The International Journal of Digital Curation
Volume 7, Issue 1 | 2012

Grammar-Based Specification and Parsing of Binary File Formats

William Underwood,

Principal Research Scientist,

Georgia Tech Research Institute

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 paper 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. This attribute grammar has been used to define a number of chunk-based and directory-based binary file formats. A parser generator has been used with some of these grammars to generate syntax checkers (recognizers) for validating binary file formats. Among the potential benefits of an attribute grammar-based approach to specification and parsing of binary file formats is that attribute grammars not only support format validation, but support generation of error messages during validation of format, validation of semantic constraints, attribute value extraction (characterization), generation of viewers or players for file formats, and conversion to current or standard file formats. The significance of these results is that with these extensions to core computer science concepts, traditional parser/compiler technologies can potentially be used as a part of a general, cost effective curation strategy for binary file formats.

Introduction

Automated tools are required for identifying and validating the formats of the huge number of files ingested into digital data and record archives. Invalid file formats can arise from data transmission errors, media deterioration, poor quality software tools, or as a result of intentional corruption. Automated tools are also needed for viewing or 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.

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. The next section discusses the traditional approach to specifying file formats and two of the major families of binary file formats. Then extensions to the concepts of context-free grammars and attribute grammars that enable the specification of binary file formats are described. An example of an attribute array grammar for a chunk-based binary file format is then presented. Recursive descent parsers for these classes of grammars are then described. Experience in using ANTLR, a parser generator for LL(k) string grammars, in generating parsers for recognizing the formats of binary files is discussed. Finally, related research is described and results are summarized.

Binary File Formats

In a binary file format specification, 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. 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.

Families of Binary File Formats

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. These two families do not exhaust the possible file structures. For instance, there are executable binary file formats and file header-body file formats that have different file structures and will be considered in the future.

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.

The Audio Interchange File Format (AIFF) developed by Apple computer in 1988 was based on Electronic Arts Interchange File Format. The Core Audio Format (CAF), Apple's replacement format for AIFF, remains a chunk-based format (Apple, 2005). The Resource Interchange File Format (RIFF) introduced in 1991 by IBM and Microsoft (1991) is also based on Electronic Arts' Interchange File Format. The Microsoft container formats, such as Audio-Video Interleave (AVI) and Waveform PCM (WAV), use RIFF as their basis. WebP (Google, 2010), a picture format recently introduced by Google, 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 (Rentz, 2008) (Microsoft Excels's File Format) is chunk-based.

File format specifications for chunk-based formats may use terms other than chunks in describing the format, for instance, "atoms" in QuickTime/MP4, "segments" in JPEG, and "tagged data representations" as in AutoCAD DXF. Underwood and Laib (2011) have identified more than 75 file formats that are chunk-based binary file formats.

Another family of binary file formats are directory-based. A directory-based file format consists of one or more directory tables, which contains 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.

Context-Free Grammars and Attribute Grammars

Context-free grammars have been widely used to define the lexical and syntactic structure of programming languages. The concept of a context-free grammar for string languages can be extended to binary files in the following manner.

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 various types.

We define a *context-free array (binary file) grammar* AG as a quintuple <N, D, Σ, S, P> where:

N is a finite set of non-terminal symbols,

D is a set of data types,

Σ 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 $\mathit{BinaryFiles} = [\mathit{Indices} \rightarrow \mathit{DataTypes}]$, where $\mathit{Indices} = \{1, \ldots, \infty\}$. The language generated by a context-free array (binary file) grammar AG is the set $L(AG) = \{w : w \in \text{BinaryFiles and } S \Rightarrow w\}$.

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.

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,

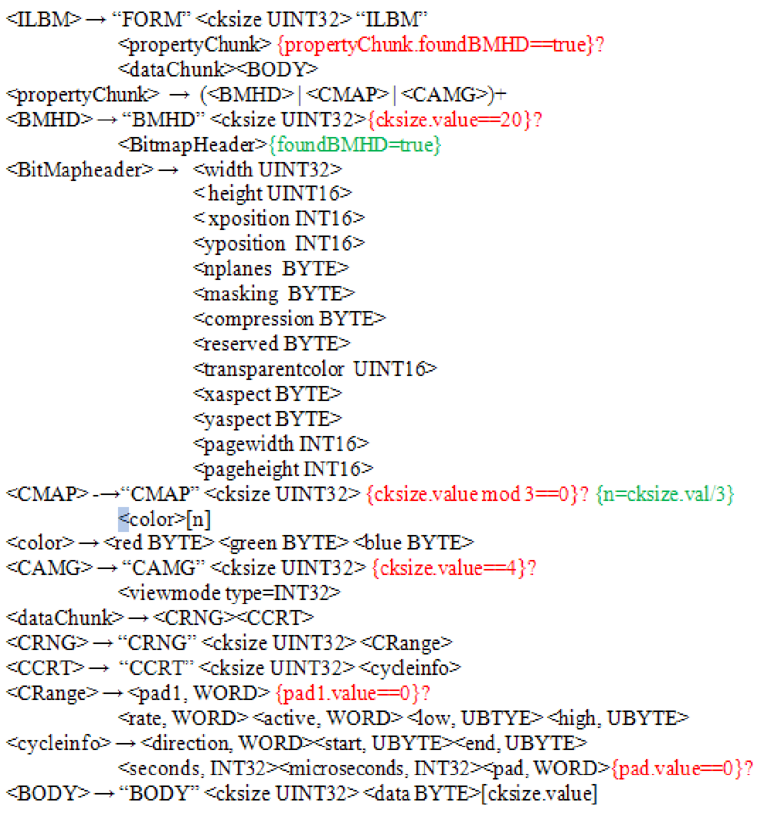
A associates each grammar symbol $X \in (N \cup \Sigma)$ with a set of attributes, and

AR associates each production $R \in P$ with a set of attribute computation rules and conditional attribute rules.

Attribute Grammars for Binary File Formats

Figure 1 shows an attribute grammar used to specify the ILBM chunk-based binary file format (Morrison, 1986). The rules of the grammar are in an Extended BNF notation. The context-free rules are in a black font. The computation attribute rules are in a red font. The conditional attribute rules are in a green font.

The initial symbol of the grammar is <ILBM>. The interpretation of the first rule is: The first four 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 four bytes contain the characters "ILBM". This must be followed by a propertyChunk, a dataChunk and a BODY chunk.

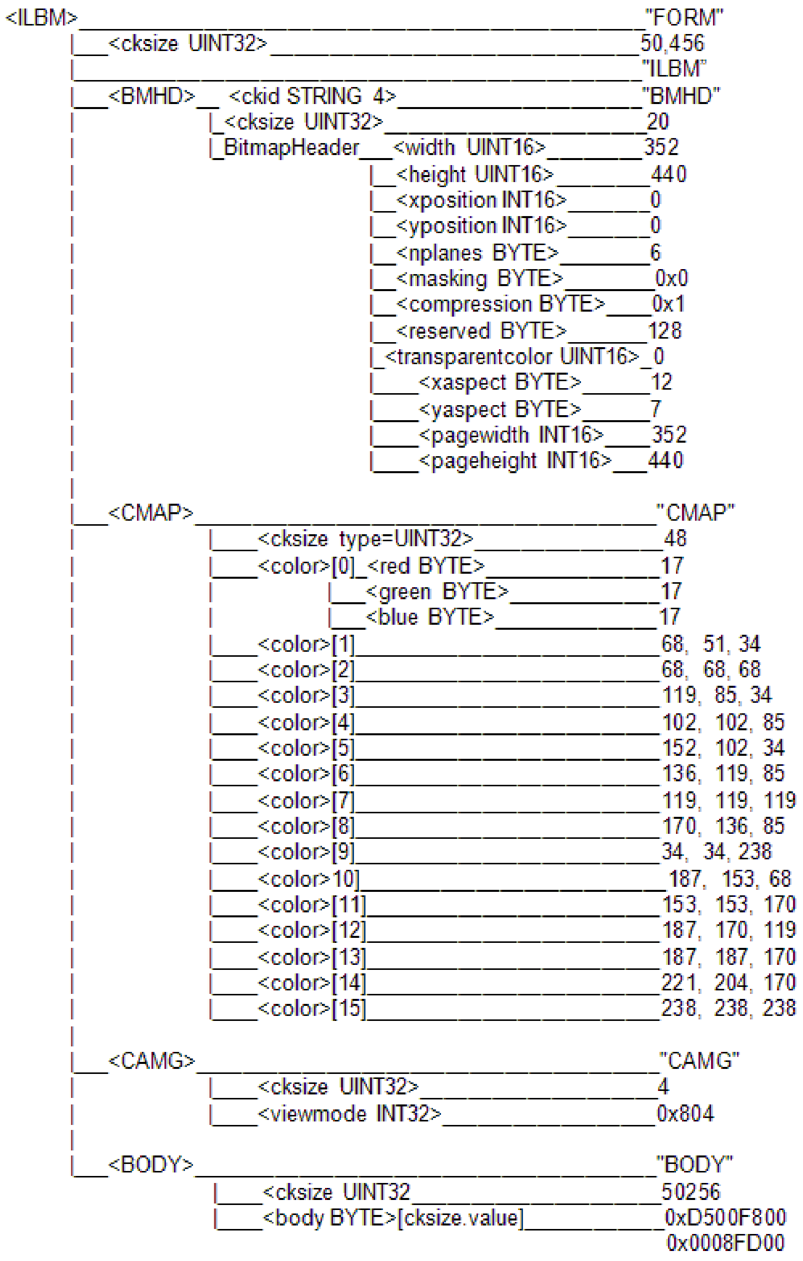


```
<ILBM> → "FORM" <cksize UINT32> "ILBM"
            <propertyChunk> {propertyChunk.foundBMHD==true}?
            <dataChunk><BODY>
<propertyChunk>  →  (<BMHD> | <CMAP> | <CAMG>)+
<BMHD> → "BMHD" <cksize UINT32>{cksize.value==20}?
            <BitmapHeader>{foundBMHD=true}
<BitMapheader> →  <width UINT32>
                  < height UINT16>
                  < xposition INT16>
                  <yposition  INT16>
                  <nplanes  BYTE>
                  <masking  BYTE>
                  <compression BYTE>
                  <reserved BYTE>
                  <transparentcolor  UINT16>
                  <xaspect BYTE>
                  <yaspect BYTE>
                  <pagewidth INT16>
                  <pageheight INT16>
<CMAP> -→"CMAP" <cksize UINT32> {cksize.value mod 3==0}? {n=cksize.val/3}
            <color>[n]
<color> → <red BYTE> <green BYTE> <blue BYTE>
<CAMG> → "CAMG" <cksize UINT32> {cksize.value==4}?
            <viewmode type=INT32>
<dataChunk> → <CRNG><CCRT>
<CRNG> → "CRNG" <cksize UINT32> <CRange>
<CCRT> →  "CCRT" <cksize UINT32> <cycleinfo>
<CRange> → <pad1, WORD> {pad1.value==0}?
            <rate, WORD> <active, WORD> <low, UBTYE> <high, UBYTE>
<cycleinfo> → <direction, WORD><start, UBYTE><end, UBYTE>
            <seconds, INT32><microseconds, INT32><pad, WORD>{pad.value==0}?
<BODY> → "BODY" <cksize UINT32> <data BYTE>[cksize.value]
```

Figure 1. An attribute grammar for the ILBM chunk-based file format.

Figure 2 shows the parse tree for an actual ILBM file as defined by the binary array attribute grammar for the ILBM file format. This parse tree was manually constructed, but a parse tree with similar structure could be generated by the parser by including computational attribute rules that construct the parse tree during a parse.

The root of the parse tree is the start symbol of the grammar <ILBM>. The unsigned integer indicating the chunk size has decimal value 50,456. The chunk size of the BMHD chunk is decimal 20. The BitmapHeader data chunk contains metadata about the bitmap stored in the BODY chunk. The colour palette is stored in the CMAP chunk.



```
<ILBM>___________________________________________"FORM"
     |___<cksize UINT32>_____________________________50,456
     |_______________________________________________"ILBM"
     |___<BMHD>__ <ckid STRING 4>____________________"BMHD"
     |           |_<cksize UINT32>____________________20
     |           |_BitmapHeader___<width UINT16>_______352
     |                           |__<height UINT16>_______440
     |                           |__<xposition INT16>______0
     |                           |__<yposition INT16>______0
     |                           |__<nplanes BYTE>_______6
     |                           |__<masking BYTE>________0x0
     |                           |__<compression BYTE>____0x1
     |                           |__<reserved BYTE>_______128
     |                           |_<transparentcolor UINT16>_0
     |                           |____<xaspect BYTE>______12
     |                           |____<yaspect BYTE>______7
     |                           |____<pagewidth INT16>____352
     |                           |____<pageheight INT16>___440
     |
     |___<CMAP>______________________________________"CMAP"
     |           |____<cksize type=UINT32>_______________48
     |           |____<color>[0]_<red BYTE>______________17
     |           |               |___<green BYTE>_____________17
     |           |               |___<blue BYTE>______________17
     |           |____<color>[1]__________________________68,  51, 34
     |           |____<color>[2]__________________________68,  68, 68
     |           |____<color>[3]__________________________119,  85, 34
     |           |____<color>[4]__________________________102,  102, 85
     |           |____<color>[5]__________________________152,  102, 34
     |           |____<color>[6]__________________________136,  119, 85
     |           |____<color>[7]__________________________119,  119, 119
     |           |____<color>[8]__________________________170,  136, 85
     |           |____<color>[9]__________________________34,  34, 238
     |           |____<color>10]__________________________187,  153, 68
     |           |____<color>[11]_________________________153,  153, 170
     |           |____<color>[12]_________________________187,  170, 119
     |           |____<color>[13]_________________________187,  187, 170
     |           |____<color>[14]_________________________221,  204, 170
     |           |____<color>[15]_________________________238,  238, 238
     |
     |___<CAMG>______________________________________"CAMG"
     |           |____<cksize UINT32>____________________4
     |           |____<viewmode INT32>___________________0x804
     |
     |___<BODY>______________________________________"BODY"
                 |____<cksize UINT32_____________________50256
                 |____<body BYTE>[cksize.value]___________0xD500F800
                                                          0x0008FD00
```

Figure 2. A parse tree for an ILBM file.

A Recursive Descent Parser for Binary File Format Grammars

A recursive descent parser is a top-down parser built from a set of recursive procedures (or a non-recursive equivalent), where each such 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 do not require a separate lexical scanner to identify the data of a file. The data types are predicted by the grammar rules.

The semantic rules of an attribute grammar can be included in the recursive descent parser to check for context-sensitive aspects of the grammar, such as array size or image dimensions, and use these values in parsing the arrays or images. The pointers to file addresses that occur in the file formats are handled by pushing the addresses encountered in the file onto a pushdown stack, along with the expected nonterminal expected at that location. When the parser exhausts the procedures implementing production rules, the topmost address and nonterminal on the pushdown stack are popped and the procedure corresponding to that nonterminal 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) for a file format is a parser that reads a file and generates error messages if the file does not conform to the syntax specified by the grammar.

Generating Parsers for Binary Formats with ANTLR

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.

ANTLR (ANother Tool for Language Recognition) is a parser generator that uses LL(k) parsing (Parr, 1995, 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 lexical scanner from lexical rules in the grammar. A lexical scanner is a program that converts a sequence of characters into tokens. A lexical scanner 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(k) grammars is not needed, because the binary file format grammars specified so far do not require lookahead of *k* symbols.

However, it has been possible to use ANTLR to test our binary file format grammars in recognizing the chunk-based and directory-based binary file formats (Underwood & Laib, 2011). This was accomplished by writing a lexical scanner 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., int16, int32, etc. This is effective, if somewhat 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.

Validation of File Formats

File format identification ascertains the purported format of a file. A capability to validate binary file formats assumes that the file format of a file has already been identified. 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 chunk-based and directory-based file formats. We also have file signature tests (magic tests) for these file types. The file format parser corresponding to the identified file format is applied, and if there are no errors, metadata corresponding to the file is updated to indicate the date that the file format was validated.

The file format validator should indicate the features of the file format that do not comply with the specification. The rules of the attribute grammar can be used to generate error messages similar to those generated by a programming language compiler. This includes such errors as field values that are out of range and structural errors. The rules of the attribute grammar can also be used to extract attribute (field) values for further description of the file.

Semantic Correctness

Validity of a file format does not guarantee semantic correctness of the digital object represented by the file format. In particular, there may be semantic constraints between the fields of a file that are not satisfied, for instance, between the fields of a database table. The semantic rules of an attribute grammar can be used to check for satisfaction of some of these constraints.

Related Research

Researchers have developed a number of data description languages for accessing the contents of files and validating file formats. ASN.1 (Abstract Syntax Notation One)[1] is an international standard whose purpose is to specify the format of data used in telecommunication protocols. It has seen limited use in the specification of ad hoc scientific data formats.

[1] ISO/IEC 8824-1:2008. Information Technology – Abstract Syntax Notation One (ASN.1): Specification of the basic notation

EAST is a data description language developed by the Consultative Committee for Space Data Systems (2010). The Data Entity Description 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. PADS (Processing Ad hoc Data Sources) was designed for use with ad hoc scientific data sets (Fisher & Grubner, 2005). A PADS compiler compiles a PADS 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 (Powell, Beckerle & Hanson, 2011). 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. The binary file format grammar described in this paper most closely resembles ANS.1 and DFDL. However, the binary file grammar presented 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.

Conclusion

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. Two of the major families of binary file formats, chunk-based and directory based, were described. Then extensions to the concepts of context-free grammars and attribute grammars that enable the specification of binary file formats were described. An example of an attribute array grammar for a chunk-based binary file format was then presented. Recursive descent parsers for these classes of grammars were 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 and the JHOVE project in creating Java-based tools for validating file formats were described.

It is concluded that it is possible to extend context-free grammars to the specification of chunk-based and directory-based binary file formats. Furthermore, these grammars can be used with recursive descent parsers (some requiring pushdown stacks) for validating the file formats of chunk-based and directory-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 a recursive descent parser (with pushdown stack, if needed) from a binary file grammar. ANTLR, which accepts LL(k) grammars as input and generates a lexical scanner, as well as a parser is too heavyweight a parser generator for this task and does not have the requisite data types built in.

Among the potential benefits of an attribute grammar-based approach to specification and parsing of binary file formats is that attribute grammars support not only an approach to format validation, but to generation of error messages during validation of format, validation of semantic constraints, attribute value extraction (characterization), generation of viewers or players for file formats, and conversion to current or standard file formats. The significance of success in this research task is that if binary file formats can be specified with binary file grammars, then only one parser generator is needed to generate the many parsers needed for validating many binary file formats. Similarly, a single compiler-compiler could be used for conversion of legacy file formats to current or standard formats. Finally, the same compiler-compiler could be used for generating viewer/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.

Acknowledgements

This research project is sponsored by the Applied Research Division of the National Archives and Records Administration, and the Army Research Laboratory under Army Research Office Cooperative Agreement W911NF-10-2-0030.

References

Apple Computer. (2005). *Core Audio Format Specification*. Retrieved from http://developer.apple.com/library/mac/#documentation/MusicAudio/Reference/CAFSpec/CAF_spec/CAF_spec.html#//apple_ref/doc/uid/TP40001862-CH210-TPXREF101

Back, G. (2002). A specification and scripting language for binary data. *Generative Programming and Component Engineering*, *2487*, 66-77.

California Digital Library. (2011). *JHOVE2 User's Guide.* Retrieved from https://bytebucket.org/jhove2/main/wiki/documents/JHOVE2-Users-Guide_20110222.pdf

Consultative Committee for Space Data Systems. (2010). *The data description language EAST specification (CCSD0010).* Retrieved from http://public.ccsds.org/publications/archive/644x0b3.pdf.

Dunckley, M. Rankin, S., Conway, E. & Giaretta, D. (2007). The use of file description languages for file format identification and validation. Paper presented at the PV 2007 Conference: Ensuring the Long-Term Preservation and Value Adding to Scientific and Technical Data, Oberpfaffenhofen/Munich, Germany. Retrieved from http://epubs.cclrc.ac.uk/work-details?w=50089

Fisher, K. & Gruber, R. (2005). PADS: A domain specific language for processing ad hoc data. Paper presented at the ACM Conference on Programming language Design and Implementation. ACM Press. Retrieved from http://www.padsproj.org/papers/pldi.pdf

Google. (2010). *WebP RIFF Container*. Retrieved from http://code.google.com/speed/webp/docs/riff_container.html

IBM & Microsoft. (1991). *Multimedia programming interface and data* specifications 1.0. Retrieved from http://www-mmsp.ece.mcgill.ca/documents/audioformats/wave/Docs/riffmci.pdf

Knuth, D.E. (1968). Semantics of context-free grammars. *Mathematical Systems Theory*, *2*, 127-145.

Knuth, D.E. (1971). Semantics of context-free grammars (corrections). *Mathematical Systems Theory*, *5*, 95-96.

Microsoft. (2004). *Advanced Systems Format (ASF) specification*. Retrieved from www.microsoft.com/windows/windowsmedia/forpros/format/asfspec.aspx

Morrison, J. (1985). *"EA IFF 85" Standard for Interchange Format Files.* Electronic Arts. Retrieved from www.martinreddy.net/gfx/2d/IFF.txt

Morrison, J. (1986). *"ILBM" IFF Interleaved Bitmap.* Electronic Arts. Retrieved from www.fine-view.com/jp/labs/doc/ilbm.txt

Parr, T.J. & Quong, R.W. (1995). ANTLR: A predicated-LL(k) parser generator. *Software: Practice and Experience 25*(7).

Parr, T. (2007). *The definitive ANTLR reference: Building domain-specific languages*. Raleigh: Pragmatic.

Powell, A.W., Beckerle, M.J., & Hanson, S.M. (2011). *Data Format Description Language (DFDL) v1.0 Specification.* Report of the Open Grid Forum. Retrieved from www.ogf.org/documents/GFD.174.pdf

Rentz, D. (2008). *Documentation of the Microsoft Excel File Format, Excel Versions 2, 3, 4, 5, 95, 97, 2000, XP, 2003.* OpenOffice.org. Retrieved from http://sc.openoffice.org/excelfileformat.pdf

Underwood, W. (2009). *Extensions of the UNIX file Command and Magic File for File Type Identification.* Technical Report ITTL/CSITD 09-02 Georgia Tech Research Institute. Retrieved from http://perpos.gtri.gatech.edu/publications/TR%2009-02.pdf

Underwood, W. & Laib, S. (2011). *Attribute grammars for validating chunk-based binary file formats*. ICL/ITDSD Working Paper, Georgia Tech Research Institute (GTRI), USA. Retrieved from http://perpos.gtri.gatech.edu/publications/index.htm