Attribute Grammars for Validating

Chunk-based Binary File Formats

William Underwood
Sandra Laib

ICL/ITDSD Working Paper 11-03

July 2011

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

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 US Government. The U.S. government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation heron.
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.

Context-free grammars are used for defining programming languages. There are established methods for using these grammars to check the syntax of programs and to compile programs into assembly or machine language programs. Grammars have also been used to define page description languages and vector graphics languages. However, current technologies for context-free grammars, syntax checkers and compilers are limited to string languages or string data types.

This report describes the extension of context-free grammars from strings to binary files. 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 array grammars. Attribute grammars have been used to define a number of chunk-based file formats. A parser generator has been used with some of these grammars to generate syntax checkers (recognizers) for validating binary file formats. The significance of these results is that with these extensions to core computer science concepts, traditional parser/compiler technologies can be used as a part of a cost effective preservation strategy for binary file formats.
TABLE OF CONTENTS

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

2 BINARY FILE FORMATS ....................................................................................................... 1

3 CONTEXT-FREE GRAMMARS AND ATTRIBUTE GRAMMARS ........................................... 3
   3.1 CONTEXT-FREE GRAMMARS FOR ARRAYS OF DATA TYPES .................................... 3
   3.2 ATTRIBUTE GRAMMARS ............................................................................................... 4
   3.3 SEMANTIC GRAMMAR ................................................................................................. 5

4. GRAMMARS FOR CHUNK-BASED BINARY FILE FORMATS ........................................... 6
   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 RECOGNIZING BINARY FILE FORMATS ............................ 11
   6.1 ANTLR .......................................................................................................................... 11
   6.2 USING ANTLR TO GENERATE A PARSER FOR A BINARY FILE FORMAT .................. 11
       6.2.1 An ANTLR Grammar for an ILBM Chunk-based File Format .............................. 11
       6.2.2 The Lexer and Parser ......................................................................................... 16
   6.3 AN ANTLR GRAMMAR FOR THE RIFF WAVEFORM PCM FILE FORMAT ................ 18

7 RELATED RESEARCH ......................................................................................................... 20

8. CONCLUSION ..................................................................................................................... 21

REFERENCES ......................................................................................................................... 24

APPENDIX A: CHUNK-BASED BINARY FILE FORMATS ............................................................ 33
   A.1 ELECTRONIC ARTS INTERCHANGE FILE FORMAT (IFF) ........................................ 33
       A.1.1 Interleaved Bitmap (ILBM) ................................................................................... 33
       A.1.2 8-bit Sampled Voice (BSVX) .............................................................................. 33
       A.1.3 Cel Animation (ANIM) ....................................................................................... 34
       A.1.4 IFF Formatted Text (FTXT) ................................................................................ 35
       A.1.5 IFF Simple Musical Score (SMUS) .................................................................... 35
       A.1.6 Planar Bitmap (PBM) ......................................................................................... 35
   A.2 STANDARD MIDI FILE FORMAT ................................................................................. 35
   A.3 AUDIO INTERCHANGE FILE FORMAT ......................................................................... 35
   A.4 RESOURCE INTERCHANGE FILE FORMATS ............................................................ 37
       A.4.1 RIFF-Waveform (WAVE) .................................................................................... 37
       A.4.2 WAVEFORMATEX ............................................................................................. 37
       A.4.3 WAVEFORMATEXTENSIBLE ......................................................................... 37
       A.4.4 Broadcast Wave Format ..................................................................................... 37
A.18.9 Electronic Arts TQI .................................................................................................................. 42
A.19 IMAGINE 3D DATA DESCRIPTION (TDDD) .................................................................................. 42
A.20 LIGHTWAVE 3D OBJECTS .............................................................................................................. 42
  A.20.1 LWOB ........................................................................................................................................... 42
  A.20.2 LWO2 ............................................................................................................................................ 42
  A.20.3 LWLO .......................................................................................................................................... 42
A.21 LUXOLOGY MODO 3D OBJECT (LXOB) ......................................................................................... 42
A.22 PAINT SHOP PRO (PSP) ................................................................................................................ 42
A.23 PROVECTOR 2D DRAWING .............................................................................................................. 43
A.24 PROWRITE DOCUMENT ................................................................................................................ 43
A.25 INTERACTIVE FICTION .................................................................................................................. 43
  A.25.1 Blorb Z-machine Resource File ................................................................................................ 43
  A.25.2 Quetzal ......................................................................................................................................... 43
A.26 APPLE QUICKTIME ....................................................................................................................... 43
A.27 MAYA IFF BITMAP ........................................................................................................................ 43
A.28 APPLE CORE AUDIO FORMAT ..................................................................................................... 43
A.29 AMERICAN LASER GAMES (MM) ................................................................................................ 43
A.30 SMUSH ANIMATION FORMAT .................................................................................................... 43
  A.30.1 SAN ............................................................................................................................................. 43
  A.30.2 SNM ........................................................................................................................................... 43
A.31 STRUCTURED DATA EXCHANGE FORMAT (SDXF) ...................................................................... 43
A.32 OGG VORBIS ..................................................................................................................................... 44
A.33 SIMPLE VECTOR ARRAY FORMAT ............................................................................................... 44
A.34 NESTED LIST FILE .......................................................................................................................... 44
A.35 CINEMA 4D ....................................................................................................................................... 44
A.36 THE SIMS INTERCHANGE FILE FORMAT ..................................................................................... 44
List of Figures

Figure 1. File format (Layout) of a RIFF container for a Webp Image ................................................................. 2
Figure 2. ILBM File ham.pic Displayed .......................................................................................................................... 7
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 File Grammar for RIFF WAVEFORM PCM File Format ............................................................... 10
Figure 6. An ANTLR Grammar for the ILBM File Format ............................................................................................. 16
Figure 7. A Java Application for Parsing an ILBM File ................................................................................................. 17
Figure 8. Output of the ILBM Parser Applied to the file BLK.IFF ............................................................................... 18
Figure 9. ANTLR Grammar for the RIFF WAVEFORM PCM File Format ................................................................. 20
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 attribute array grammars for a chunk-based binary file formats are presented. In section 5, recursive descent parsers for these classes of grammars are described. In section 6, experience in using ANTLR, a parser generator for LL(*) string grammars, in generating parsers for recognizing the formats of binary files is discussed. In section 7, related research is described. Section 8 summarizes the results.

2 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. 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>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>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 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 the file formats of 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]. An IFF file itself is one entire IFF chunk. 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, and "tagged data representations" as 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) [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 [Rentz 2008] (Microsoft Excel’s File Format) is chunk-based.

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 for a type of information 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 N → {N ∪ Σ}*

The set of all strings over a vocabulary Σ is denoted by Σ*. The language generated by a context-free grammar G is the set L(G) = {w: w ∈ Σ* and S ⇒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 file grammar \(BG\) 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 \(N \rightarrow \{N \cup \Sigma\}^*\).

Let DataTypes indicate the union of all the values of all datatypes \(D\). The set of all binary files is \(\text{BinaryFiles} = \{\text{Indices} \rightarrow \text{DataTypes}\}\), where \(\text{Indices} = \{1, \ldots, \infty\}\). The language generated by a context-free array (binary file) grammar \(AG\) is the set \(L(AG) = \{w: w \in \text{BinaryFiles} \text{ and } S \Rightarrow w\}\).

By primitive nonterminal is meant a symbol which would be replaced by one or more terminal symbols were an additional production rule applied. In other words, a primitive nonterminal is a nonterminal that appears on the left-hand side of productions which only have one or more terminal symbols on the right. We adopt the notation shown below for primitive nonterminals.

\[
<\text{primitive_nonterminal_symbol type=datatype_name constant=value}>
\]

### 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 GA 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 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 in top-down parsing.

### 3.3 Semantic Grammar

An interpreter for a binary file format that is using a grammar to display or play the contents of a file or to convert it to another file format needs to produce a semantic representation that is appropriate to the task. Natural language understanding components of dialog systems face a similar need to produce a semantic representation that is appropriate to the dialogue task. To address this need, some developers of dialogue systems make use of an approach called semantic grammar [Issar and Ward 1993; Ward and Issar 1994]. A semantic grammar is a CFG in which the non-terminals on the left-hand side of rules correspond to semantic entities. A parser for a semantic grammar produces a labeling of the input string with semantic node labels, for example,
A semantic grammar is just what we need to file in the values of a programming language data structure. Semantic grammars are not applicable to generic systems, but are limited to specific domains. This is not a problem in specifying or parsing a binary file format since a grammar for a binary file format is not meant to apply to all binary file formats, but just to a single format or to a family of file formats.

4. Grammars for Chunk-based Binary File Formats

The attribute grammar that we present will be a mixture 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 field names of the file format spec are represented as nonterminals in the grammar. Constant values in fields are terminals that appear on the right-hand side of productions. Nonterminal fieldnames have data types associated with them.

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 converted to a context-free binary 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 mixture 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 parse tree for the file ham.pic.
Figure 2. ILBM File ham.pic Displayed

<ILBM> → “FORM” <cksize TYPE=UINT32> “ILBM” <BMHD> <CMAP> <CAMG> <BODY>
{ILBM.cksize = csize.value}

<BMHD> → “BMHD” <cksize TYPE=uint32> <BitmapHeader>
<BitMapheader> → <width type=UINT32>
< height type=UINT16>
< xposition type=INT16>
< yposition type=INT16>
< nplanes type=BYTE>
< masking type=BYTE>
< compression type=BYTE>
< reserved type=BYTE>
< transparentcolor type=UINT16>
< xaspect type=BYTE>
< yaspect type=BYTE>
< page width type=INT16>
< pagewidth type=INT16>
< pageheight type=INT16>

<CMAP> → “CMAP” <cksize type=UINT32> <color>[n] {n=cksize.val/3}
<CAMG> → “CAMG” <cksize type=UINT32> <viewmode type=INT32>

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 is 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 \textit{cksize} inserted. To convert this grammar into a binary file format grammar, data types are added for primitive non-terminals.

\[
\text{<WAVE-form>} \rightarrow \text{‘RIFF’ cksize ‘WAVE’} \\
\text{<fmt-ck>} \\
[\text{<fact-ck>}] \\
[\text{<cue-ck>}] \\
[\text{<playlist-ck>}] \\
[\text{<assoc-data-list>}] \\
\text{<wave-data>}
\]

\[
\text{<fmt-ck>} \rightarrow \text{‘fmt’ cksize <common-fields> <wBitsPerSample:WORD>}
\]

\[
\text{<common-fields>} \rightarrow \text{<wFormatTag:WORD value}=0x0001> \\
\text{<wChannels:WORD> \\} \\
\text{<dwSamplesPerSec:DWORD> \\} \\
\text{<dwAvgBytesPerSec:DWORD> \\} \\
\text{<wBlockAlign:WORD>}
\]

\[
\text{<wave-data>} \rightarrow \{\text{<data-ck> | <wave-list>}\}
\]

\[
\text{<data-ck>} \rightarrow \text{data cksize <wave-data>}
\]

\[
\text{<wave-list>} \rightarrow \text{LIST cksize ‘wavl’} \{\text{<data-ck> | <silence-ck>}\}
\]

\[
\text{<silence-ck>} \rightarrow \text{slnt cksize <dwSamples:DWORD>}
\]

\[
\text{<fact-ck>} \rightarrow \text{factxksize <dwFileSize:DWORD>}
\]

\[
\text{<cue-ck>} \rightarrow \text{cue cksize <dwCuePoints:DWORD> <cue-point>+)
\]

\[
\text{<cue-point>} \rightarrow \text{<dwName:DWORD> \\
\text{<dwPosition:DWORD> \\
\text{<fccChunk:FOURCC> \\
\text{<dwChunkStart:DWORD> \\
\text{<dwBlockStart:DWORD> \\
\text{<dwSampleOffset:DWORD>}
\]

\[
\text{<playlist-ck>} \rightarrow \text{‘plst’ cksize <dwSegments:DWORD> <play-segment>)}
\]

\[
\text{<play-segment>} \rightarrow \text{<dwName:DWORD> \\
\text{<dwLength:DWORD> \\
\text{<dwLoops:DWORD}
\]

\[
\text{<assoc-data-list>} \rightarrow \text{‘LIST’ cksize ‘adtl’ <labl-ck><note-ck><ltxt-ck><file-ck>
\]

\[
\text{<labl-ck>} \rightarrow \text{‘labl’ cksize <dwName:DWORD> <data:ZSTR}
\]

\[
\text{<note-ck>} \rightarrow \text{‘note’ cksize <dwName:DWORD> <data:ZSTR}
\]
A recursive descent parser is a top-down parser built from a set of recursive (or non 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. A recursive descent parser for the binary file format grammars defined in this paper does not require a separate lexer to identify the data of a file. The data types are predicted by the grammar rules.

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 use these values in parsing the chunk data or images. The pointers to file addresses that occur in the file formats are 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 corresponding to the nonterminal at that position is executed.

Recursive descent parsers with a pushdown stack can be used with binary file format 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 Recognizing Binary File Formats

What we want is a parser generator whose input is a binary file attribute grammar for a particular binary file format, and whose generated output is the Java source code of a recursive descent parser (with a pushdown stack if needed) 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(*) 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 recognizer for that language. ANTLR supports generating code in a number of the programming languages including C, Java, JavaScript and Python.

ANTLR also generates a 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. Furthermore, the capability to recognize LL(*) grammars is not needed, because the binary file format grammars specified so far do not require lookahead of k symbols.

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

It is possible to use ANTLR to test our binary file format 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 file grammars.

6.2.1 An ANTLR Grammar for an ILBM Chunk-based File Format

Figure 6 shows a specification for the ILBM file format that is input to ANTLR. The output of ANTLR is a lexer and parser for ILBM files.
grammar ilbm;

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 ilbm
ilbm  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: " + $ckSize.value); }
{($ckSize.value > 0)? data
 { 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 uee]{
    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
```java
{$value = (int) getUByteValue($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 6. An ANTLR Grammar for the ILBM File Format.

### 6.2.2 The Lexer and Parser

When provided the file described in the previous section, ANTLR generates the `ilblexer` and `ilbmparser` Java programs. The Java application shown in Figure 6 uses these programs to validate the ILBM file BLK.IFF. It produces the output shown in Figure 7.

```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 7. A Java Application for Parsing an ILBM File

FILE: C:\Users\sl170\Pictures\IFF-ilbm\BLK.IFF
ID: ILBM Size: 2556
ID: BMHD Size: 20
  Width: 200
  Height: 144
  X Position: 0
  Y Position: 0
  Number of Panes: 6
  Masking:0
  Compression:1
  Reserved1:0
  TransParent Color:0
  X Aspect:3
  Y Aspect:3
  Page Width:200
  Page Height:144
ID: CMAP Size: 48
  Color[0]: {0, 0, 0}
  Color[1]: {0, 128, 128}
  Color[2]: {0, 255, 255}
  Color[3]: {255, 0, 255}
  Color[4]: {0, 255, 0}
  Color[5]: {255, 0, 0}
  Color[6]: {0, 0, 255}
  Color[7]: {255, 255, 0}
  Color[8]: {128, 128, 128}
  Color[9]: {192, 192, 192}
  Color[10]: {128, 0, 0}
  Color[11]: {0, 128, 0}
  Color[12]: {128, 128, 0}
  Color[13]: {0, 0, 128}
  Color[14]: {128, 0, 128}
6.3 An ANTLR Grammar for the RIFF Waveform PCM File Format

Figure 9 shows a specification for the WAVEFORM PCM file format that is input to ANTLR. The output of ANTLR is a lexer and parser for WAVEFOR PCM files.

```plaintext
grammar waveform_pcm;

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 contained 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;
    }

    // Default ByteOrder to Little Endian
    String byteOrder = "II";
}

// Rules used to create the Parser

// start symbol waveform_pcm
waveform_pcm returns [boolean result]
```
ckId = fourCC {$ckId.value.equals("RIFF")}?
ckSize = dWordValue
formType = fourCC {$formType.value.equals("WAVE")}?
formatSubChunk
dataSubChunk
{ $result = true; }

fourCC returns [String value]
: BYTE BYTE BYTE BYTE {$value = $text;};

formatSubChunk :
subChunkId = fourCC {$subChunkId.value.equals("fmt ")}?
ckSize = dWordValue
commonFields
pcmSpecificField ;

commonFields :
formatTag = wordValue  // Format category
channels = wordValue   // Number of channels
samplesPerSec = dWordValue  // Sampling rate
avgBytesPerSec = dWordValue  // For buffer estimation
blockAlign = wordValue  // Data block size
;

pcmSpecificField :
bitsPerSample = wordValue  // Sample size
;

dataSubChunk :
subChunkId = fourCC {$subChunkId.value.equals("data")}?
ckSize = dWordValue
waveData[ckSize.value] ;

waveData [long size] :
{(size >= 0)?=> BYTE {size--};}+

// defines the long data type in terms of BYTE terminals
dWordValue returns [long value] throws UnsupportedEncodingException
: b1 = BYTE b2 = BYTE b3 = BYTE b4 = BYTE
{ if (byteOrder.equals("II") ) {
  $value = (long)
  { getUByteValue($b1) |
    getUByteValue($b2) << 8 |
    getUByteValue($b3) << 16 |
    getUByteValue($b4) << 24 };
} else {
  $value = (long)
  { getUByteValue($b1) << 24 |
    getUByteValue($b2) << 16 |}
getUByteValue($b3) << 8 | getUByteValue($b4) );

});//
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
{ if (byteOrder.equals("II") ) {
    $value = (int)
    ( getUByteValue($b1) | getUByteValue($b2) << 8 );
} else {
    $value = (int)
    ( getUByteValue($b1) << 8 | getUByteValue($b2) );
}
}
catch[RecognitionException re]{
    reportError(re);
    recover(input, re); }
catch[UnsupportedEncodingException uee]{
    System.out.println("Unsupported Encoding Exception"); } //Terminal rule used to create the Lexer
BYTE
    : . ;

Figure 9. ANTLR Grammar for the RIFF WAVEFORM PCM File Format

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.

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 recognizers for file formats.

JHOVE (JSTOR/Harvard Object Validation Environment) is an extensible system designed to provide automated and efficient identification and validation of the formats of digital files. JHOVE is a format-specific digital object validation API written in Java. JHOVE supports validation of the following formats: AIFF, ASCII, GIF, HTML, JPEG, JPEG 2000, PDF, TIFF, UTF-8, WAVE, and XML. JHOVE2 is second generation validation environment [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 file grammars and parsers have been constructed for the chunk-based file formats AIFF, JPEG, and WAVE as well as the directory-based format TIFF. However, the research reported herein differs from that of the JHOVE project 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 format were then presented. Recursive descent parsers for this classes of grammars was then described. Experience in using ANTLR, a parser generator for LL(*) 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 JHOVE project in 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 validating the file formats of chunk-based binary files. It remains to be determined whether these attribute grammars based on binary file (array) grammars are adequate to define other families of binary file formats.

To be a practical technology for generating recognizers (validators) for binary file formats, a parser generator is needed that creates from a binary file grammar a recursive descent parser (with pushdown stack, if needed). ANTLR, which accepts LL(*) grammars as input and generates a lexer as well as a parser is too heavyweight a parser generator for this task, and does not have the requisite data types built in.

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 Appendix a. 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 (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 scanner.

Appendix A lists the names of about 75 file formats that belong to the family of “chunk-based” file formats. We have specifications for each of file formats and samples of files with these formats.

These file formats have a similar structure. We have developed binary file 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 ANTLR to generate top-down recursive descent parsers for the file formats defined with these grammars.

The significance of success in this research task is that if binary file formats can be specified with binary file 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 be used for conversion of legacy file formats to current or standard formats. Finally, the same parser generator can be used for generating viewer/players for most file formats. This would increase
the likelihood of preserving and making available into the indefinite future hose digital records encoded in binary file formats.
References


http://caad.arch.ethz.ch/info/maya/manual/

[Amigan 2011] Amigan Software. IFF FORM and Chunk Registry
http://amigan.1emu.net/reg/iff.html


www.aesnashville.org/PDFs/AIFF%20Specs.pdf


[ASF 2001] ASF format version 1.0, reconstruction, January, 2001
http://avifile.sourceforge.net/asf-1.0.htm


http://usa.autodesk.com/adsk/servlet/item?linkID=10809853&id=12272454&siteID=123112


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


27


http://code.google.com/p/ll-parser-generator/source/browse/trunk/Generator/parser3240/Main.java?r=42

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


[Oppenheim 1988] Dave Oppenheim. Standard MIDI Files 0.06, March 1, 1988  
www.ibiblio.org/emusic-l/info-docs-FAQs/MIDI-doc/MIDI-SMF.txt

Software: Practice and Experience. 25, 7 (July), 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/
http://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://xiph.org/vorbis/doc/Vorbis_I_spec.html

http://www.wotsit.org/list.asp?search=okt&button=GO%21
Appendix A: Chunk-based Binary File Formats

This appendix references specifications for about 75 chunk-based binary file formats and discusses some of those specifications.

A.1 Electronic Arts Interchange File Format (IFF)
[Amigan 2011]

Electronic Arts and Commodore-Amiga introduced the chunk-based file format in the specification for the Interchange File Format [Morrison 1985].

A.1.1 Interleaved Bitmap (ILBM)
[Morrison 1986]

This file format was discussed in section 4.1 of this report.

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 file grammar.

Form_8SVX_chunk → “FORM” cksize “8SVX” ckdata
cksise → ULONG

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
In general, the BODY has ctOctave octaves of data. The highest frequency octave comes first, comprising the fewest samples: oneShotHiSamples + repeatHiSamples. Each successive octave contains twice as many samples as the next higher octave but the same number of cycles. The lowest frequency octave comes last with the most samples: \(2^{ctOctave-1} \times (\text{oneShotHiSamples} + \text{repeatHiSamples})\). The number of samples in the BODY chunk is

\[
(2^0 + \ldots + 2^{(ctOctave-1)}) \times (\text{oneShotHiSamples} + \text{repeatHiSamples})
\]

**A.1.3 Cel Animation (ANIM)**

[Sparta 1988]
A.1.4 IFF Formatted Text (FTXT)
[Shaw et al 1985]

A.1.5 IFF Simple Musical Score (SMUS)
[Morrison 1986b]

A.1.6 Planar Bitmap (PBM)
[REWiki 2009]

A.2 Standard Midi File Format
Oppenheim [1988] described the midi file format using the pseudo EBNF notation shown below [Int. MIDI Assoc 2008]. The “length of header data” is 6. Header data consist of three 16-bit integers.

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

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 recasts the specification as a context-free pseudo EBNF grammar. Apple [1991] also specified a compressed version AIFC of the Audio Interchange file.

\[
\text{AudioAIFFfile} \rightarrow \text{"FORM" cksize "AIFF" localchunks}
\]
localhunks -> CommonChunk SoundDataChunk

CommonChunk -> “COMM” cksize numChannels numSampleFrames sampleSize sampleRate

SoundDataChunk -> “SSND” cksize offset blockSize soundData[]

MarkerChunk -> “MARK” cksize Markers[ ]

Markers -> marker markers

Marker -> markerid position markerName

InstrumentChunk -> “INST” cksize baseNote detune lowNote highNote lowVelocity highVelocity gain sustainLoop releaseLoop

MIDIDataChunk -> “MIDI” cksize MIDIData[]

AudioRecordingChunk -> “AESD” cksize AESChannelStatusData[24]

ApplicationSpecificChunk -> “APPL” cksize applicationSignature data[]

CommentsChunk -> “COMT” cksize numComments comments[ ]

Comments -> Comment Comments

Comment -> timeStamp marker count text

The name, author, copyright and annotation chunks are included in the definition of every "EA IFF 85" file. All are text chunks; their data portion consists solely of text. Each of these chunks is optional.

nameprop -> “NAME” size CHAR[size]

copyrightck -> "(c) " size CHAR[size]

authorck -> “AUTH” size CHAR[size]

annotation -> annotationck annotation

annotation -> annotationck

36
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, picture format introduced in (2010) by Google, also uses RIFF as a container.

A.4.1 RIFF-Waveform (WAVE)
[IBM & Microsoft 1991]

This file format was discussed in section 4.2 of this report.

A.4.2 WAVEFORMATEX
[Microsoft 1994]

A.4.3 WAVEFORMATEXTENSIBLE
[Microsoft 2003b]

A.4.4 Broadcast Wave Format
[EBU 2001]

A.4.5 Windows Audio-Video Interleave (AVI)
[Microsoft 2005]

A.4.6 Animated Windows Cursor (ANI)
[Houghtaling 1996]

A.4.7 RIFF MIDIfile (RMID)
[IBM & Microsoft 1991]

A.4.8 Device-Independent Bitmap (RIFF RDIB)
The RDIB format consists of a Windows 3.0 DIB enclosed in a RIFF chunk [IBM & Microsoft 1991].

A.4.9 WebP

WebP is a method of lossy compression for digital images. The degree of compression is adjustable so a user can choose the trade-off between file size and image quality. WebP typically
achieves an average of 39% more compression than JPEG and JPEG 2000, without loss of image
quality. A WebP file consists of VP8 image data and a container based on RIFF. The format of a
WebP file is shown below [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>size of data chunk sarting 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>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>

**A.5 JPEG**

**A.5.1 JPEG/JFIF**

[CCITT 1992], [Hamilton 1992], [Cuturicu and Fromme, 1999]

JPEG/JFIF file format:

- header (2 bytes): $ff, $d8 (SOI) (these two identify a JPEG/JFIF file)
- for JFIF files, an APP0 segment is immediately following the SOI marker, see below
- any number of "segments" (similar to IFF chunks), see below
- trailer (2 bytes): $ff, $d9 (EOI)

Segment format:

- header (4 bytes):
  - $ff identifies segment
  - n type of segment (one byte)
  - sh, sl size of the segment, including these two bytes, but not
    including the $ff and the type byte. Note, not Intel order:
    high byte first, low byte last!

**A.5.2 JPEG/Exif**

[JEIDA 2002]
A.5.3 JPEG 2000

“Every marker is two bytes long. The first byte consists of a single 0xFF byte. The second byte denotes the specific marker and can have any value in the range 0x01 to 0xFE. Many of these markers are already used in ITU-T Rec. T.81 | ISO/IEC 10918-1 and ITU-T Rec. T.84 | ISO/IEC 10918-3 and shall be regarded as reserve unless specifically used. A marker segment includes a marker and associated parameters, called marker parameters. In every marker segment the first two bytes after the marker shall be an unsigned big endian integer value that denotes the length in bytes of the marker parameters (including two bytes of this length parameter but not the two bytes of the marker itself).” [ISO 2000, page 13]

A.5.4 JPEG 2000 Codestream


A.6 Advanced Systems Format

Microsoft’s Advanced Systems Format (ASF) [2001] 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).

A.6.1 Windows Media Audio 7-9

Microsoft 2004]

A.6.2 Windows Media Video 7-9.1

[Microsoft 2004]

A.7 Portable Network Graphics

[ISO 15948:2003]

PNG file -> PNGSignature headerck paletteck
headerck -> IHDR cksize Width Height BitDepth ColourType
                  Compressionmethod Filtermethod Interlacementmethod

paletteck -> PLTE cksize entries

entries -> entry entries

entry -> red green blue
A.7.1 Multiple-image Network Graphics
   [Randers-Pehrson 2001]

A.7.2 JPEG Network Graphics
   [Randers-Pehrson 2001]

A.8 Binary Interchange File Format
   [Rentz 2008]

A.9 3D Studio

A.9.1 3D Studio File Format – 3ds
   [Autodesk 1997]

A.9.2 3D Studio Material Library Files
   [Fercoq 1996]

A.10 Anim8or
   [Glanville 2003]

A.11 Animator
   [Crazy Talk]

A.11.1 Animator Cel
   [Crazy Talk]

A.11.2 Animator Pro Fli
   [Haaland]

A.11.3 Animator Pro fLC

A.11.1 Animator Cel
   [Crazy Talk]

A.11.4 Animator Pro PIC
   [Maischein]
A.12 Audio Manager Module  
[Foo 1995]

A.13 Autodesk Drawing Interchange Files (DXF) (Binary)  
[Autodesk 2009]

A.14 Caligari TrueSpace Scene/Object (Binary)  
[Caligari 1998]

A.15 CorelDRAW Vector Graphics File  
(Versions 3.0 – X3)  
[Novikov and Fillipov]

A.16 Corel Metafile Exchange Image  
[Corel 1998]

A.17 Music Modules

A.17.1 DigiTrekker Module Format  
[Beham 1995]

A.17.2 Graoumf Tracker modules  
[Soras 1995]

A.17.3 Oktalyzer  
[Zappa 1994]

A.18 EA Multimedia Game File Formats  
[Anisimovsky 2002a]

A.18.1 Electronic Arts Music (.asf)  
[Anisimovsky 2002b]

A.18.2 Electronic Arts CMV  
[MultimediaWiki 2008a]

A.18.3 Electronic Arts DCT  
[MultimediaWiki 2008b]
A.18.4 Electronic Arts MAD
[MultimediaWiki 2008c]

A.18.5 Electronic Arts MPC
[MultimediaWiki 2008d]

A.18.6 Electronic Arts VP7
[MultimediaWiki 2008e]

A.18.7 Electronic Arts TGV
[MultimediaWiki 2008f]

A.18.8 Electronic Arts TGQ
[MultimediaWiki 2008g]

A.18.9 Electronic Arts TQI
[MultimediaWiki 2009]

A.19 Imagine 3D Data Description (TDDD)
[Kirvan 1994]

A.20 Lightwave 3D Objects

A.20.1 LWOB
[LightWave 1996]

A.20.2 LWO2
[Lightwave 2001a]

A.20.3 LWLO
[Ferguson 1995]

A.21 Luxology Modo 3D Object (LXOB)
[Luxology 2006]

A.22 Paint Shop Pro (PSP)
[Jasc 1998]
A.23 ProVector 2D Drawing
   [Cunniff and Orr]

A.24 ProWrite Document
   [Bayless 1987]

A.25 InteractiveFiction

A.25.1 Blorb Z-machine Resource File
   [Plotkin]

A.25.2 Quetzal
   [Frost 1997a, 1997b]

A.26 Apple QuickTime
   [Apple 2007]

A.27 Maya IFF Bitmap
   [Alias-wavefront 1999]

A.28 Apple Core Audio Format
   [Apple 2005]

A.29 American Laser Games (MM)
   [MultimediaWiki 2008h]

A.30 Smush Animation Format

A.30.1 SAN
   [MultimediaWiki 2006]

A.30.2 SNM
   [MultimediaWiki 2009b]

A.31 Structured Data eXchange Format (SDXF)
   [RFC 3072]
A.32 Ogg Vorbis  
[Pfeiffer 2003] [Xiph 2010]

A.33 Simple Vector Array Format  
[Martin 2007]

A.34 Nested List File

Nested List File (NLF) is a general, chunk-based file metaformat for binary data that is used by Onda as the metaformat for its compressed files. [Blank Aspect 2010a 2010b]

A.35 Cinema 4D  
[Losch 1997]

A.36 The Sims Interchange File Format

Jamie Doornbos is the lead programmer of Maxis' "The Sims". [Baum & Noel 2002]