable characters of size or the
// hex representation of the array as a string that is
// length of size. dtData [long size] returns [String value]
  :  byteArray[$size]
    { if(Utilities.isPrintable($text))
        $value = $text;
      else
        $value = Utilities.uBytesToHex($text);
    }
  ;
dtString [long size] returns [String value]
  :  byteArray[$size]
    {$value = $text;}
  ;
dtZString [long size] returns [String value]
  :  byteArray[$size - 1]
```
{$value = $text;}

dtUByte
(Utilities.isValidZString($text))?
;

dtHexString [long size] returns [String value]
: byteArray[$size]
{$value = Utilities.uBytesToHex($text);}
;

dtLong [String byteOrder] returns [long value]
: byteArray[4]
{$value = Utilities.uBytesToLong($text, $byteOrder);}
;

dtDWord [String byteOrder] returns [long value]
: byteArray[4]
{$value = Utilities.uBytesToDWord($text, $byteOrder);}
;

dtUWord [String byteOrder] returns [int value]
: byteArray[2]
{$value = Utilities.uBytesToUWord($text, $byteOrder);}
;

dtWord [String byteOrder] returns [int value]
: byteArray[2]
{$value = Utilities.uBytesToWord($text, $byteOrder);}
;

dtUbyte returns [int value]
: byteArray[1]
{$value = Utilities.uByte($text);}
;

byteArray [long size]
locals [long index = 0]
: ({$index < $size)?
   BYTE
   ($index += 1;})+
;

Figure 13. Datatype rules used by all binary array attribute grammars
The single lexer rule and all the data type rules are a data type grammar. This makes it possible to simply import the data type grammar into the specific grammar for each chunk-based file format. A grammar for a file format needs rules that define the layout (structure) of the file format plus the data type (lexical) rules to handle the data type definition and the translation of numeric data into a usable form. A rewritten version of the ilbm grammar is shown in Figure 14. It imports the data type rules.

```plaintext
grammar ilbm;
import dataType;
// Nonterminal rules used to create the Parser
ilbm
locals [String byteOrder = "BE", boolean foundBMHD]
  :  formID = ckID
      {$formID.text.equals("FORM")}? // check that first ckID == FORM
    ckSize
    ilbmID = ckID
      {$ilbmID.text.equals("ILBM")}? // check that second ckID == ILBM
    propertiesLoop
    body
  ;
propertiesLoop
locals [String ckName]
  :  chunkName1 = ckID
      {$ckName = $chunkName1.text;}
      (?!$ckName.equals("BODY")? // check for the end of the loop.
        property[$ckName]
        chunkName2 = ckID {$ckName = $chunkName2.text;}
      )* 
  ;
catch [FailedPredicateException fpe]
{
  // Do nothing; $ckName == BODY
  // This exception is used as an exit for the propertyLoop
  // It only occurs when the next $ckName is equal to "BODY".
  // The BODY is where the actual picture is stored.
}
```
property[String currentID]
    :   {$currentID.equals("BMHD")}? bmhd
      | {$currentID.equals("CMAP")}? cmap
      | {$currentID.equals("CAMG")}? camg
      | {$currentID.equals("CRNG")}? crng
      | additionalProperty[currentID]
	;

bmhd
    :   ckSize {$ckSize.value == 20}? 
        bitMapHeader
        {$ilbm::foundBMHD = true;}
	;

bitMapHeader
    :   width = dtUWord{$ilbm::byteOrder}
        height = dtUWord{$ilbm::byteOrder}
        xPosition = dtWord{$ilbm::byteOrder}
        yPosition = dtWord{$ilbm::byteOrder}
        nPanes = dtUByte
        masking = dtUByte
        compression = dtUByte
        reserved1 = dtUByte
        transparentColor = dtUWord{$ilbm::byteOrder}
        xAspect = dtUByte
        yAspect = dtUByte
        pageWidth = dtWord{$ilbm::byteOrder}
        pageHeight = dtWord{$ilbm::byteOrder}
	;

cmap
locals [int count, int index]
@init {
    $count = 1;
    $index = 0;
}
    :   ckSize {($ckSize.value % 3) == 0}? 
        ({{$count <= $ckSize.value}? colorRegister[$index++] {$count += 3;}}+ 
	;
Figure 14. The ILBM File Format Grammar
A wave_form grammar is shown in Figure 15. It also imports the datatype rules.

```
grammar wave_form; // Define a grammar called wave_form
import dataType;
// Nonterminal rules used to create the Parser
wave_form
locals [String byteOrder = "LE", boolean foundFmtCk]
    wave = dtString[4] {wave.value.equals("WAVE")}?
    chunkOrListLoop
      ;
chunkOrListLoop
  : (ckID = dtString[4]
    chunkOrList[ckID.value])+
    ;
chunkOrList[String ckID]
  : {$ckID.equals("fmt ")}? fmt_ck {wave_form::foundFmtCk = true;}
    | {$ckID.equals("fact")}? fact_ck
    | {$ckID.equals("data") && wave_form::foundFmtCk}? data_ck
    | {$ckID.equals("cue ")}? cue_ck
    | {$ckID.equals("plist")}? playlist_ck
    | {$ckID.equals("LIST")}?
      (ckSize = dtDWord[wave_form::byteOrder]
        typeList = dtString[4]
        listType[ckSize.value, typeList.value])
    | unknownChunk
    ;
fmt_ck
  : ckSize = dtDWord[wave_form::byteOrder]
    common_fields
    format_specific_fields[common_fields.formatTag, ckSize.value - 14]
    ({Utilities.isOdd(ckSize.value)}? pad_byte)?
    ;
common_fields returns [int formatTag]
```
// Format category
wFormatTag = dtUWord[$wave_form::byteOrder]
{$formatTag = $wFormatTag.value;}
// Number of channels
wChannels = dtUWord[$wave_form::byteOrder]
// Sampling rate
dwSamplesPerSec = dtDWord[$wave_form::byteOrder]
// For buffer estimation
dwAvgBytesPerSec = dtDWord[$wave_form::byteOrder]
// Data block size
wBlockAlign = dtUWord[$wave_form::byteOrder]

/** <format-specific-fields> consist of zero or more bytes of parameters.
* Which parameters occur depends on the WAVE format category. Validation
* should allow for (and ignore) and unknown parameters. */
format_specific_fields [int formatTag, long sizeRest]
returns[int bitsPerSample]
:
  (($sizeRest > 0)?
    (($formatTag == 1)? (pcm_specific_fields[$sizeRest]
        {$bitsPerSample = $pcm_specific_fields.bitsPerSample;})
     | dtHexString[$sizeRest])
  |)

pcm_specific_fields [long sizeRest] returns[int bitsPerSample]
:
  wBitsPerSample = dtUWord[$wave_form::byteOrder] // Sample rate
  {$bitsPerSample = $wBitsPerSample.value;}
  (($sizeRest - 2) > 0)? dtHexString[$sizeRest - 2])?

fact_ck
:
  ckSize = dtDWord[$wave_form::byteOrder]
dwFileSize = dtDWord[$wave_form::byteOrder]

data_ck
:
  ckSize = dtDWord[$wave_form::byteOrder]
dtHexString[$ckSize.value]
  (($Utilities.isOdd($ckSize.value))? pad_byte)?
; 
cue_ck:
  dtDWord[$wave_form::byteOrder]
dwCuePoints = dtDWord[$wave_form::byteOrder]
cue_point_array[DWcuePoints.value]
;
cue_point_array [long size]
locals [long index]
  ({$index < $size}?
cue_point
  {$index += 1;})+
;
cue_point:
  dwName = dtDWord[$wave_form::byteOrder]
dwPosition = dtDWord[$wave_form::byteOrder]
fccChunk = dtString[4]
dwChunkStart = dtDWord[$wave_form::byteOrder]
dwBlockStart = dtDWord[$wave_form::byteOrder]
dwSampleOffset = dtDWord[$wave_form::byteOrder]
;
playlist_ck:
  ckSize = dtDWord[$wave_form::byteOrder]
dwSegments = dtDWord[$wave_form::byteOrder]
play_segment_array[DWsegments.value]
;
play_segment_array [long size]
locals [long index]
  ({$index < $size}?
play_segment
  {$index += 1;})+
;
play_segment:
  dwName = dtDWord[$wave_form::byteOrder]
dwLength = dtDWord[$wave_form::byteOrder]
dwLoops = dtDWord[$wave_form::byteOrder]
;
listType[long size, String typeList]
    :  {typeList.equals("adtl")}? assoc_data_list[size]
        | {typeList.equals("wavl") && $wave_form::foundFmtCk}? wave_list[size]
        | {typeList.equals("INFO")}? info_list[size]
        | unknownChunkLoop[size]
    ;

wave_list[long size]
locals[long index = 4]
    :  {index < size}?
        waveListType = dtString[4] dataOrSInt[waveListType.value]
        {index += 4 + dataOrSInt.text.length();})+
    ;
dataOrSInt[String dataType]
    :  {dataType.equals("data")}? data_ck
        | {dataType.equals("slnt")}? slnt_ck
    ;

slnt_ck
    :  ckSize = dtDWord[$wave_form::byteOrder]
        dtHexString[ckSize.value]
        {(Utilities.isOdd(ckSize.value))? pad_byte)?
    ;

assoc_data_list[long size]
locals[long index = 4]
    :  {index < size}?
        assoc_dataType = dtString[4] assoc_data_type[assoc_dataType.value]
        {index += 4 + assoc_data_type.text.length();})+
    ;
assoc_data_type[String assoc_dataType]
    :  {assoc_dataType.equals("labl")}? labl_ck
        | {assoc_dataType.equals("note")}? note_ck
        | {assoc_dataType.equals("ltxt")}? ltxt_ck
        | {assoc_dataType.equals("file")}? file_ck
    ;

labl_ck
    :  ckSize = dtDWord[$wave_form::byteOrder]
        dwName = dtDWord[$wave_form::byteOrder]
labal = dtZString[\$ckSize.value - 4]
((Utilities.isOdd($ckSize.value))? pad_byte)?
;

note_ck
: ckSize = dtDWord[$wave_form::byteOrder]
dwName = dtDWord[$wave_form::byteOrder]
comment = dtZString[\$ckSize.value - 4]
((Utilities.isOdd($ckSize.value))? pad_byte)?
;

ltxt_ck
: ckSize = dtDWord[$wave_form::byteOrder]
dwName = dtDWord[$wave_form::byteOrder]
dwSampleLength = dtDWord[$wave_form::byteOrder]
dwPurpose = dtString[4]
wCountry = dtUWord[$wave_form::byteOrder]
wLanguage = dtUWord[$wave_form::byteOrder]
wDialect = dtUWord[$wave_form::byteOrder]
wCodePage = dtUWord[$wave_form::byteOrder]
($ckSize.value - 20 > 0)? dtData[$ckSize.value - 20])?
((Utilities.isOdd($ckSize.value))? pad_byte)?
;

dwMedType = dtDWord[$wave_form::byteOrder]
($ckSize.value - 8 > 0)? dtData[$ckSize.value - 8])?
((Utilities.isOdd($ckSize.value))? pad_byte)?
;

info_list [long size]
locals [long index = 4]
: ($index < $size)?
    typeInfo = dtString[4]
    info_type[typeInfo.value]
    ($index += $info_type.text.length() + 4;)+
;

info_type[String typeInfo]
: {$typeInfo.equals("ISFT")}? isft
| {$typeInfo.equals("IENG")}? ieng
| {$typeInfo.equals("ICRD")}? icrd
| unknownChunk

; isft
: strSize = dtDWord[$wave_form::byteOrder]
software = dtZString[strSize.value]
((Utilities.isOdd(strSize.value))? pad_byte)?

; ieng
: strSize = dtDWord[$wave_form::byteOrder]
engineer = dtZString[strSize.value]
((Utilities.isOdd(strSize.value))? pad_byte)?

; icrd
: strSize = dtDWord[$wave_form::byteOrder]
creationDate = dtZString[strSize.value]
((Utilities.isOdd(strSize.value))? pad_byte)?

; unknownChunk
: dtDWord[$wave_form::byteOrder]
dtData[dwDWord.value]
((Utilities.isOdd(dwDWord.value))? pad_byte)?

; pad_byte
: dtUByte

; unknownChunkLoop [long size]
locals [long index = 4]
: (($index < $size)?
    dtString[4]
    unknownChunk
    ($index += $unknownChunk.text.length() + 4;))+

Figure 15. The Wave_Form File Format Grammar
6.4 Generating Parse Trees for Binary Array Grammars

There are two ways to generate parse trees using ANTLR. One way is to create a tree grammar. The other way is to embed print statements as actions in the grammar itself. Both methods are awkward and make it difficult to both read and specify grammars. With the advent of ANTLR4 [Parr 2012], this has changed. ANTLR4 has a built in parse tree walker. It automatically creates nodes for each rule in the grammar and a terminal node for each token.

6.4.1 How ANTLR4 Generates Parse Trees for String Attribute Grammars

ANTLR4 provides a command line parameter that allows the user to specify whether they would like the parse tree displayed as a nested list style tree or as a graphical parse tree in a graphical window. Figure 16 shows a parse tree displayed as a nested list.

\[
\text{prog (stat (expr 193) \ \|\ \n) (stat a = (expr 5) \ \|\ \n) (stat b = (expr 6) \ \|\ \n) (stat (expr (expr a) + (expr (expr b) * (expr 2))) \ \|\ \n) (stat (expr (expr ( (expr (expr 1) + (expr 2)) )) * (expr 3)) \ \|\ \n)}
\]

*Figure 16. Parse tree shown as a nested list.*

Figure 17 shows the parse tree output in a graphical user interface window.

*Figure 17. Parse tree shown as a graph.*
6.4.2 Using ANTLR4 to Generate Parse Trees for Binary Attribute Grammars

There are two problems with using the ANTLR4 function for displaying the parse tree as a nested list. Since the parse tree is displayed as one continuous string, the parse tree becomes almost unreadable when the input file is longer than a couple of lines. If one uses the ANTLR4 parse tree function on an attribute grammar that has been extended to enable binary data types, the output becomes even more unreadable. In string grammars, a token is a printable string. In a binary array grammar, the byte is not always a printable character. When a printable character for a byte is unavailable, a space is used by ANTLR4. Also, in the case of binary grammars, data types are represented as byte arrays of different lengths. Even a string is read as an array of byte tokens where each token value is separated by a space. The data type long integer is an array of 4 bytes. The decimal number that the bytes of data represent is what is important, not the individual bytes.

Figure 18 shows a display of the ILBM formatted file Docking.bru.

![Figure 18. ILBM File Docking.bru](image)

Figure 19 shows a nested list style parse tree created from a parse of the file using the ILBM ANTLR grammar shown in Figure 14. This single string requires 3 pages to printout and is almost unreadable. Most of the terminal bytes are represented as spaces or odd characters. The actual size of the image is shown to be 6594 bytes. However, to conserve space in this paper, in Figure 19 only the first three bytes and the last three bytes of the image are shown separated by an ellipsis.
Figure 19. Parse Tree Created by ANTLR4

The parse tree of binary array grammars can be made more readable by (1) pretty printing the nested lists so that the levels of the tree can be shown, and (2) by outputting the values returned by the datatype functions.

6.4.3 Extensions to ANTLR4 to Generate Pretty Printed Parse Trees with Data Type Values

ANTLR4 creates the nested list style parse tree by using a built-in parse tree walker. When ANTLR4 generates a parser, it creates a RuleContext for every rule. EachRuleContext has a RuleNode interface. The RuleNode contains a list of children, and an index into both a list of Rules and a list of RuleNames. A RuleNode child can be either a RuleNode or a TerminalNode. The TerminalNode is associated with text that matches its TokenType. When an ANTLR4 created parser calls its top rule, it returns the rule’s RuleContext. To get the parse tree string the user calls the toStringTree method of the returned context. The toStringTree method creates a string with its ruleName as the head of the list followed by a space then appends resulting strings from its recursive calls to the toStringTree method of all its children. The recursive calling stops when it reaches a TerminalNode, which simply returns a string containing its matching text.

A group of utilities, called prettyPrintTree, was created to pretty print a tree. This overloaded function behaves in a manner similar to the toStringTree method. This method takes a TreeNode parameter and walks the RuleNodes until it finds a RuleNode with “dt” as the start of its ruleName. As shown in Figure 14, every data type rule starts with “dt”. All the data type rules call binaryArray to build the data type from the stored data. Every data type rule has a return
value called “value”. When the data type rule is reached it returns the string value of its “value” attribute. Another difference between the toStringTree method and the prettyPrintTree method is that while ruleName is followed by a space in toStringTree, it is followed by a newline character and is indented according to the depth of its embedding in prettyPrintTree. The prettyPrintTree utility methods are shown in Appendix C.

Though the method is called prettyPrintTree, it does not actually print the parse tree. It formats the parse tree for printing or displaying in a window.

Figure 20 shows the parse tree that is result of calling the prettyPrintTree utility method with the top-level RuleContext returned from parsing the DOCKING.BRU ilbm file. Only a few of the 6594 bytes in the image are shown in the figure. The pretty printed parse tree is easier to read and understand than the nested lists shown in Figure 19.

```plaintext
(ilbm
  (ckID
    (dtString FORM))
  (ckSize
    (dtLong 6724))
  (ckID
    (dtString ILBM))
  (propertiesLoop
    (ckID
      (dtString BMHD))
    (property
      (bmhd
        (ckSize
          (dtLong 20)))
      (bitmapHeader
        (dtWord 706)
        (dtWord 328)
        (dtWord 0)
        (dtWord 0)
        (dtUByte 2)
        (dtUByte 0)
        (dtUByte 1)
        (dtUByte 194)
        (dtWord 0)
        (dtUByte 20)
        (dtUByte 11)
        (dtWord 706)
        (dtWord 450))))
  (ckID
    (dtString ANNO))
  (property
    (additionalProperty
```
Ver: Written by ASDG's Art Department Professional IFF3.0.4 (05.))

cid

data

property
cmap

cid

data

property
camg

cid

data

property
additionalproperty
cid

data

body

cid

data

hexstring
6.4.4 How ANTLR4’s Attribute Grammars, Semantic Predicates and Action Clauses Help with Parsing and Validation

Along with the ability to pass attribute values to and from rules, ANTLR4 allows rules to create local variables that can be seen and modified by rules below them in the parse tree. It is best to pass variables to and from rules using its parameters. But, this is not always efficient. Many file formats use a single byte order for the entire file. It is inconvenient to pass the byte order from every rule down to the data type level. By setting a local variable at the top level of a grammar, all rules in the grammar can access the variable by prefacing the variable name with the top-level rule’s name followed by two colons. Any rule can access one of its ancestor’s local variables in this way. As seen in Figures 14 and 15, the ilbm grammar and wave_form grammar each have a top-level variable defined as a string called byteOrder. In the ilbm grammar byteOrder is set to “BE” for big endian and in the wave_form grammar the byteOrder is set to “LE” for little endian.

Semantic predicates can be used in two ways. They can be used in guiding the parser or in validating the file format. The property rule below is an example of semantic predicates being used to guide the parse.

```
property[String currentID]
     :  {$currentID.equals("BMHD")}? bmhd
     | {$currentID.equals("CMAP")}? cmap
     | {$currentID.equals("CAMG")}? camg
     | {$currentID.equals("CRNG")}? crng
     | additionalProperty[currentID]    ;
```

The value of the semantic predicate `currentID` is not available until it is read from the file. Without this value, there no way to determine which rule should be called next. When the `currentID` is equal to the string “BMHD”, there are only two possible following rules—the bmhd rule or the additionalProperty rule. Because the bmhd path occurs first, it is chosen. If all the `currentID` tests fail, the additionalProperty rule would be the only possible rule. In that case, the value of chunk size is used. If the chunk size evaluates to odd, there is a pad byte at the end of the chunk.

Another use of semantic predicates is in validation of file formats. The top-level rule of the wave_form grammar shown below uses semantic predicates to support validation. Semantic
predicates used in this way are similar to assert statements in the C programming language. If the predicates evaluate to false, a failed predicate exception is thrown by the parser and everything following the predicate is ignored. The local foundFmtCk variable at the top-level is set to true when the format chunk rule is successful. Later, the foundFmtCk will be used as a semantic predicate to validate that the format chunk comes before the data chunk or before the LIST chunk with the list type “wavl”.

```java
wave_form
locals [String byteOrder = "LE", boolean foundFmtCk]
    wave = dtString[4] {wave.value.equals("WAVE")}? chunkOrListLoop;
```

ANTLR4’s action clauses can also be used to guide the parser or to aid in validating the file format. In action clauses, mathematical computations can be performed and variables can be assigned values. An if statement inside an action clause can be used to throw an exception or to send a message to the standard output or to the error output. The bmhd rule below shows the top-level variable ilbm foundBMHD being set to true after successfully parsing the bitmap header chunk. The body rule shows the foundBMHD value being tested for the purpose of validating the file format.

```java
bmhd
    : ckSize {$ckSize.value == 20}? bitMapHeader
        {$ilbm::foundBMHD = true;} ;
body
    : {$ilbm::foundBMHD}?
        ckSize dtHexString[$ckSize.value] ;
```

Shown below is the unknownChunkLoop rule that is found in the waveform grammar. It is called when a list is encountered that has an unknown list type. There is no way to know how many chunks it contains or what the chunks are. The action clause increments the index so the loop can be terminated when the size of the accumulated chunks equal the size of the list chunk.

```java
unknownChunkLoop [long size]
locals [long index = 4]
    : ({$index < $size)? dtString[4]
        unknownChunk
        {$index += $unknownChunk.text.length() + 4;})+ ;
```
7 Related Research

Researchers have developed a number of data description languages to support validation of file formats. EAST (Enhanced ADA Subset) is a data description language developed by the Consultative Committee for Space Data Systems [2010]. The Data Entity Dictionary Specification Language (DEDSL) can be used in conjunction with EAST for defining semantic information. The EAST description is used to interpret and provide access to information in binary and text files.

ASN.1 (Abstract Syntax Notation One) is an international standard which aims at specifying the structure of telecommunication and computer networking protocols between heterogeneous systems [Larmouth, 1999; Dubuission 2000; Walkin 2006].

DATASCRIPT [Back 2002] supports specifying and parsing binary data and has been used to manipulate Java jar files and ELF object files. The PADS/C data description language can be used to define the file formats of text and binary files [Fisher & Gruber 2005]. The PADS/C compiler compiles the description into tools that can be used to recognize, manipulate and transform the data into other formats.

The Data Format Description Language (DFDL) is being developed by a Working Group of the Open Grid Forum [OGF 2011]. The data description language is based on a subset of W3C XML Schema using <xs:appinfo> annotations to carry the extra information necessary to describe non-XML physical representations. Version 1 of the language specification was published in 2011 and a parser is being implemented.

Each of the data description languages described above can be used to define data types and file structures of binary files. However, they are based on file structure declarations such as those of C or Java. The binary file format grammar described in this paper most closely resembles DFDL. However, the binary file grammar described in this paper is the only data description language based on formal grammars that is used for creating parsers for file formats.

JHOVE (JSTOR/Harvard Object Validation Environment) is an API written in Java for the validation of the formats of digital files [Harvard University Library 2011]. JHOVE supports validation of the following formats: AIFF, GIF, HTML, JPEG, JPEG 2000, PDF, TIFF, WAVE, XML and ASCII and UTF-8 encoded text. JHOVE2 is a rewrite of JHOVE. It currently contains validation modules for the following formats: ICC Color Profile, SGML, ESRI Shapefile, TIFF, WAVE, XML and UTF-8 encoded text [California Digital Library 2011].

The research reported in this paper is similar in intent to that of the JHOVE projects—validation of binary file formats. As a matter of fact, binary array grammars and parsers are being
constructed for the chunk-based file formats AIFF, JPEG, JPEG 2000, WAVE and ESRI Shapefile. However, the research reported herein differs from that of the JHOVE projects in that what is sought is a technology for generating validators for binary file formats from grammars specifying the binary file format.

8. Conclusion

The research question addressed by his research is whether it is possible to extend the context-free grammars used to specify the syntax of programming languages to the specification of binary file formats and to use these grammars with parsers for validating the file formats of binary files. A major family of binary file formats, chunk-based, was described. Then extensions to the concepts of context-free grammars and attribute grammars were described that enable the specification of binary file formats. Examples of attribute array grammars for two chunk-based binary file formats were then presented. Recursive descent parsers for this class of grammars was then described. Experience in using ANTLR, a parser generator for LL(k) string grammars, in generating parsers for recognizing the formats of binary files was discussed. Finally, related research in data description languages for binary file formats was described and the JHOVE2 project which is creating Java-based tools for validating file formats was described.

It is concluded that it is possible to extend context-free grammars to the specification of chunk-based binary file formats. Furthermore, these grammars can be used with recursive descent parsers for parsing the file formats of chunk-based binary files. It remains to be determined whether these attribute grammars based on binary array grammars are adequate to define other families of binary file formats.

A capability to validate binary file formats assumes that the file format of a file is already known. In this research, we use a file type identifier based on the UNIX file command and a magic file that we have created [Underwood 2009]. We have samples of the chunk-based file formats discussed in this report including those referenced in Appendices A and B. We also have file signature tests (magic tests) for all these file types.

We would like to have a parser generator (also called a compiler–compiler) that would take as input an attribute grammar for a binary file format and generate a parser or interpreter that recognizes the file format or that interprets the file format and displays or plays the contents. The closest parser generator that we have found to having these capabilities is ANTLR and ANTLR4 (www.antlr.org/). ANTLR will generate a top-down recursive descent parser from a context-free attribute grammar. However, it works with strings, and generates a separate lexical analyzer.
Appendices A and B show the names of 86 file formats that belong to the family of “chunk-based” file formats. We have specifications for most of these file formats and samples of files with these formats.

These file formats have a similar structure. We have developed binary array grammars for two of these binary file formats. We should be able to do so for all of them. We used these grammars with the extensions to ANTLR4 to generate top-down recursive descent parsers for the file formats defined with these grammars.

In this paper it is shown how to extend ANTLR4 to generate parsers for binary array attribute grammars by creating lexical rules and functions for binary data types. It is also shown how to generate parse trees represented as nested lists that can be pretty printed.

In this paper, it has not yet been demonstrated how to create a parser that validates binary file formats with error messages for noncompliant syntax or semantics. Validation of binary file formats via binary array attribute grammars will be demonstrated in a forthcoming report on binary array attribute grammars for directory-based binary file formats.

The significance of success in this research task is that if binary file formats can be specified with binary array attribute grammars, then only one parsing generator is needed to generate the parsers for verifying that a file conforms to a particular format. Similarly, the same parsing generator can possibly be used for conversion of legacy file formats to current or standard formats. Finally, the same parser generator can possibly be used for generating viewers/players for most file formats. This would increase the likelihood of preserving and making available into the indefinite future those digital records encoded in binary file formats.
References


[ETSI 2012] ETSI TS 26.244; Transparent end-to-end packet switched streaming service (PSS); 3GPP file format (3GP) 2012 www.3gpp.org/ftp/Specs/html-info/26244.htm


[Glanville 2003] R. S. Glanville. Format of Anim8or .an8 Files - V0.85, 21-Sep-03. [http://www.anim8or.com/resources/an8_format.txt](http://www.anim8or.com/resources/an8_format.txt)


http://147.91.177.212 extra/fileformat/graphics/pic/picpro.txt

http://svaf.sourceforge.net/svaf.html

http://netghost.narod.ru/gff/vendspec/micriff/ms_riff.txt


[Microsoft 2003b] Microsoft Corporation. Multiple Channel Audio Data and WAVE Files


[ MultimediaWiki 2006] Smush Animation Format, August 2006
[MultimeduiaWiki 2008a] Electronic Arts CMV, April 2008  


[MultimediaWiki 2008g] Electronic Arts TGQ, November 2008  


http://sk1project.org/modules.php?name=Products&product=cdrexplorer


www.ibiblio.org/emusic-l/info-docs-FAQs/MIDI-doc/MIDI-SMF.txt

Software: Practice and Experience. 25, 7 (July 1995), 789-810.  


[Peters] Christoph Peters. The Ultimate 3D model file format (v. 2.0)  
http://ultimate3d.org/U3DModelFileFormatV2.htm

http://xiph.org/ogg/doc/rfc3533.txt

www.eblong.com/zarf/blorb/  
www.ifarchive.org/if-archive/infocom/interpreters/specification/blorb_format.txt

www.randelshofer.ch/multishow/javadoc/ch/randelshofer/media/riff/RIFFParser.html

www.libpng.org/pub/mng/spec/#Credits

http://sc.openoffice.org/excelfileformat.pdf

[REWiki 2009] reWiki. .IFF. 2009  
http://rewiki.regengedanken.de/wiki/.IFF

www.ietf.org/rfc/rfc3072.txt

http://fortunaty.net/com/textfiles/computers/ftxt

http://www.wotsit.org/list.asp?search=GTK&button=GO%21

http://amigan.1emu.net/reg/ANIM.txt


Appendix A: Binary Array Grammars for Chunk-based Binary File Formats

This appendix sketches binary array grammars for some of binary chunk-based file formats.

A.1 Electronic Arts Interchange File Format (IFF)

A.1.1 Interleaved Bitmap (ILBM)
A binary array attribute grammar for this file format was presented in section 4.1 of this report [Morrison 1986].

A.1.2 8-bit Sampled Voice (8SVX)
The file format specification for the 9-bit sampled voice file format uses C programming language data structures to define the format [Hayes and Morrison 1985]. The following is a re-expression of that specification as a context-free binary array grammar. It does not yet include synthesized and inherited attributes.

```xml
<8SVX> → “FORM” <cksize INT32> “8SVX” <ckdata>
<ckdata> → <voice8header> <bodyck>
<ckdata> → <voice8header> <nameck> <copyrightck> < bodyck>
<ckdata> → <voice8header> <copyrightck> <nameck> < bodyck>
<ckdata> → <voice8header> <nameck> < bodyck>
<ckdata> → <voice8header> <copyrightck> < bodyck>
<ckdata> → <voice8header> <authorck> < bodyck>
<ckdata> → < voice8header> <annotation> < bodyck>
<ckdata> → < voice8header> <atakck> < releaseck> < bodyck>
<nameck> → “NAME” <cksize INT32 kk> CHAR[kk]
<copyrightck> → ‘(c) ’ <cksize INT32 ll> CHAR[ll]
<authorck> → “AUTH” <cksize INT32 mm> CHAR[mm]
<annotation> → <annotationck> < annotation>
<annotation> → < nnotationck>
<annotationck> → “ANNO” <cksize INT32 i> CHAR[i]
<atakck> → “ATAK” <cksize INT32 nn> <EGpoint>[nn]
<releaseck> → “RLSE” <cksize INT32 m> <EGpoint>[m]
<EGpoint> → <duration UWORD> < dest FIXED>
<bodyck> → “BODY” <cksize INT32 n> samples[n]
```
A.2 Standard Midi File Format

Oppenheim [1988] described the midi file format using the pseudo EBNF notation shown below [Int. MIDI Assoc 2008].

```
<Midi> → <headerchunk> <trackchunks>
<headerchunk> → "MThd" <length of header data> <header data>
<header data> → <format> <ntrks> <division>
<trackchunks> → <trackchunk> <trackchunks>
<trackchunk> → "MTrk" <length of track data> <track data>
<track data> → <MTrk event>+ 
<MTrk event> → <delta-time> <event>
<event> → <MIDI event> | <sysex event> | <meta-event>
<MIDI event> → <MIDI channel message>
<sysex event> → 0xF0 <length> <bytes to be transmitted after F0>
<sysex event> → 0xF7 <length> <all bytes to be transmitted>
<meta-event> → 0xFF <type> <length> <bytes>
```

The “length of header data” is a 32-bit big-endian integer with value 6. Header data consists of three 16-bit integers. The pseudo EBNF can be more precisely specified with the following context-free binary array grammar. This grammar does not yet include synthesized and inherited attributes

```
<Midi> → <headerchunk> <trackchunks>
<headerchunk> → "MThd" <headerdatalength BEINT32 6> <headerdata>
<headerdata> → <format BEINT16> <ntrks BEINT16> <division BEINT16>
<trackchunks> → <trackchunk> <trackchunks>
<trackchunk> → "MTrk" <trackdatalength BEINT32> <trackdata>
<trackdata> → <MTrkevent>+ 
<MTrkevent> → <delta-time VARLENQUAN> <event>
<event> → <MIDI event> | <sysex event> | <meta-event>
<MIDI event> → <MIDI channel message>
<sysex event> → 0xF0 <length VARLENQUAN k> <bytes to be transmitted after F0>[k]
<sysex event> → 0xF7 <length VARLENQUAN m> <all bytes to be transmitted>[m]
<meta-event> → 0xFF <type BYTE> <length VARENQUAN n> <bytes>[n]
```

The datatype VARLENQUAN is a Variable –Length Quantity. “These numbers are represented 7 bits per byte, most significant bits first. All bytes except the last have bit 7 set, and the last byte has bit 7 clear.”

A.3 Audio Interchange File Format

The Audio Interchange File Format (AIFF) developed by Apple Computer in 1988 was based on Electronic Arts’ Interchange File Format. The file format is specified using C data structure
definitions [Apple 1989]. The following grammar recasts the specification as a context-free pseudo EBNF grammar. It does not yet include the data types per a binary array grammar or synthesized and inherited attributes.

```xml
<AIFF> --> "FORM"<cksize INT32>"AIFF"<localchunks>

<localhunks> --> <CommonChunk><SoundDataChunk>

<CommonChunk> --> COMM"<cksize INT32 18><numChannels INT16>
  <numSampleFrame UINT32><sampleSize INT16>
  <sampleRate FLOAT80>

<SoundDataChunk> --> "SSND"<cksize INT32 k><offset UINT32>
  <blockSize UINT32><soundData UBYTE>[k-8]

<MarkerChunk> --> "MARK"<cksize INT32 mm><Markers>[mm]

<Markers> --> <Marker><Markers>

<Marker> --> <markerid UINT16><position UINT32><markerName PSTRING>

<InstrumentChunk> --> "INST"<cksize INT32>
  <baseNote CHAR>
  <detune CHAR>
  <lowNote CHAR>
  <highNote CHAR>
  <lowVelocity CHAR>
  <highVelocity CHAR>
  <gain INT16>
  <sustainLoop> <Loop>
  <releaseLoop> <Loop>

<Loop> --> <playmode UINT16><beginloop markerid><endloop markerid>

<playmode UINT16> --> 0 | 1 | 2

<MIDIDataChunk> --> "MIDI"<cksize INT32 nn><MIDIData>[nn]

<AudioRecordingChunk> --> "AESD"<cksize INT32><AESChannelStatusData>[24]

<ApplicationSpecificChunk> --> "APPL"<cksize INT32 kk>
  <applicationSignature CHAR[4]>
  <data pstring>[kk-4]

<CommentsChunk> --> "COMT"<cksize INT32 i>
  <numcomment UINT16><comments>[i-4]

<Comment> --><timestamp UINT32>
  <marker markerid>
  <count UINT16 j>
  <text CHAR>[1]
```
**A.4 Resource Interchange File Formats**

The Resource Interchange File Format (RIFF) introduced by IBM and Microsoft [1991] is also based on Electronic Arts’ Interchange File Format. The Microsoft container formats like Audio-Video Interleave (AVI) and Waveform PCM (WAV) use RIFF as their basis [Microsoft 1992] [Randelshofer 2011]. WebP, a picture format introduced in (2010) by Google, also uses RIFF as a container.

**A.4.1 RIFF-Waveform (WAVE)**

This file format was discussed in section 6.3 of this report [IBM & Microsoft 1991].

**A.4.2 WAVEFORMATEX**

The WAVEFORMATEX type must contain both a fact chunk and an extended wave format description within the ‘fmt’ chunk [Microsoft 1994]. The fact chunk specifies the length of the file in samples. Wave_Format-PCM has value 0x0001 for wFormat Tag. WAVEFORMATEX has different values for wFormat Tag. It also adds cbsize, the count in bytes of the extra size to the common fields.

```plaintext
<common-fields> -> <wFormatTag UINT16>
    <wChannels UINT16>
    <nSamplesPerSec UINT32>
    <nAvgBytesPerSec UINT32>
    <nBlockAlign UINT16>
    <wBitsPerSample UINT16>
    <cbsize UINT16 m>
    <extradata>[m]
```

**A.5 JPEG**


The JPEG standard specifies the codec, which defines how an image is compressed into a stream of bytes and decompressed back into an image, but not the file format used to contain that stream. The Exif and JFIF standards define the commonly used file formats for interchange of JPEG-compressed images.

**A.5.1 JPEG File Interchange Format (JFIF)**

A JPEG image consists of a sequence of segments, each beginning with a marker, each of which begins with a 0xFF byte followed by a byte indicating what kind of marker it is. Some markers consist of just those two bytes; others are followed by two bytes indicating the length of marker-
data that follows. The length includes the two bytes for the length, but not the two bytes for the marker. [Hamilton 1992, Cuturicu and Fromme 1999, Ecma 2009]

APP0 is the Application Segment for JFIF.

\[
\text{<JFIF> → <SOI><APP0><segments> <EOI>}
\]

\[
\text{<APP0> → 0xFFE0 <segsize UINT16> “JFIF” <NUL><majversion ubyte><minversion ubyte><densityunit ubyte><Xdensity UINT16><Ydensity UINT16><thumbnailwidth ubyte tw><thumbnailheight ubyte th><thumbnail data 24-bits> [tw x th]}
\]

\[
\text{<segments> → <segment> <segments>}
\]

\[
\text{<segment> → <marker><Segsize UINT16 n> <data>[n-2]}
\]

\[
\text{<marker> → 0xFF <type BYTE>}
\]

\[
\text{<SOI> → 0xFFD8}
\]

\[
\text{<EOI> → 0xFFD9}
\]

\[
\text{<NUL> → 0x00}
\]

\[
\text{A.5.2 JPEG/Exif}
\]

APP1 is the Application Segment for EXIF for JPEG files. [JEIDA 1998, JEITA 2002] Attribute Information for JPEG compressed files is stored in the Tag information format of TIFF 6.0.

\[
\text{<Exif> → <SOI><APP1>[APP2]> <DQT><DHT><SOF><SOS><Compressed_Data><EOI>}
\]

\[
\text{<APP1> → 0xFFE1 <segsize UINT16> “Exif” <NUL><NUL><TIFF_Header>}
\]

\[
\text{<TIFF_Header> → "II"|"MM")|0x2A<Exif_IFD-Pointer BEINT32 8><Exif_IFD>}
\]

\[
\text{<Exif_IFD> → <ExifVersion> | <ImageWidth> | <Image_Length> | <BitsPerSample> | <Compression>}
\]

\[
\text{<ExifVersion> → <Tag UINT16 0x9000><Type UINT16 0x07><Count UINT32 4>"0201"}
\]

\[
\text{<APP2> → 0xFFE2<segsize UINT16>"}FPXR"<NUL><Version BYTE 0x00><Contents>}
\]

\[
\text{<DQT> → 0xFFDB<segsize UINT16 197>}
\]

\[
\text{<Y byte 0x00><QTY BYTE>[64]}
\]

\[
\text{<Cb BYTE 0x01><QTCb BYTE[64]}
\]

\[
\text{<Cr BYTEv 0x02><QTCr BYTE>[64]}
\]

\[
\text{<DHT> → 0xFFC4<segsize UINT16 0x01A2>data[418]}
\]

\[
\text{<SOF> → 0xFFF0<segsize UINT16 17>}
\]

\[
\text{<Precision UBYTE x08>}
\]

\[
\text{<Vert_Lines><Horiz_Lines> <Components>}
\]

\[
\text{<Component_No> <H0_V0> <Quantization_Designation>}
\]

\[
\text{<Component_No> <H1_V1> <Quantization_Designation>}
\]

\[
\text{<Component_No> <H2_V2> <Quantization_Designation>}
\]
SPIFF is a JPEG file format that is intended to replace the de facto JFIF (JPEG File Interchange Format). SPIFF includes all of the features of JFIF and adds more functionality. The SPIFF file format is specified in annex F of ITU-T Recommendation T.84. [ITU 1998]

Microsoft’s Advanced Systems Format (ASF) version 1 [Kuznetkov 2001] is also a chunk-based container format. It was developed about 1995-96, but its specification was never published.

The format does not specify with which codec the video or audio should be encoded. It just specifies the structure of the video/audio stream. The most common file formats (codecs?) contained within an ASF file are Windows Media Audio (WMA) and Windows Media Video (WMV).

```
<ASF> → <HeaderObject><DataObject>
<HeaderObject>→<Header_Object_ID GUID 75B22630-668E-11CF-A6D9-00AA0062CE6C>
  <Size LEINT64>
  <NumHeaderObjects LEINT32>
  <Reserved1 BYTE 0x01>
  <Reserved2 BYTE 0x02>
  <FilePropertiesObject>
  <HeaderExtensionObject>
  <StreamPropertiesObject>

<FilePropertiesObject> → <GUID 8CABDCA1-A947-11CF-8EE4-00C00C205365>
  <Size LENT64>
  <FileID GUID>
  <FileSize LEINT64>
  <CreationDate LEINT64>
  <DataPacketsCount LEINT64 m>
  <PlayDuration LEINT64>
  <SendDuration LEINT64>
  <Preroll LEINT64>
  <Flags Bytes[4]>
  <MinDataPacketSize LEInt32>
  <MaxDataPacketSize LEINT32>
  <MaxBitRate LEINT32>

<StreamPropertiesObject> → <ID GUID B7DC0791-A9B7-11CF-8EE6-00C00C205365>
  <Object_Size LEUINT64>
  <Stream_Type GUID>
  <Error_Correction_Type GUID>
  <Time_Offset LEUINT64>
  <Type-Specific_Data_Length LEUint32 m>
  <Error_Correction_Data_Length LEUINT32 n>
  <Flags BYTE[2]>
```
A.6.1 Windows Media Audio 7-9

When the Stream Type of the Stream Properties Object has the value ASF_Audio_Media = F8699E40-5B4D-11CF-A8FD-00805F5C442B, the ASF audio media type that populates the Type-Specific Data field of the Stream Properties Object is represented using the following structure.

<Type-Specific_Data BYTE>[m] → <CodecID LEWORD>
   <nChannels WORD>
   <nSamplesPerSec LEINT32>
   <nAvgBytesPerSec LEINT32>
   <nBlockAlign WORD>
   <wBitsperSample LEINT16>
   <CbSize LEINT16 n>
   <Codec Specific Data BYTE>[n]

The <CodecID> specifies the unique ID (FormatTag) of the codec used to encode the audio data. The following values indicate codecs for Windows Media Audio 7-9.

<table>
<thead>
<tr>
<th>Codec name</th>
<th>Codec ID/</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
A.6.2 Windows Media Video 7-9.1

When the Stream Type of the Stream Properties Object has the value ASF_Video_Media = BC19EFC0-5B4D-11CF-A8FD-00805F5C442B, the ASF video media type that populates the Type-Specific Data field of the Stream Properties Object is represented using the following structure. [Microsoft 2004]

```
<Type-SpecificData> → <EncodedImageWidth LEINT32>
  <EncodedImageHeight LEINT32>
  <ReservedFlslgs INT16 2>
  <FormatDataSize LEUINT32 n>
  <FormatData>

<FormatData> → <FormatDataSize LEINT32 m>
  <ImageWidth LEINT32>
  <ImageHeight LEINT32>
  <Reserved LEINT16 1>
  <nBitsPerPixel LEINT16>
  <CompressionID CHAR[4]>
  <ImageSize LEINT32>
  <HorizontalPixelsPerMeter LEINT32>
  <VerticalPixelsPerMeter LEINT32>
  <nColorsUsed LEINT32>
  <nImportantColors LEINT32>
  <CodecSpecificData BYTE>
```

The following values of CompressionID indicate codecs for Windows Media Video 7-9.

<table>
<thead>
<tr>
<th>CompressionID</th>
<th>Compression Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>WMV1</td>
<td>Windows Media Video 7</td>
</tr>
<tr>
<td>WMV2</td>
<td>Windows Media Video 8</td>
</tr>
</tbody>
</table>
A.7 Portable Network Graphics
[ISO/IEC 15948:2003]

<PNG> -> <PNGSignature><headerck><paletteck><imagedata><trailer>
<PNGSignature> -> 0x89"PNG"0xODOA1A0A
<headerck> -> <cksize UINT32>“IHDR”
     <Width UINT32>
     <Height UINT32>
     <BitDepth UINT8>
     <ColourType UINT8>
     <Compressionmethod UINT8>
     <Filtermethod UINT8>
     <Interlacemethod UINT8>
     <CRC DWORD>
<paletteck> -> <cksize UINT32>“PLTE”<
     <entries><CRC DWORD>
<entries> -> <entry><entries>
<entry> -> <red BYTE>
     <green BYTE>
     <blue BYTE>
<imagedata> -> <cksize UINT32 n>"IDAT"<data BYTE>[n]
<trailer> -> cksize UINT32>“IEND”<CRC DWORD>

A.8 Apple QuickTime Movie

[Apple 2007]

<QTFF> -> <file type atom><movie atom>

<file type atom> -> <atomsize UINT32><type string[4]=”ftyp”>
     <major_brand string[4]=”qt “> <minor_version 0x20050300>
     <compatible_brands string[4]>[atomsize-16]

<movie atom> -> <atomsize UINT32><type string[4]=”moov”>
     <movie header atom>
     <track atom>

<movie header atom> -> <atom size UINT32><type string[4]=”mvhd”>
     <version BYTE>
     <flags BYTE[3]>
     <creation time UINT32>
<modification time UINT32>
<time scale UINT32>
<Duration UINT32>
<preferred rate UINT32>
<preferred volume UINT16>
<reserved BYTE[10]=0x00>
<matrix structure UINT32[9]>
<preview time UINT32>
<preview duration UINT32>
<poster time UINT32>
<selection time UINT32>
<selection duration UINT32>
<current time UINT32>
<next track id UINT32>

<track atom> → <atom size UINT32><type string[4]="trak">
    <track header atom>
    <media atom>

<track header atom> → <atom size UINT32> <type string[4]="tkhd">
    <version BYTE>
    <flags BYTE[3]>
    <creation time UINT32>
    <modification time UINT32>
    <track ID UINT32>
    <reserved UINT32=0>
    <duration UINT32>
    <reserved BYTE[8]=0x00>
    <layer UINT16>
    <alternate group UINT16>
    <volume UINT16>
    <reserved UINT16=0>
    <matrix structure INT32[9]>
    <track width UINT32>
    <track height INT32>

<media atom> → <atom size UINT32><type string[4]="mdia">
    <media header atom>

<media header atom> → <atom size UINT32><type string[4]="mdhd"
A.9 ESRI Shapefile

[ESRI 1998]

<shapefile>→ <filecode BE INT32=9994>
    <unused INT32>[5]
    <filelength BE INT32>
    <version LE INT32=1000>
    <shapetype LE INT32>
    <Xmin LE FP64>
    <Ymin LE FP64>
    <Xmax LE FP64>
    <Ymax LE FP64>
    <Zmin LE FP64>
    <Zmax LE FP64>
    <Mmin LE FP64>
    <Mmax LE FP64>
    <records>
<records> → <record><records> | <record>
<record> → <recordnumber BE INT32>
    <contentlength BE INT32>
    <recordcontent>
<recordcontent> → <NullShapeRecord> | <PointRecord> | <MultiPointRecord> | <PolyLineRecord> | <PolygonRecord> | <PointMRecord> | <MultiPointMRecord> | <PolyLineMRecord> | <PolygonMRecord> | <PointZRecord> |
<MultiPointZRecord> |  
<PolyLineZRecord> |  
<PolygonZRecord> |  
<MultiPatchRecord>  
<NullShapeRecord> → <shapetype LEINT32=0>  
<PointRecord> → <shapetype LEINT32=1><Point>  
<MultiPointRecord> → <shapetype LEINT32=8><MultiPoint>  
<PolyLineRecord> → <shapetype LEINT32=3><PolyLine>  
<PolygonRecord> → <shapetype LEINT32=5><Polygon>  
<PointMRecord> → <shapetype LEINT32=21><PointM>  
<MultiPointMRecord> → <shapetype LEINT32=28><MultiPointM>  
<PolyLineMRecord> → <shapetype LEINT32=23><PolyLineM>  
<PolygonMRecord> → <shapetype LEINT32=25><PolygonM>  
<PointZRecord> → <shapetype LEINT32=11><PointZ>  
<MultiPointZRecord> → <shapetype LEINT32=18><MultiPointZ>  
<PolyLineZRecord> → <shapetype LEINT32=13><PolyLineZ>  
<PolygonZRecord> → <shapetype LEINT32=15><PolygonZ>  
<MultiPatchRecord> → <shapetype LEINT32=31><MultiPatch>  
<Point> → <X LEFP64> <Y LEFP64>  
<MultiPoint> → <Box>  
  <NumPoints LEINT32>  
  <Points>[NumPoints]  
<Box> → <Xmin LEFP64><Ymin LEFP64> <Xmax LEFP64><Ymax LEFP64>  
<PolyLine> → <Box>  
  <NumParts LEINT32>  
  <NumPoints LEINT32>  
  <Parts LEINT32>[NumParts]  
  <Points>[NumPoints]  
<Polygon> → <Box>  
  <NumParts LEINT32>  
  <NumPoints LEINT32>  
  <Parts LEINT32>[NumParts]  
  <Points>[NumPoints]  
<PointM> → <X LEFP64><Y LEFR64><Measure LEFP64>  
<MultiPointM> → <Box>  
  <NumPoints LEINT32>  
  <Point>[NumPoints]  
  <Mmin LEFP64>  
  <Mmax LEFP64>  
  <Marray LEFP64>[BNumPoints]  
<PolyLineM> → <Box>  
  <NumParts LEINT32>
<NumPoints LEINT32>
<Parts LEINT32>[NumParts]
<Point>[NumPoints]
<Mmin LEFP64>
<Mmax LEFP64>
<Marray LEFP64>[NumPoints]

<Box>
<NumParts LEINT32>
<NumPoints LEINT32>
<Parts LEINT32>[NumParts]
<Point>[NumPoints]
<Mmin LEFP64>
<Mmax LEFP64>
<Marray LEFP64>[NumPoints]

<PointZ> → <Box>
<X LEFP64>
<Y LEFP64>
<Z LEFP64>
<M LEFP64>

<MultiPointZ> → <Box>
<NumPoints>
<Point>[NumPoints]
<Zmin LEFP64>
<Zmax LEFP64>
<Zarray LEFP64>[NumPoints]
<Mmin LEFP64>
<Mmax LEFP64[^$]>
<Marray LEFP64>[NumPoints]

<PolyLineZ> → <Box>
<NumParts LEINT32>
<NumPoints LEINT32>
<Parts LEINT32>[NumParts]
<Point>[NumPoints]
<Zmin LEFP64>
<Zmax LEFP64>
<Zarray LEFP64>[NumPoints]
<Mmin LEFP64>
<Mmax LEFP64>
<Marray LEFP64>[NumPoints]

<PolygonZ> → <Box>
<NumParts LEINT32>
<NumPoints LEINT32>
<Parts LEINT32>[NumParts]
<Point>[NumPoints]
<Zmin LEFP64>
<Zmax LEFP64>
<Zarray LEFP64>[NumPoints]
<Mmin LEFP64>
<Mmax LEFP64>
<Marray LEFP64>[NumPoints]

MultiPatch → Box

<NumParts LEINT32>
<NumPoints LEINT32>
<Parts LEINT32>[NumParts]
<PartTypes LEINT32>[NumParts]
<Point>[NumPoints]

<Zmin LEFP64>
<Zmax LEFP64>
<Zarray LEFP64>[NumPoints]
<Mmin LEFP64>
<Mmax LEFP64>
<Marray LEFP64>[NumPoints]
## Appendix B: Other Binary Chunk-based File Formats

<table>
<thead>
<tr>
<th>File Format Name</th>
<th>Reference</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>CEL Animation (ANIM)</td>
<td>Sparta 1988</td>
<td>Electronic Arts IFF</td>
</tr>
<tr>
<td>IFF Formatted Text (FTXT)</td>
<td>Shaw et al 1985</td>
<td>“</td>
</tr>
<tr>
<td>IFF Simple Musical Score (SMUS)</td>
<td>Morrison 1986b</td>
<td>“</td>
</tr>
<tr>
<td>Planar Bitmap (PBM)</td>
<td>REWiki 2009</td>
<td>“</td>
</tr>
<tr>
<td>Audio Interchange File Format - Compressed</td>
<td>Apple 1991</td>
<td></td>
</tr>
<tr>
<td>WAVEFORMATEXTENSIBLE</td>
<td>Microsoft 2003b</td>
<td>Resource Interchange File Formats (RIFF)</td>
</tr>
<tr>
<td>Broadcast Wave Format (BWF)</td>
<td>EBU 2001</td>
<td>“</td>
</tr>
<tr>
<td>Windows Audio-Video Interleave (AVI)</td>
<td>Microsoft 2005</td>
<td>“</td>
</tr>
<tr>
<td>Animated Windows Cursor (ANI)</td>
<td>Houghtaling 1996</td>
<td>“</td>
</tr>
<tr>
<td>RIFF MIDIfile (RMID)</td>
<td>IBM &amp; Microsoft 1991</td>
<td>“</td>
</tr>
<tr>
<td>DIVX Media Format (DMF)</td>
<td>Wikipedia 2012</td>
<td>“</td>
</tr>
<tr>
<td>Device Independent Bitmap (RIFF RDIB)</td>
<td>IBM &amp; Microsoft 1991</td>
<td>“</td>
</tr>
<tr>
<td>WebP</td>
<td>Google 2010</td>
<td>“</td>
</tr>
<tr>
<td>JPEG Titled Image Pyramid (JTIP)</td>
<td>ISO/IEC 10918-3:1997</td>
<td></td>
</tr>
<tr>
<td>JPEG-LS</td>
<td>ISO/IEC 14495-1:1999</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ISO/IEC 14495-2:2003</td>
<td></td>
</tr>
<tr>
<td>Multiple-Image Network Graphics</td>
<td>Randers-Pehrson 2001</td>
<td></td>
</tr>
<tr>
<td>JPEGn Network Graphics</td>
<td>Randers-Pehrson 2001</td>
<td></td>
</tr>
<tr>
<td>3D Studio File format – 3ds</td>
<td>Autodesk 1997</td>
<td></td>
</tr>
<tr>
<td>3D Studio Material Library File</td>
<td>Fercoq 1996</td>
<td></td>
</tr>
<tr>
<td>Anim8or</td>
<td>Glanville 2003</td>
<td></td>
</tr>
<tr>
<td>Animator Pro Cel</td>
<td>Crazy Talk</td>
<td></td>
</tr>
<tr>
<td>Format/Module</td>
<td>Author/Creator</td>
<td>Description</td>
</tr>
<tr>
<td>------------------------------------------------------------------------------</td>
<td>----------------------</td>
<td>--------------------------------------------------</td>
</tr>
<tr>
<td>Animator Pro Fli</td>
<td>Haaland</td>
<td></td>
</tr>
<tr>
<td>Animator FLC</td>
<td>Comuphase</td>
<td></td>
</tr>
<tr>
<td>Animator Pro PIC</td>
<td>Maischein</td>
<td></td>
</tr>
<tr>
<td>Audio Manager Module</td>
<td>Foo 1995</td>
<td></td>
</tr>
<tr>
<td>Autodesk Drawing Interchange Files (DXF) (Binary)</td>
<td>Autodesk 2009</td>
<td></td>
</tr>
<tr>
<td>Caligari Truespace Scene/Object (Binary)</td>
<td>Caligari 1998</td>
<td></td>
</tr>
<tr>
<td>CorelDraw Vector Graphics File (Versions 3.0-X3)</td>
<td>Novikov and Fillipov</td>
<td></td>
</tr>
<tr>
<td>DigiTrekker Module Format</td>
<td>Beham 1995</td>
<td>Music Modules</td>
</tr>
<tr>
<td>Graoumf Tracker Modules</td>
<td>Sora 1995</td>
<td>&quot;</td>
</tr>
<tr>
<td>Oktalyzer</td>
<td>Zappa 1994</td>
<td>&quot;</td>
</tr>
<tr>
<td>Electronic Arts Music (.asf)</td>
<td>Anisimovksy 2002b</td>
<td>EA Multimedia Game File Formats, Anisimovksy 2002a</td>
</tr>
<tr>
<td>Electronic ARTS CMV</td>
<td>Multimedia Wiki 2008a</td>
<td>&quot;</td>
</tr>
<tr>
<td>Electronic Arts DCT</td>
<td>Multimedia Wiki 2008b</td>
<td>&quot;</td>
</tr>
<tr>
<td>Electronic Arts MAD</td>
<td>Multimedia Wiki 2008c</td>
<td>&quot;</td>
</tr>
<tr>
<td>Electronic Arts MPC</td>
<td>Multimedia Wiki 2008d</td>
<td>&quot;</td>
</tr>
<tr>
<td>Electronic Arts VP6</td>
<td>Multimedia Wiki 2008e</td>
<td>&quot;</td>
</tr>
<tr>
<td>Electronic Arts TGV</td>
<td>Multimedia Wiki 2008f</td>
<td>&quot;</td>
</tr>
<tr>
<td>Electronic Arts TGQ</td>
<td>Multimedia Wiki 2008g</td>
<td>&quot;</td>
</tr>
<tr>
<td>Electronic Arts TQI</td>
<td>Multimedia Wiki 2009a</td>
<td>&quot;</td>
</tr>
<tr>
<td>Imagine 3D Data Description (TDDD)</td>
<td>Kirvan 1994</td>
<td></td>
</tr>
<tr>
<td>LWOB</td>
<td>Lightwave 1996</td>
<td>Lightwave 3D Objects</td>
</tr>
<tr>
<td>LWO2</td>
<td>Lightwave 2001a</td>
<td>&quot;</td>
</tr>
<tr>
<td>LWLO</td>
<td>Ferguson 1995</td>
<td>&quot;</td>
</tr>
<tr>
<td>Luxology Modo 3D Object (LXOB)</td>
<td>Luxology 2006</td>
<td></td>
</tr>
<tr>
<td>Pant Shop Pro (PSP)</td>
<td>Jasc 1998</td>
<td></td>
</tr>
<tr>
<td>ProVector 2D Drawing</td>
<td>Cuni9ff and Orr</td>
<td></td>
</tr>
<tr>
<td>ProWrite Document</td>
<td>Bayless 1987</td>
<td></td>
</tr>
<tr>
<td>Blorb Z-machine Resource File</td>
<td>Plotkin</td>
<td>Interactive Fiction</td>
</tr>
<tr>
<td>Quetzal</td>
<td>Frost 1997a, 1997b</td>
<td>Interactive Fiction</td>
</tr>
<tr>
<td>------------------</td>
<td>-------------------</td>
<td>---------------------</td>
</tr>
<tr>
<td>Apple QuickTime</td>
<td>Apple 1996, 2001</td>
<td></td>
</tr>
<tr>
<td>Apple itunes Video</td>
<td>ISO/IEC 14496-14:2003</td>
<td></td>
</tr>
<tr>
<td>JPEG 2000</td>
<td>ISO/IEC 15444-1:2004</td>
<td></td>
</tr>
<tr>
<td>JPEG 2000 Codestream</td>
<td>ISO/IEC JTC 1/SC 29/WG 1 Annex A: Codestream Syntax</td>
<td></td>
</tr>
<tr>
<td>MPEG-4</td>
<td>ISO/IEC 14496-14:2003</td>
<td></td>
</tr>
<tr>
<td>Motion JPEG 2000</td>
<td>ISO/IEC 15444-3, ITU-T T.802</td>
<td></td>
</tr>
<tr>
<td>3GPP Media</td>
<td>ETSI 2012</td>
<td></td>
</tr>
<tr>
<td>3GPP2 Media</td>
<td>3GPP2 2003</td>
<td></td>
</tr>
<tr>
<td>Maya IFF Bitmap</td>
<td>Alias-Wavefront 1999</td>
<td></td>
</tr>
<tr>
<td>Apple Core Audio format</td>
<td>Apple 2005</td>
<td></td>
</tr>
<tr>
<td>American Laser Games (MM)</td>
<td>Multimedia Wiki 2008h</td>
<td></td>
</tr>
<tr>
<td>Smush Animation Format (SAN)</td>
<td>Multimedia Wiki 2006, Multimedia Wiki 2009b</td>
<td></td>
</tr>
<tr>
<td>Structured Data eXchange Format (SDXF)</td>
<td>RFC 3072</td>
<td></td>
</tr>
<tr>
<td>Ogg Vorbis</td>
<td>Pfeiffer 2003, Xiph 2010</td>
<td></td>
</tr>
<tr>
<td>Simple Vector Array Format</td>
<td>Martin 2007</td>
<td></td>
</tr>
<tr>
<td>Nested List File (NLF)</td>
<td>Blank Aspect 2010a, 2010b</td>
<td></td>
</tr>
<tr>
<td>Cinema4D</td>
<td>Losch 1997</td>
<td></td>
</tr>
<tr>
<td>The Sims Interchange File Format</td>
<td>Baum &amp; Noel 2002</td>
<td></td>
</tr>
</tbody>
</table>
Appendix C: Utilities to Pretty Print a Nested List Style Parse Tree

private static int depth = 0;
public static String prettyPrintTree(Tree t) throws Exception {
    return prettyPrintTree(t, (List<String>) null);
}

public static String prettyPrintTree(Tree t, Parser recog) throws Exception {
    String[] ruleNames = recog != null ? recog.getRuleNames() : null;
    List<String> ruleNamesList =
        ruleNames != null ? Arrays.asList(ruleNames) : null;
    return prettyPrintTree(t, ruleNamesList);
}

public static String prettyPrintTree(Tree t, List<String> ruleNames) throws Exception {
    String ruleName = Trees.getNodeText(t, ruleNames);
    StringBuilder buf = new StringBuilder();
    depth++;
    buf.append("(");
    buf.append(ruleName);
    buf.append(' ');
    if (ruleName.startsWith("dt")) {
        // Since t is a Rule Node and it starts with dt,
        // it must be the context of one of the data type rules.
        // All the data type rules have a return field named value.
        Object value = t.getClass().getField("value").get(t);
        if (value instanceof String) {
            String str = (String) value;
            if (depth + str.length() > 0) {
                buf.append(prettyPrintTree(str));
            } else {
                buf.append(str);
            }
        } else {
            // Handle non-string values here...
        }
    }
    buf.append(' )');
    return buf.toString();
}
buf.append(t.getClass().getField("value").get(t));
}
} else if (t.getChildCount() > 0) {
    for (int i = 0; i < t.getChildCount(); i++) {
        buf.append('n');
        for (int j = 0; j < depth; j++) {
            buf.append(" ");
        }
        buf.append(prettyPrintTree(t.getChild(i), ruleNames));
    }
}
buf.append("\n");
depth--;
return buf.toString();
}
public static String prettyPrintTree(String str) throws Exception {
    StringBuilder buf = new StringBuilder();
    int subLength = 70 - depth;
    do {
        buf.append('n');
        for (int j = 0; j < depth; j++) {
            buf.append(" ");
        }
        if (str.length() > subLength) {
            buf.append(str.substring(0, subLength));
            str = str.substring(subLength + 1);
        } else {
            buf.append(str);
        }
    } while (depth + str.length() > 70);
    return buf.toString();
}